Microsoft

Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence

Douglas McMurtry
Andrew Oakley
John Sharp
Mani Subramanian
Hanz Zhang

patterns & practices

Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence

Douglas McMurtry
Andrew Oakley
John Sharp
Mani Subramanian
Hanz Zhang

ISBN: 978-1-62114-030-6

Contents

Preface

All applications use data. Even the simplest "Hello, World!" program displays data. Most applications also need to store data somewhere. In the world of business systems, this data might be a list of the products and services that the business sells, together with information about the customers who have purchased these items, and the details of the various orders that they have placed.

In theory, a business system can store information by using any convenient data storage technology, but since the early 1970s a sizable majority of solutions have based their storage requirements around a relational database. Relational database technology was firmly grounded in the single-site enterprise solutions of the 1970s, 80s, and 90s. However, since the turn of the millennium, many large organizations now look to building distributed solutions using the Internet as their communications infrastructure; an environment that was never envisaged back in the 1970s. Additionally, advances in computing power and disk storage technology now enable organizations to gather, store, and process much more information that was possible 30 or 40 years ago. In this environment issues surrounding scalability, throughput, responsiveness, and the flexibility to handle complex, dynamic relationships between data items have become paramount. Many relational database management systems do not sit comfortably in this setting, and struggle in the face of ever expanding connectivity and potentially massive datasets. This situation has led to several significant large organizations forgoing relational databases and instead implementing their own custom solutions, specifically designed to meet these requirements. A number of these database solutions have been openly documented, and the software for them is available in the public domain; organizations can freely download the software and integrate it into their own systems. Collectively, these solutions are referred to as "NoSQL" databases.

There is an ever-expanding range of NoSQL databases currently available, covering a variety of different data formats and structures. In most cases, a NoSQL database is designed to provide efficient data storage and access to support a specific pattern of use, such as managing highly-connected networks of objects with complex interrelationships, storing and accessing documents with many queryable fields, or providing fast access to blobs containing almost anything. As a result, an organization is likely to find that no single NoSQL database meets all of its requirements, and it might be necessary to incorporate more than one such database into their solutions. Organizations seeking to use a NoSQL database are therefore faced with a twofold challenge:

- Which NoSQL database(s) best meet(s) the needs of the organization?
- How does an organization integrate a NoSQL database into its solutions?

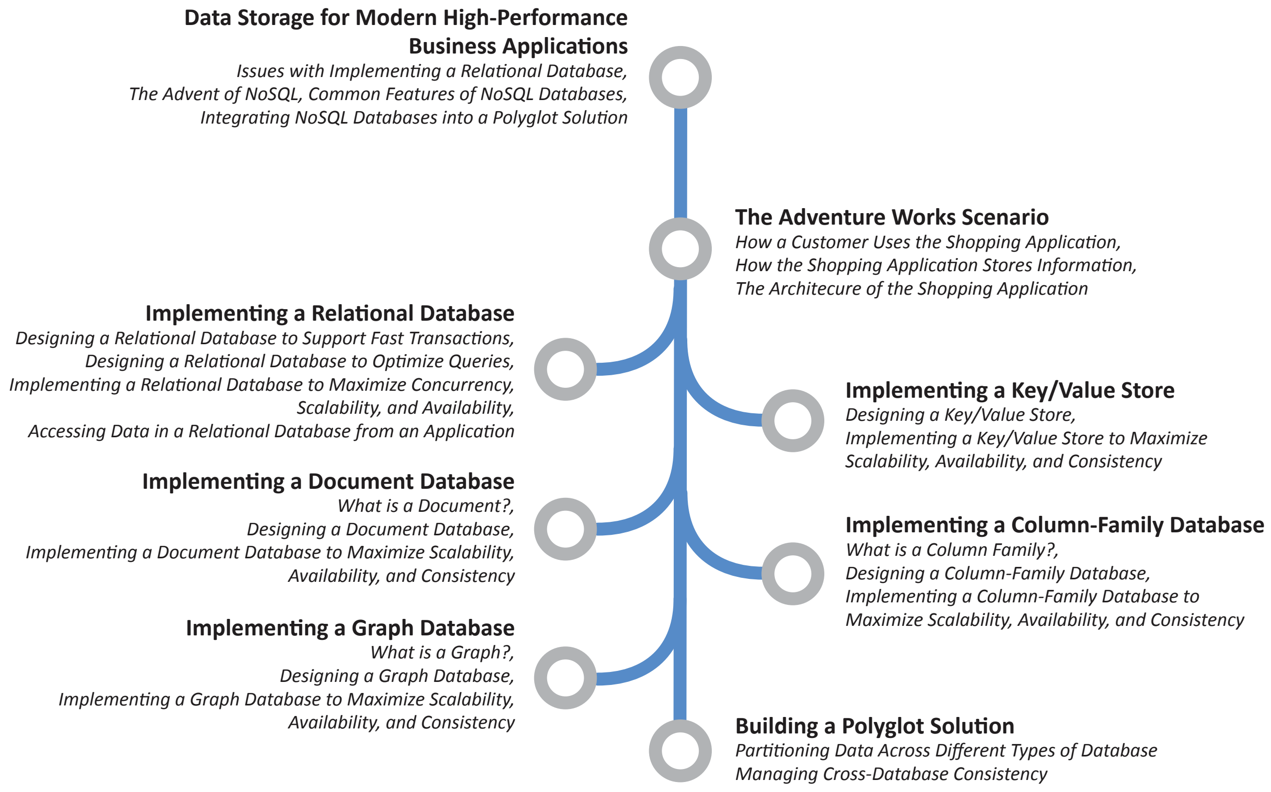
This guide focuses on the most common types of NoSQL database currently available, describes the situations for which they are most suited, and shows examples of how you might incorporate them into a business application. The guide summarizes the experiences of a fictitious organization named Adventure Works, who implemented a solution that comprised an assortment of different databases. The architecture of the solution is based on a web service that connects to each of the databases. The rationale that the developers at Adventure Works adopted was to select the most appropriate data storage technology that met the specific business requirements of each part of the application. The result is a polyglot solution, with various parts of the data held in different databases, but combined together by the logic in the web service.

Who This Book Is For

This book is intended for architects, developers, and information technology professionals who design, build, or maintain large-scale applications and services that store data in a database, and that have to handle requests from a large number of users. Much of the guidance in this book is intended to be generic and apply to any operating system and NoSQL database implementation. However, many of the examples shown are based on Microsoft technologies. To understand the sample code provided with this book, you should be familiar with the Microsoft .NET Framework, the Microsoft Visual Studio development system, ASP.NET MVC, Microsoft SQL Server, and the Microsoft Visual C# development language. If you wish to run the sample application in the cloud using Windows Azure, familiarity with the Windows Azure Table service is also useful. Additionally, to configure and run the polyglot version of the sample application, you should be familiar with the NoSQL technologies used by this application. For more information, see the section "What You Need to Use the Code" later in this preface.

Why This Book Is Pertinent Now

Other than being nonrelational, there is currently no formal definition of what constitutes a NoSQL database. Furthermore, there are currently no standard APIs in existence for applications to use to interact with a NoSQL database; each NoSQL database offers its own library (or sometimes a set of libraries). Integrating different NoSQL databases into a single seamless, extensible solution is a challenge facing an increasing number of developers. For example, developers need to understand how to ensure consistency across different databases, and how to maintain the integrity of the relationships between data held in different databases. There are also many occasions when a relational database is a better solution than a NoSQL database. This guide aims to help you understand how to design solutions that take advantage of the most appropriate database technology.


Data Storage for Modern High-Performance Business Applications
Issues with Implementing a Relational Database, The Advent of NoSQL, Common Features of NoSQL Databases, Integrating NoSQL Databases into a Polyglot Solution
The Adventure Works Scenario
How a Customer Uses the Shopping Application, How the Shopping Application Stores Information, The Architecure of the Shopping Application
Implementing a Relational Database
Designing a Relational Database to Support Fast Transactions, Designing a Relational Database to Optimize Queries, Implementing a Relational Database to Maximize Concurrency, Scalability, and Availability, Accessing Data in a Relational Database from an Application
Implementing a Key/Value Store
Designing a Key/Value Store, Implementing a Key/Value Store to Maximize Scalability, Availability, and Consistency
Implementing a Document Database
What is a Document?, Designing a Document Database, Implementing a Document Database to Maximize Scalability, Availability, and Consistency
Implementing a Column-Family Database
What is a Column Family?, Designing a Column-Family Database, Implementing a Column-Family Database to Maximize Scalability, Availability, and Consistency
Implementing a Graph Database
What is a Graph?, Designing a Graph Database, Implementing a Graph Database to Maximize Scalability, Availability, and Consistency
Building a Polyglot Solution
Partitioning Data Across Different Types of Database Managing Cross-Database Consistency

How This Book Is Structured

This is the road map of the guide.

Chapter	Summary
Chapter 1, "Data Storage for Modern High-Performance Business Applications"	This chapter provides an overview of the common challenges that organizations have encountered with the relational model and relational database technology, and discusses how NoSQL databases can help to address these challenges.
Chapter 2, "The Adventure Works Scenario"	This chapter describes the business requirements of the Adventure Works Shopping application, and summarizes the architecture of the solution that Adventure Works built, based on web services and a combination of SQL and NoSQL databases.
Chapter 3, "Implementing a Relational Database"	This chapter describes how to design a relational database, and summarizes the conflicting requirements that you may need to consider to implement a database that supports efficient transactions and fast queries.
Chapter 4, "Implementing a Key/Value Data Store"	This chapter describes the principles that underpin most large-scale key/value stores, and summarizes the concerns that you should address to use a key/value store for saving and querying data quickly and efficiently.
Chapter 5, "Implementing a Document Database"	This chapter describes the primary features of common document databases, and summarizes how you can design documents that take best advantage of these features to store and retrieve structured information in an optimal manner.
Chapter 6, "Implementing a Column-Family Database"	This chapter provides information on how to design the schema for a column-family database to best meet the needs of applications that perform column-centric queries.
Chapter 7, "Implementing a Graph Database"	This chapter describes how to design a graph database to support the analytical processing performed by an application.
Chapter 8, "Building a Polyglot Solution"	This chapter focuses on the challenges faced by developers building a business solution with the data spanning different types of databases. It summarizes the major concerns of a polyglot solution, and describes some strategies for addressing these concerns.

This guide also includes appendices that describe how the sample application works, and why the developers at Adventure Works selected the various databases that the application uses.

All of the chapters include references to additional resources such as books, blog posts, and papers that will provide additional detail if you want to explore some of the topics in greater depth. For your convenience, there is a bibliography online that contains all the links so that these resources are just a click away. You can find the bibliography at: *http://msdn.microsoft.com/en-us/library/dn320459.aspx.*

What You Need to Use the Code

These are the system requirements for building and running the sample solution:

- Microsoft Windows 7 with Service Pack 1, Microsoft Windows 8, Microsoft Windows Server 2008 R2 with Service Pack 1, or Microsoft Windows Server 2012 (32 bit or 64 bit editions)
- Microsoft Internet Information Server (IIS) 7.0 or later
- Microsoft Visual Studio 2012 Ultimate, Premium, or Professional edition.
- Visual Studio 2012 Update 2
- Windows Azure SDK for .NET, version 2.0 or later (includes the Windows Azure Tools for Visual Studio)
- Microsoft SQL Server 2012, or SQL Server Express 2012
- (Optional) If you wish to deploy the application to Windows Azure, you will also need a Windows Azure subscription

If you wish to configure the solution to use the Windows Azure Table service in the cloud as a key/value store for holding shopping cart information, you must have a valid Windows Azure subscription.

To install and run the polyglot database solution, you must have the following additional software installed and configured:

- MongoDB (version 2.2.3 or later), if you wish to store the product catalog in a document database.
- Neo4J (version 1.8 or later), if you wish to store product recommendations by using a graph database.

You can download the sample code from *http://www.microsoft.com/en-us/download/details.aspx?id=40294*

Who's Who?

This book uses a sample application that illustrates integrating applications with the cloud. A panel of experts comments on the development efforts. The panel includes a cloud specialist, a software architect, a software developer, and a database professional. The delivery of the sample application can be considered from each of these points of view. The following table lists these experts.

Jana is a software architect. She plans the overall structure of an application. Her perspective is both practical and strategic. In other words, she considers the technical approaches that are needed today and the direction a company needs to consider for the future.

"It's not easy to balance the needs of the company, the users, the IT organization, the developers, and the technical platforms we rely on."

Markus is a senior software developer. He is analytical, detail oriented, and methodical. He's focused on the task at hand, which is building a great cloud-based application. He knows that he's the person who's ultimately responsible for the code.

"For the most part, a lot of what we know about software development can be applied to different environments and technologies. But, there are always special considerations that are very important."

Poe is database specialist. He is an expert on designing and deploying databases, whether they are relational or nonrelational. Poe has a keen interest in practical solutions; after all, he's the one who gets paged at 03:00 when there's a problem.

"Implementing databases that are accessed by thousands of users involves some big challenges. I want to make sure our databases perform well, are reliable, and are secure. The reputation of Adventure Works depends on how users perceive the applications that access our databases."

Bharath is a cloud specialist. He checks that a cloud-based solution will work for a company and provide tangible benefits. He is a cautious person, for good reasons.

"The cloud provides a powerful environment for hosting large scale, well-connected applications. The challenge is to understand how to use this environment to its best advantage to meet the needs of your business".

If you have a particular area of interest, look for notes provided by the specialists whose interests align with yours.

The Team Who Brought You This Guide

This guide was produced by the following individuals:

- Program and Product Management: Andrew Oakley (Microsoft Corporation)
- Development: Hanz Zhang (Microsoft Corporation), Douglas McMurtry (Ciber), Mauro Krikorian (Southworks SRL)
- Test team: Mani Subramanian and Carlos Farre (Microsoft Corporation), Partha Krishnasamy (Infosys Ltd), Nicolas Botto (Digit Factory), Vivek Patel (Infosys Ltd)
- Documentation: John Sharp (Content Master Ltd)
- Editor: RoAnn Corbisier (Microsoft Corporation)
- Illustrations and book layout: Chris Burns (Linda Werner & Associates Inc)
- Release Management: Nelly Delgado (Microsoft Corporation)

1 Data Storage for Modern High-Performance Business Applications

In the days of classic business systems analysis, the first tasks that designers typically had to perform were to understand exactly what information a proposed new system needed to store, and how the business was going to use this information. After much consideration, the result was usually a schema that described the structure of the data.

The schemas generated by many common data analysis tools consist of a collection of relational tables and views. There are many reasons for this, not least of which are the simplicity and elegance the relational model, and the fact that the relational model is well-understood by most database specialists.

However, although the relational model is a very powerful abstraction that helps to clarify and solve many data storage problems, it also has its limitations. This chapter describes some of the common challenges that organizations have encountered with the relational model and relational database technology, and discusses how NoSQL databases can help to address these challenges.

Issues with Implementing a Relational Database

Traditionally, businesses have invested a lot of time, effort, and money in modeling their processes and rationalizing the structure of the data that these processes use. The "business database" has become a key element of the IT infrastructure of many organizations, holding the data that supports the day-to-day operations. The core applications that a business depends upon often share this same database. This strategy ensures that the organization does not end up with a disparate set of schemas and duplicated data (possibly in different formats). In essence, the database is the central repository for all the key business systems. The database can also act as the point of integration between business systems, enabling them to work with each other and share information, as shown in Figure 1. In this example, the Sales application creates and manages orders for products that an organization sells, the Inventory application is used to monitor and control the stock levels for each product, and a Customer Relationship Management (CRM) application handles customer information and is used to coordinate marketing campaigns. The data access logic in each application, in conjunction with the database management system software, takes responsibility for coordinating access to the data shared by these applications, as well as ensuring data integrity, and maintaining the consistency of information.

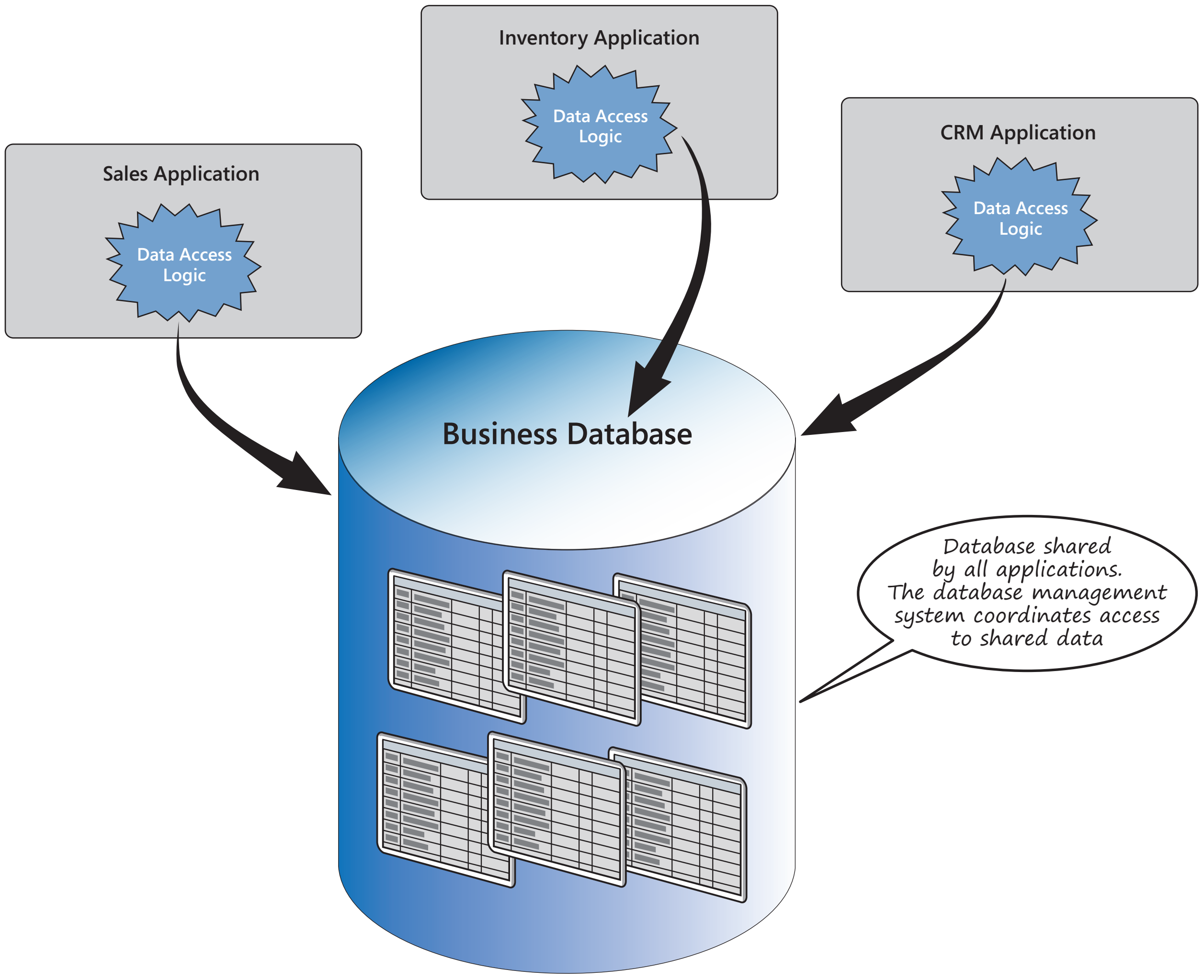


Figure 1
The database as the integration point for business applications

This strategy looks fine in theory, but in practice many organizations have discovered several failings with this approach, including:

- It can take a long time to perform a complete and thorough data analysis, resulting in an extended lead time before developers could actually start writing any code. Business users want their applications up and running quickly. After all, they are the ones who need the system to help them make money for the organization.
- A relational database design consists of a set of tables and you access them by using a query language, most commonly SQL. Programmers prefer to write applications based on other abstractions, such as objects and methods. This mismatch between the database model and the programming model requires developers to either implement a mapping layer in their applications to convert between them, or learn how to access the database directly, possibly by embedding SQL statements directly in their code and using a vendor-specific API to send these requests to the database and convert the data returned into a collection of objects.

- Business requirements can change and new business requirements may appear. Data requirements are subject to change as a result. The longer it takes to construct a data model, the more likely it is to be out of date before the system is complete. A formal approach to database design does not lend itself to building agile business solutions. Organizations that are unable to adapt their systems to rapidly evolving requirements soon become extinct.
- Once a relational schema has been fixed it can become very difficult to modify, especially as more and more applications depend on it. If data requirements change, then any change in the structure of the schema will have an impact on every application that uses the database. If you revise the schema, then there is a real risk that key applications in your organization may stop working correctly unless you can take steps such as performing an impact analysis of every proposed change.
- Successful businesses grow, and the amount of data that they need to hold tends to grow accordingly. A database initially designed to hold many hundreds of records might not scale well if it has to hold millions, or even billions of items. This situation could result in the need to redesign the database to support vastly different volumes of data, with the corresponding impact on the systems that use the database.
- The business may have expanded across multiple, geographically dispersed sites. Additionally, it is now very commonplace for users to be sitting at a remote desktop, or running the application through a web browser connected across the Internet. A database designed to support applications running at a single location might not perform as well when applications access to it from all points of the globe. Network failure or extended network latency may cause database resources to become locked for an extended period, or unavailable, to an application running remotely.
- Sometimes the relational model is just not the best way to describe the data required by an application. The relational model is great at enabling you to implement the bulk of business requirements, but not all relationships can be easily described by the relational algebra that such a model is designed to support.

Licensing costs can also be a significant factor. Many high-end relational database systems that include scalability and high-availability features are very expensive. This can be a massive barrier for many small to medium sized business that have to manage significant amounts of data but only have a limited budget. Many NoSQL offerings are open source, which makes them a very attractive option to these organizations.

The common factors that transcend most of these concerns are agility, flexibility, performance, and scalability. For business systems that are critical to the day-to-day operations of an organization, database availability is also an important issue. The following sections summarize these concerns.

Agility and Programmability

The *impedance mismatch* between the structure of data in the database and the models used by most applications has an adverse effect on the productivity of developers. Developers can spend a significant amount of time developing a mapping layer that converts data between the object model used by the application and the tables stored in the database, rather than focusing on the business logic of the application itself. Using a commercial Object-Relational Mapping (ORM) layer such as the Microsoft ADO.NET Entity Framework can reduce some of the effort involved, but it is not always an ideal solution.

Chapter 3, "Implementing a Relational Database" describes the use of ORMs and the Entity Framework with a relational database.

Flexibility

Chapter 3 summarizes strategies for reducing the overhead of multi-table joins by denormalizing the database schema.

In theory, the relational model is extremely flexible and can model almost any type of data and relationships. In practice, overfamiliarity with the relational model has led many developers to design solutions and user interfaces that overemphasize the tabular way in which the data is stored rather than implementing a data storage schema that meets the business requirements; a case of bending the solution to match the technology. The situation is not helped by the nature of SQL. As a query language it is brilliant for specifying from which tables a query should return data, and how to join, sort, and aggregate data. However, SQL is extremely focused on data as a set of rows and columns, and not all information can be shoehorned easily into this shape. Furthermore, the tendency of relational database designers to focus on a pure, normalized schema can lead to a database that contains a large set of narrow tables. The result is that most queries will need to perform join operations to reconstruct the entities that business applications need, and this process can be slow and resource intensive.

Performance and Scalability

Many systems developed in the 1970s and 1980s made assumptions about the volume of requests that were likely to occur, the number of users making these requests, and the locations of these users. In many cases, it was also assumed that these statistics were unlikely to vary much over time. Relational database technology can work very well if it is tuned carefully to match these assumptions.

If the volume of traffic increased, many organizations elected to scale up the hardware on which it was running to handle the additional increased load. However, there comes a point at which scaling-up is no longer feasible, either through the limitations of the hardware or the expense of maintaining large servers. At this point you can scale out by partitioning the data into a set of smaller databases and running each database on a separate commodity (and cheaper) server.

You can partition data vertically (splitting tables into subsets of columns and implementing each subset as a separate table) and horizontally (splitting tables into subsets of rows, and implementing each subset as a separate table where each table contains the same columns). You can place different partitions on different servers to spread the load and improve scalability.

Figure 2 shows an example of horizontal partitioning across servers. In this example, the data for different rows is divided across tables running on different servers (each table has the same schema). When the various applications query this data, the requests are routed to the appropriate partition. This schema is also known as *sharding*. Many relational databases support sharding by implementing a technique known as federation; the different database servers maintain metadata that identifies which rows they contain, and an application can connect to any server that uses this metadata to route the request to the appropriate server. However, federation may require additional configuration, and can be expensive in terms of licensing costs. For this reason, it is common to implement sharding manually, by incorporating the routing strategy into the data access logic of the applications that use the data, as shown in Figure 2. This is also known as the *Shared Nothing* approach because the database servers do not share any information about the data that they hold with any other servers.

Sharding is also a key scalability strategy employed by many NoSQL databases. The section "Improving Scalability and Reducing Network Latency" later on in this chapter provides more information.

Chapter 3 discusses vertical and horizontal partitioning strategies for relational databases in more detail.

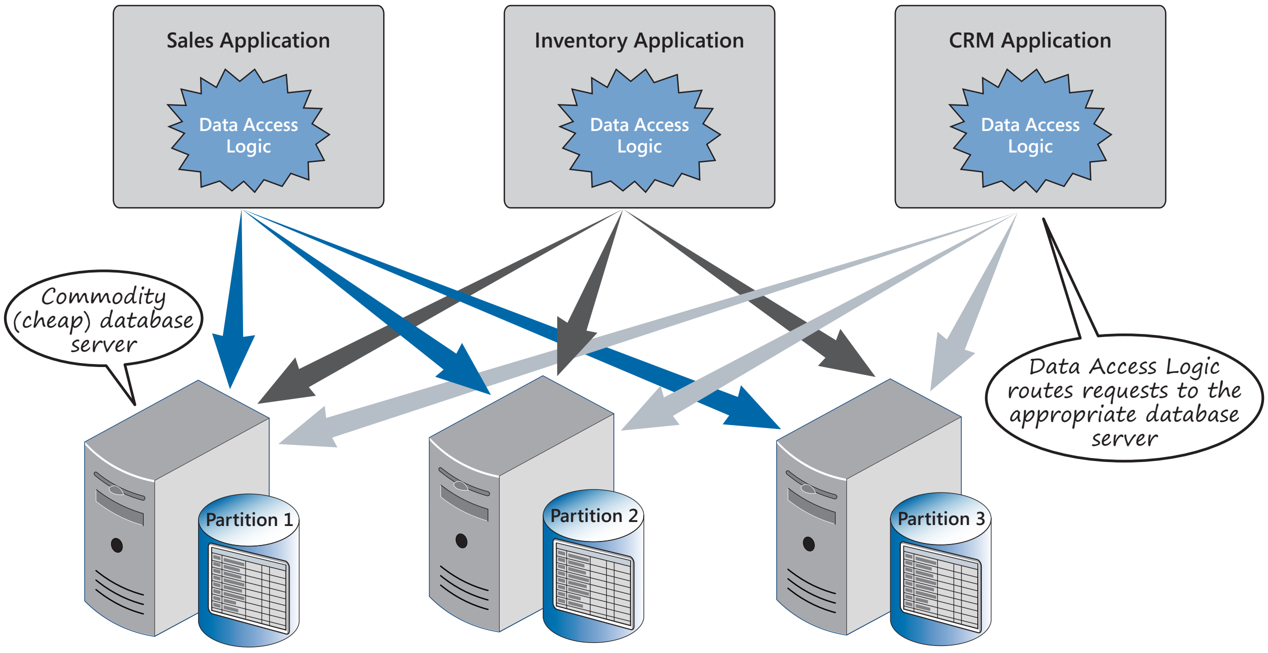


Figure 2
Scaling out a database using sharding

Partitioning enables you to handle more data and users, but can have an important impact on the performance of query operations that need to join data held in different partitions, so you need to design your partitions carefully to minimize these occurrences. Additionally, in a partitioned system, transactions that update data held in different partitions can have a significant effect on the throughput and responsiveness of your applications. Relational databases place great emphasis on ensuring that data is consistent, and they achieve this by implementing ACID (Atomic, Consistent, Isolated, and Durable) transactions. An ACID transaction that modifies data in multiple tables only completes successfully if every modification is successful; if any part of the transaction fails, then the entire transaction is rolled back and any work already completed is undone. To prevent concurrent users from seeing partial updates as a transaction progresses, most relational databases lock the data as it is changed, and then only unlock the data when the transaction completes successfully or it fails. In either case, the locks ensure that other users only see a consistent view of the data. If they try and access locked data they will be blocked until the lock is released. For short-lived transactions, concurrent users may only be delayed for a short while, but the longer a transaction takes, the more noticeable this delay gets.

If you scale out a relational database, transactions that previously encompassed data in a single database might now have the complication of spanning multiple databases, requiring careful coordination of locks across databases. In the early 1990s, the X/Open group defined a model for distributed transaction processing and published the XA specification. The XA specification describes how to use a protocol called the Two Phase Commit (2PC). The details are unimportant here except to state that this protocol enables an application to implement ACID transactions across more than one database. This consistency comes at a price however.

When the XA specification was first introduced, fast, private corporate networks had become commonplace and issues such as network latency were not considered too important. However, as the 1990s progressed, the increasing use of the Internet, and the World Wide Web in particular enabled many businesses to take advantage of the connectivity that was now available to build online systems. These systems had to support a widely varying number of concurrent users, generating an unpredictable volume of traffic, from sites located anywhere in the world. Additionally, the opportunities to capture and store more data meant that databases inevitably grew larger. Unfortunately, the emphasis that distributed transactions place on consistency can impact the performance of a system that spans a large public network with unpredictable connectivity and throughput characteristics; the more dispersed the databases that participate in a distributed transaction, and the slower the transaction becomes. A slow transaction locks data for longer, increasing the likelihood that a concurrent request for the same data will be blocked. As the traffic increases, the more likely such collisions are to occur, and the system can grind to a halt. You can take steps to reduce the scope for such slow-down, but it is very difficult to eliminate it altogether.

Chapter 3 provides more information on implementing efficient transactions in a relational database. You can also consider trading consistency for performance by implementing BASE (Basically Available, Soft state, Eventual consistency) transactions, although this strategy often requires additional infrastructure to work successfully. Appendix A, "Replicating, Distributing, and Synchronizing Data" in the guide "Building Hybrid Applications in the Cloud on Windows Azure" *available from patterns & practices describes a possible implementation based on Windows Azure Topics and Subscriptions.*

In more general terms, you can follow the Command Query Responsibility Segregation (CQRS) pattern to divide a distributed system up into functionality that is responsible for writing data and functionality that handles queries. For more information, download the patterns & practices guide "CQRS Journey" from MSDN.

Availability

The relational model does not address the issue of availability, although many relational database management systems offer high availability (for a price). In most cases, availability is implemented by maintaining a copy of the database. This copy might be hosted by a separate failover server that is brought online if the primary server hosting the database fails, or it might be sent to one or more additional servers that provide load-balancing for concurrent users (user requests may be transparently directed to whichever server is least-heavily loaded).

However, increased availability complicates consistency. If you maintain multiple copies of data, and that data changes, you need to ensure that each copy is modified consistently. The more changes that you need to make, spanning more databases, the slower transactions can become.

The Advent of NoSQL

A number of well-known organizations, building Internet-facing systems that capture huge amounts of data and that support massive numbers of concurrent requests, felt that relational technology did not provide the scalability, availability, and performance that they required. Instead, they decided to create their own non-relational solutions tailored to their requirements. Other organizations have followed a similar strategy leading to a selection of non-relational database systems, each designed to solve a specific set of problems. Collectively, this ever-expanding set of database systems is referred to as *NoSQL*.

The term NoSQL *is an historic anomaly, originally intended to convey the idea that applications do not use SQL to interact with the database. In a lot of literature on the subject NoSQL is defined as meaning "not only SQL," but this description would also fit many common relational database systems that implement custom extensions to the SQL language (such as Microsoft Transact-SQL), and these are not NoSQL databases in the manner that most NoSQL enthusiasts would recognize. A better term might have simply have been "non-relational," but the NoSQL tag has stuck and is now in common use.*

Throughout this book, we refer to a NoSQL database when we mean a non-relational database.

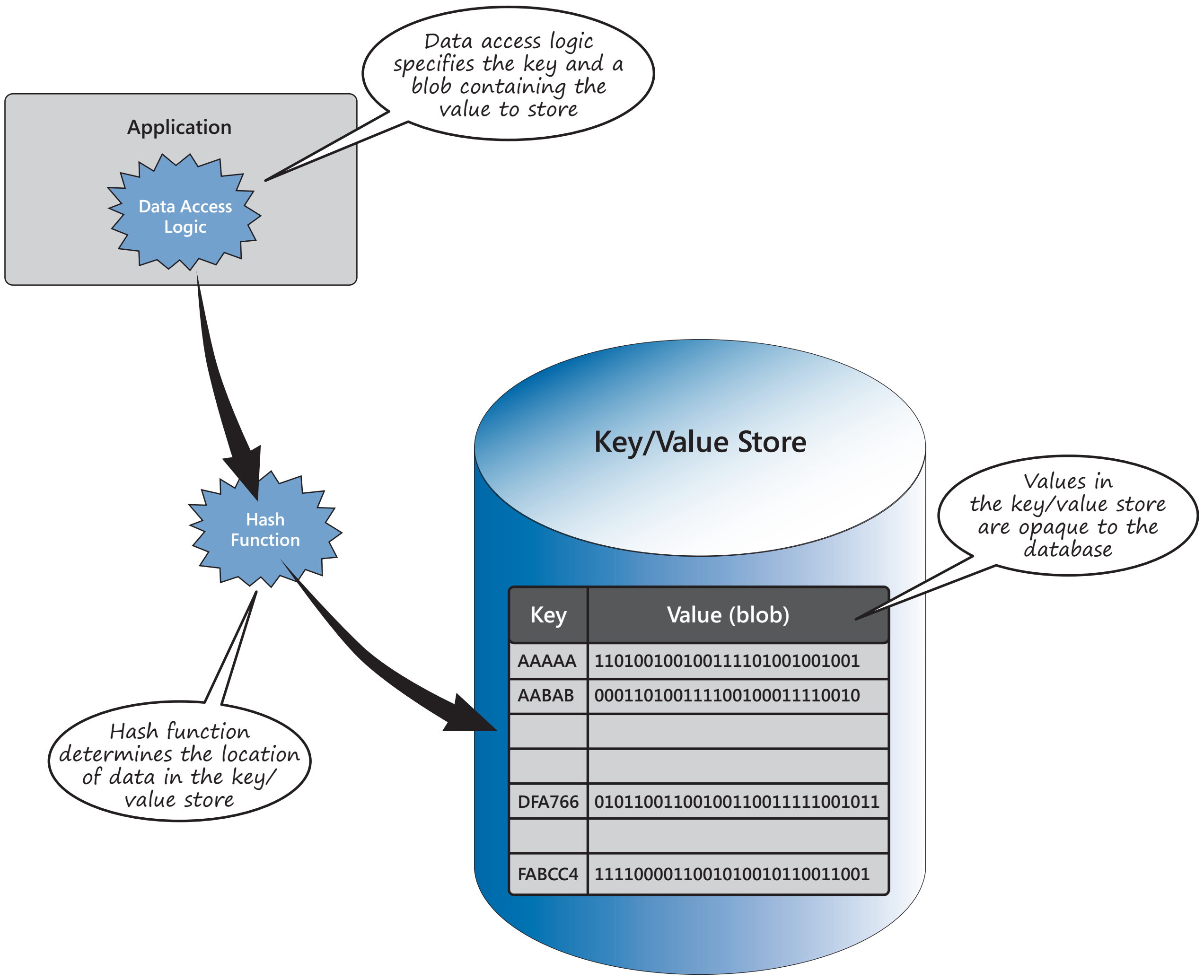
NoSQL databases come in a variety of shapes and functionality. Arguably, the only feature that unifies them is that they are not relational. Other than that, they can differ quite markedly from each other. However, the software industry has attempted to categorize NoSQL databases into a small set of functional areas: key/value stores, document databases, column-family databases, and graph databases. The following sections summarize the distinguishing features of each of these categories.

Some NoSQL databases fit naturally into a single category, while others include functionality that spans categories. For example, some key/value stores can also exhibit some of the behavior of document databases. Therefore, you should not view these categories as a collection of definitive and constraining functional areas, but as a continuum of features.

The article "Getting Acquainted with NoSQL on Windows Azure" available on the ISV Developer Community blog provides an overview of common NoSQL databases and describes some common implementations. You can also find more general information on the NoSQL-Database.org website.

Key/Value Stores

A key/value store implements arguably the simplest of the NoSQL storage mechanisms, at least at a conceptual level. A key/value store is essentially a large hash table. You associate each data value with a unique key, and the key/value store uses this key to determine where to store the data in the database by using an appropriate hashing function. The hashing function is selected to provide an even distribution of hashed keys across data storage. The values deposited in a key/value store are opaque to the database management system software. Essentially, values are BLOBs and the key/value store simply retrieves or stores a value by using the key provided by the application. For this reason, most key/value stores only support simple query, insert, and delete operations. To modify a value an application must overwrite the existing data for the entire value. An application can store whatever data it likes, although some key/value stores impose limits on the maximum \size of values. In most implementations, reading or writing a single value is an atomic operation (if the value is large, writing may take some time).


Data access logic specifies the key and a blob containing the value to store
Application
Data Access Logic
Key/Value Store
Values in the key/value store are opaque to the database
Hash Function
Hash function determines the location of data in the key/value store

Key	Value (blob)
AAAAA	1101001001001111010010010001
AABAB	0001101001111001000111110010
DFA766	0101100110010011001111110010 11
FABCC4	1111000011001010010110011001

FIGURE 3
The conceptual structure of a key/value store

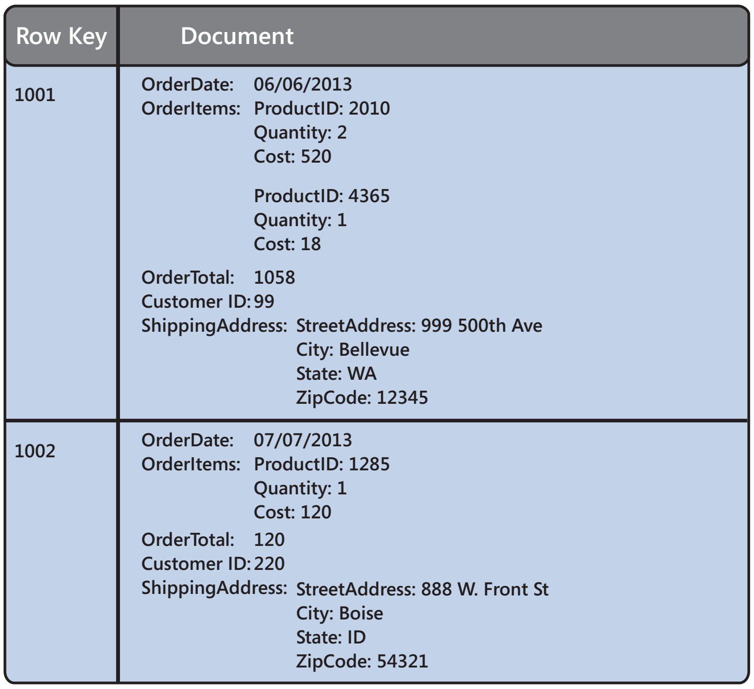
Document Databases

Chapter 4, "Implementing a Key/Value Store" provides detailed information on creating and using a key/value store in a business application.

A document database is similar in concept to a key/value store except that the values stored are documents. A document is a collection of named fields and values, each of which could be simple scalar items or compound elements such as lists and child documents. The data in the fields in a document can be encoded in a variety of ways, including XML, YAML, JSON, BSON, or even stored as plain text.

The term "document" in this context does not imply a free-form text structure, rather a document is the name given to a collection of the related data items that constitute an entity.

The fields in the documents are exposed to the database management system, enabling an application to query and filter data by using the values in these fields. Figure 4 shows an example of a couple of documents holding sales order information in a document database. The items that comprise each order are held as a list with the order information.



Row Key	Document		
1001	OrderDate:	06/06/2013	
	OrderItems:	ProductID: 2010	
		Quantity: 2	
		Cost: 520	
		ProductID: 4365	
		Quantity: 1	
		Cost: 18	
	OrderTotal:	1058	
	Customer ID:	99	
	ShippingAddress:		StreetAddress: 999 500th Ave
			City: Bellevue
			State: WA
			ZipCode: 12345
1002	OrderDate:	07/07/2013	
	OrderItems:	ProductID: 1285	
		Quantity: 1	
		Cost: 120	
	OrderTotal:	120	
	Customer ID:	220	
	ShippingAddress:		StreetAddress: 888 W. Front St
			City: Boise
			State: ID
			ZipCode: 54321

Figure 4
An example set of documents in a document database

The database in this example is designed to optimize query access to sales order information. In the application that uses this database, the most common operations performed on this data are queries that retrieve the details of each order. The details of the order are unlikely to change, so the denormalized structure imposes little overhead on updates while ensuring that queries can be satisfied with a single read operation.

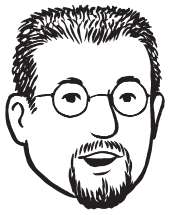
A problem that a relational database specialist might have with this example concerns the shipping address that is embedded into the sales order. In a typical relational database, this information would be held in a separate table, enabling it to be modified without having to hunt down all the orders that reference it. However, the relational approach is not necessarily an advantage. A sales order is an historical document, and the shipping address should indicate where the order was actually sent. If the address of the customer changes at a later date, the shipping address for orders already fulfilled should not be changed (indeed, doing so may break the auditing requirements of the system). Implementing this functionality in a relational database would require maintaining version histories and timestamps for each shipping address, and each query that required the shipping address for an order would be more complicated as a result. The document database is more naturally tuned to this type of information, and queries are simpler as a result.

A typical document contains the entire data for an entity. The items that constitute an entity are application specific; for example, they could be the details of a customer, an order (as shown in Figure 4), or a combination of both. Additionally, a document store does not require that all documents have the same structure. This free-form approach confers great flexibility on applications, which can store different data in documents as business requirements change.

Chapter 5, "Implementing a Document Database" describes how to create and use a document database in more detail.

An application can query documents by using the document key (which is hashed, to help distribute data evenly), but a more common approach is to retrieve documents based on the value of one or more fields in a document. Some document databases support indexing to facilitate fast lookup of documents based on an indexed field. Additionally, many document databases support in-place updates, enabling an application to modify the values of specific fields in a document without rewriting the entire document. Read and write operations over multiple fields in a single document are usually atomic.

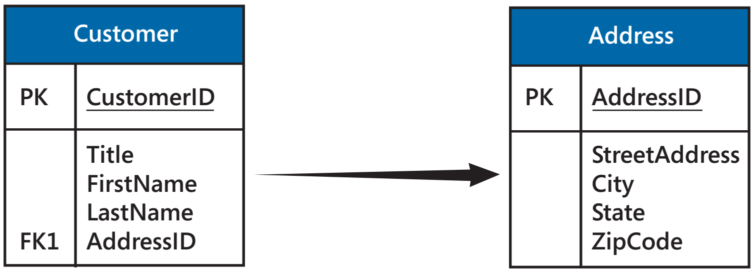
Column-Family Databases



Column-family databases are also known as *Wide Column* databases or *Extensible Record* databases.

A column-family database organizes its data into rows and columns, and in its simplest form a column-family database can appear very similar to a relational database, at least conceptually. However, the real power of a column-family database lies in its denormalized approach to structuring sparse data.

For example, if you need to store information about customers and their addresses in a relational database (ignoring the need to maintain historical data as described in the previous section), you might design a schema similar to that shown in Figure 5. This diagram also shows some sample data. In this example, customer 1 and customer 3 share the same address, and the schema ensures that this address information is not duplicated. This is a standard way of implementing a one-to-many relationship.



Customer Table

CustomerID	Title	FirstName	LastName	AddressID
1	Mr	Mark	Hanson	500
2	Ms	Lisa	Andrews	501
3	Mr	Walter	Harp	500

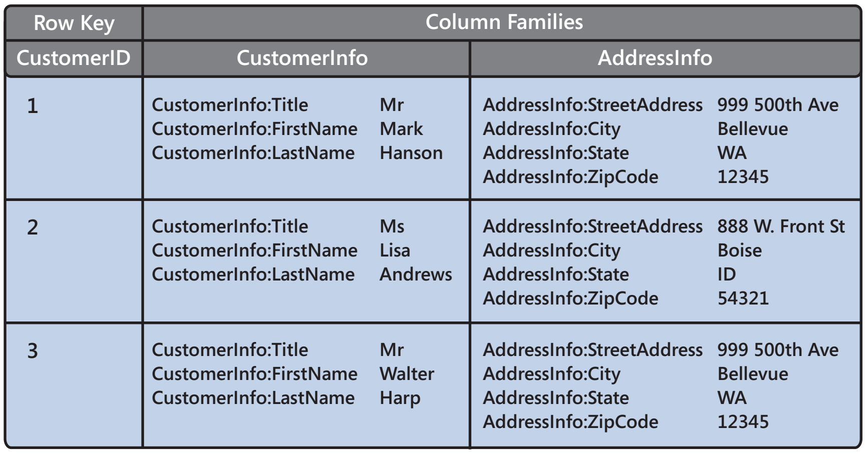
Address Table

AddressID	StreetAddress	City	State	ZipCode
500	999 500th Ave	Bellevue	WA	12345
501	888 W. Front St	Boise	ID	54321

Figure 5
Implementing a one-to-many relationship in a relational database

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

The purpose of a column-family database is to handle situations such as this very efficiently. You can think of a column-family database as holding tabular data comprising rows and columns, but you can divide the columns into groups known as *column-families*. Each column-family holds a set of columns that are logically related together. Figure 6 shows one way of structuring the same information as Figure 5 by using a column-family database to group the data into two column-families holding the customer name and address information. Other schemes are possible, but you should implement your column-families to optimize the most common queries that your application performs. In this case, queries that retrieve the addresses of customers can fetch the data with fewer reads than would be required in the corresponding relational database.



Row Key	Column Families			
CustomerID	CustomerInfo		AddressInfo	
1	CustomerInfo:Title	Mr	AddressInfo:StreetAddress	999 500th Ave
	CustomerInfo:FirstName	Mark	AddressInfo:City	Bellevue
	CustomerInfo:LastName	Hanson	AddressInfo:State	WA
			AddressInfo:ZipCode	12345
2	CustomerInfo:Title	Ms	AddressInfo:StreetAddress	888 W. Front St
	CustomerInfo:FirstName	Lisa	AddressInfo:City	Boise
	CustomerInfo:LastName	Andrews	AddressInfo:State	ID
			AddressInfo:ZipCode	54321
3	CustomerInfo:Title	Mr	AddressInfo:StreetAddress	999 500th Ave
	CustomerInfo:FirstName	Walter	AddressInfo:City	Bellevue
	CustomerInfo:LastName	Harp	AddressInfo:State	WA
			AddressInfo:ZipCode	12345

Figure 6
The structure of data in a column-family database

A relational database purist will argue that this column-family schema is less optimal than the relational schema because of the data redundancy; the address details for customers 1 and 3 are repeated. However, the details of a physical building are unlikely to change very often (buildings don't move), so although this data occupies more space than it would in a relational database, maintaining this duplicate information is unlikely to have much overhead.

The illustration in Figure 6 is conceptual rather than physical, and is intended to show the logical structure of the data rather than how it might be materially organized. Each row in a column-family database contains a key, and you can fetch the data for a row by using this key. Unlike a key/value store or a document database, most column-family databases store data in key order rather than by computing a hash to determine the location, so you should pay close attention to how applications are likely to query and sort data and select the row key appropriately.

Many implementations also allow you to create indexes over specific columns in a column-family, and this feature enables you to retrieve data by referencing the values stored in the individual columns rather than the row key. In the example shown above, if you created an index over the **State** column in the **AddressInfo** column-family, you could easily find all customers with a **State** value of **WA**.

Column-family databases are designed to hold vast amounts of data (hundreds of millions of rows). Even if data is indexed, it can take a little time to find an item. Many column-family databases implement Bloom filters to help determine whether an item actually exists before attempting to retrieve it. In this way, the database does not have to spend time performing fruitless searches. For more information, see Chapter 6, "Implementing a Column-Family Database."

Another feature of a typical column-family database is that the structure of the information in a column-family can vary from row to row; different rows can have different fields and the data does not confirm to a rigid layout. In the example shown in Figure 6, this variability is useful if the database needs to include the details of customers in other countries. For example, the United Kingdom includes county names rather than states in addresses, and uses post codes instead of zip codes. If Mark Hanson relocated to the UK, his address details might change as shown in Figure 7. Note that if you query a column-family database by referencing an indexed column that does not exist in every row, such as the **State** column described previously, the rows without this column will simply be omitted from the results (they will not be included in the index that spans this column). This feature enables a column-family database to hold sparse data very efficiently.

In a column-family database, you can think of a column as a simple name/value pair and a column-family as a collection of columns. Unlike a relational database, the names of columns do not have to be static (they can actually be generated from data items).

Row Key	Column Families			
CustomerID	CustomerInfo		AddressInfo	
1	CustomerInfo:Title CustomerInfo:FirstName CustomerInfo:LastName	Mr Mark Hanson	AddressInfo:StreetAddress AddressInfo:City AddressInfo:County AddressInfo:PostCode	999 Thames St Reading Berkshire RG99 922
2	CustomerInfo:Title CustomerInfo:FirstName CustomerInfo:LastName	Ms Lisa Andrews	AddressInfo:StreetAddress AddressInfo:City AddressInfo:State AddressInfo:ZipCode	888 W. Front St Boise ID 54321
3	CustomerInfo:Title CustomerInfo:FirstName CustomerInfo:LastName	Mr Walter Harp	AddressInfo:StreetAddress AddressInfo:City AddressInfo:State AddressInfo:ZipCode	999 500th Ave Bellevue WA 12345

Figure 7
A column-family database with different fields for the columns

Although Figures 6 and 7 show the column-families for customers appended together to form a single logical entity they are most likely to be stored separately, with the **CustomerInfo** column-family on one disk and the **AddressInfo** column-family on another, in a simple form of vertical partitioning. You should really think of the structure in terms of column-families rather than rows. The data for a single entity that spans multiple column-families will have the same row key in each column-family. As an alternative to the tabular layout shown previously, you can visualize the data shown in Figure 7 as the following pair of structures.

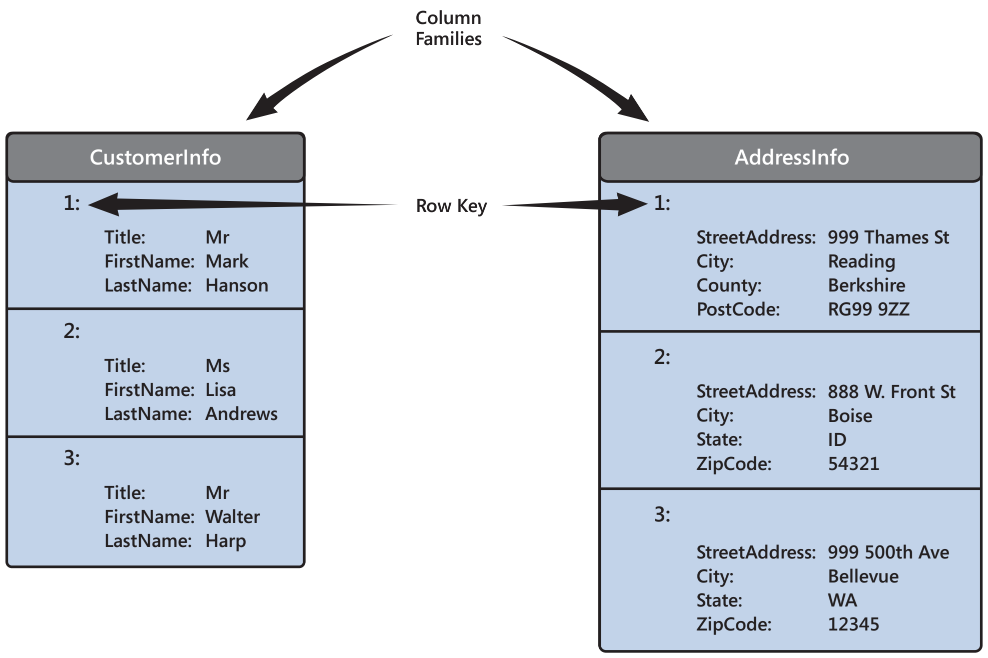


Figure 8
An alternative view of the structure of the data in a column-family database

Chapter 6, "Implementing a Column-Family Database" provides more information on designing and using a column-family database in a business application.

In a column-family database, read and write operations for the part of a row in a single column-family are usually atomic, although some implementations provide atomicity across the entire row (spanning multiple column-families) as well.

Graph Databases

Just like the other categories of NoSQL databases, a graph database enables you to store entities, but the main focus is on the relationships that these entities have with each other. A graph database stores two types of information; nodes that you can think of as instances of entities, and edges which specify the relationships between nodes. Nodes and edges can both have properties that provide information about that node or edge (like columns in a table). Additionally, edges can have a direction indicating the nature of the relationship.

The purpose of a graph database is to enable an application to efficiently perform queries that traverse the network of nodes and edges, and to analyze the relationships between entities. Figure 9 shows an organization's personnel database structured as a graph. The entities are the employees and the departments in the organization, and the edges indicate reporting lines and the department in which employees work. In this graph, the arrows on the edges show the direction of the relationships.

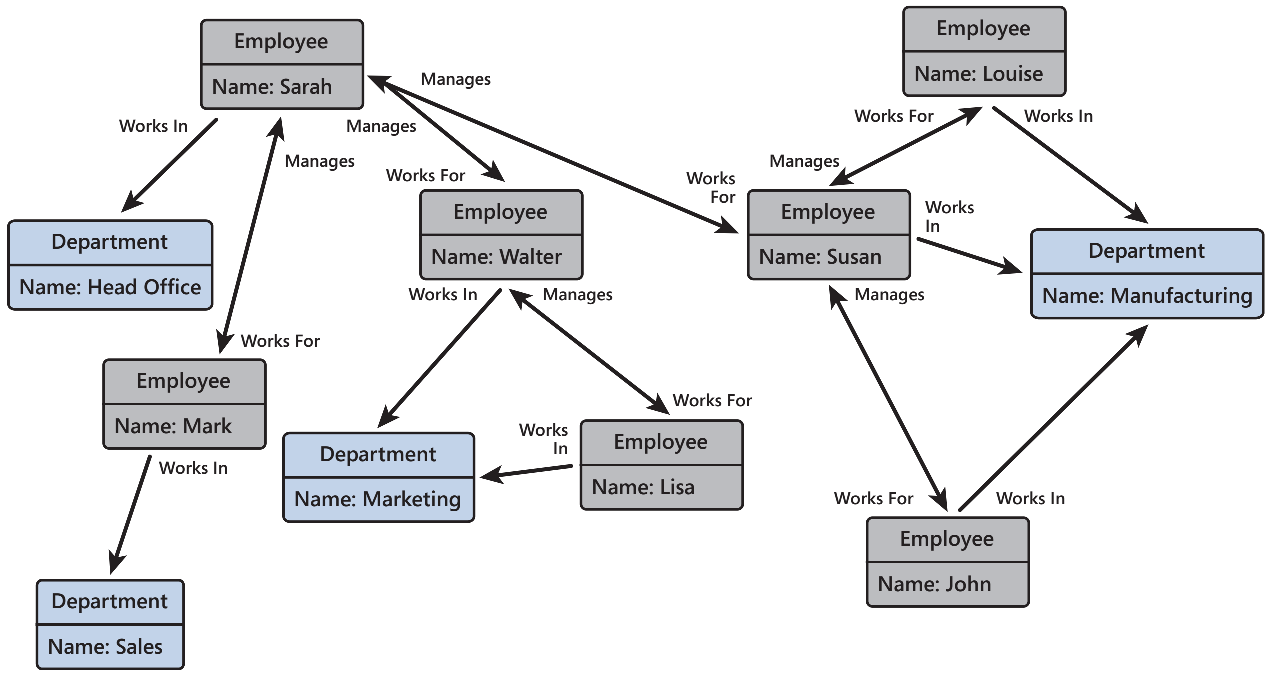


Figure 9
Personnel information structured as a graph

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

Chapter 7, "Implementing a Graph Database" describes how to create and use a graph database in more detail.

Common Features of NoSQL Databases

Despite their differences, there are some common features that underpin most NoSQL databases. These features are the result of the common requirement for most NoSQL databases to provide high scalability and performance, enabling large numbers of requests to be performed very quickly and efficiently. However, to meet these goals, many NoSQL databases make tradeoffs between speed and consistency. This section describes the approach that many developers using NoSQL databases follow to achieve fast performance, while at the same time ensuring that the data that the databases hold remains meaningful.

Designing Aggregations

Relational databases conceptually hold data as rows in tables, but key/value stores, document databases, and column-family databases all use their own non-relational structures to group related data together into aggregates, whether this data is an opaque BLOB, a document, or a collection of values held in a column-family associated with a given row key. The common features that these three NoSQL examples share is that an aggregate typically identifies a discrete set of data values, and a NoSQL database usually ensures that operations that affect a single aggregate are atomic.

An aggregate in a NoSQL database is similar to a row in a table in a relational database.

The term "aggregation" comes from Domain-Driven Design (DDD), a collection of guidelines and patterns that you can use to design and implement elegant object-oriented systems. In a DDD approach, an aggregate is simply a collection of items that should be handled as a single atomic unit.

An aggregate is the unit of consistency in most NoSQL databases. You can consider operations that update multiple fields in a single aggregate as exhibiting ACID properties, although operations that modify data spread across more than one aggregate are unlikely to provide the same guarantees of consistency (some NoSQL databases do support ACID transactions that span aggregates, but these tend to be the exceptions rather than the rule). Note that in most column-family databases the unit of aggregation is the column-family and not the row, so a write operation that modifies data in several column-families for the same row key might not be atomic.

Graph databases tend not to use aggregates in the same manner as key/value stores, document databases, or column-family databases. This is due to their primary focus being the relationships between entities. The write operations performed on a single node or an edge are often atomic, and some graph databases implement ACID semantics for operations that define relationships spanning multiple nodes and edges.

The aggregate is also the unit of I/O employed by many NoSQL databases. When an application retrieves a value from key/value storage, a document from a document database, or a set of values from a column-family, the entire aggregate (BLOB, document, or row from column-family) is fetched and passed to the application.

When you implement a NoSQL database, design it with aggregates in mind. Make sure that you understand how applications will query data, and how they are likely to process related information in operations. Design the database to optimize query access, store related data together in the same aggregate, and handle any denormalization issues when you write data back to the database.

Currently, few NoSQL databases provide any design tools. Additionally, the schemaless nature of most NoSQL databases means that they do not provide the equivalent of the CREATE TABLE statements available in SQL (there are some exceptions). You typically write your own application code to create the aggregates for a NoSQL database, using the APIs provided by the NoSQL database vendor.

Materializing Summary Data

One advantage that relational databases have over NoSQL solutions is their generality. If you can store information by using the relational model you can perform almost any query over this data, although the resulting SQL may contain complex joins and subqueries. To reduce the overhead associated with locating and retrieving the data for a complex query at runtime, relational databases provide views. You can think of a view as a precompiled, pre-optimized SQL statement that an application can query in the same way as a simple table. When an application queries a view, the underlying SQL statements that define the view are run.

Some relational database management systems take the notion of views a stage further. For data that is queried often but updated relatively infrequently, they support materialized views. A materialized view is a pre-computed dataset that represents the data exposed through a view; essentially it is a copy of the data that the SQL statement for a regular view would retrieve, and it is stored in a separate table. When an application queries a materialized view, it simply retrieves this copy of the data. Materialized views can become stale, and it is an application design decision to determine the frequency with which the data in a materialized view should be refreshed.

Although most NoSQL databases do not provide views in the same way as relational databases, many NoSQL databases enable you to adopt a similar approach to materialized views for applications that need to combine or summarize data. For example, if you store customer and address information in a column-family database such as that shown in Figures 6 and 7, but you regularly need to perform an analysis of customers based on their geographies such as "how many customers are located in WA?," you could define an additional column-family that contains summary information, as shown in Figure 10. In this case, the row keys are the states from the **AddressInfo** column-family.

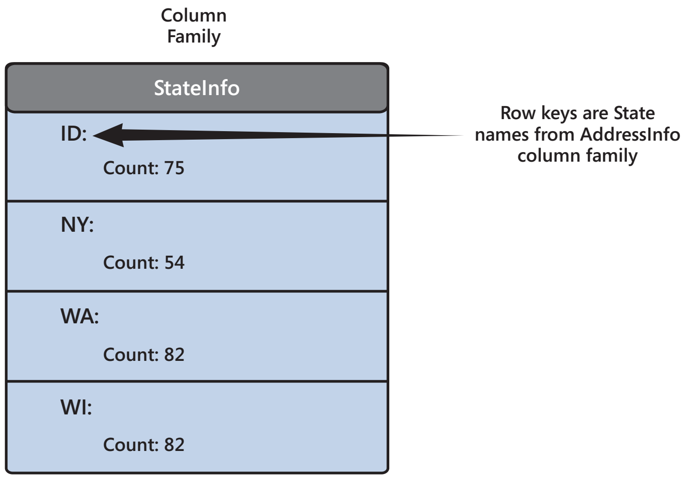


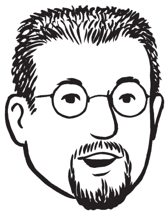
Figure 10
Storing summary information in a column-family database

For a relatively static dataset such as customers and address, the overhead of maintaining summary information might be small compared to the cost of computing this data every time an analysis is performed. However, for more dynamic data, such as customer orders, this overhead can become significant. In these situations, you should determine how accurate and up-to-date such summary information needs to be. If the data does not need to be completely fresh all of the time, you can perform the calculations on a periodic (daily, weekly, or even monthly) basis and update the summary information.

You can find detailed information about using map/reduce frameworks to analyze and summarize data at the mapreduce.org website.

Many NoSQL databases provide map/reduce frameworks that enable you to implement incremental updates for summary data in a NoSQL database. Most map/reduce frameworks generate summary data on a per-aggregate basis, and then combine these per-aggregate summaries into totals. As an optimization mechanism, if the data in an aggregate is unchanged, the summary data for that aggregate does not have to be recomputed.

Implementing High Availability



If you design aggregates carefully, you can optimize the way in which many map/reduce frameworks generate summary data.

According to Brewer's CAP Theorem, a distributed system can only implement two out the following three features at any one time; consistency, availability, and partition tolerance. In a distributed environment, most relational database systems focus on consistency whereas most NoSQL databases are designed to maximize availability.

Many relational database systems implement high availability through clustering. They maintain copies of data across several computers, and if one machine fails, requests can be transparently redirected to a working computer. However, the relational model, which places great emphasis on data consistency, was not designed with high-availability in mind, and the effort of maintaining redundant and consistent data across multiple databases while keeping performance at an acceptable level can consume considerable resources.

In contrast, most NoSQL databases are explicitly designed around high-availability, and their architecture is geared towards maintaining high performance and availability. Absolute consistency is a lower priority, as long as data is not lost and eventually becomes consistent.

NoSQL databases commonly implement one of two replication schemes for ensuring that data is available; primary/secondary replication (sometimes referred to as master/subordinate replication) and peer-to-peer replication, as described in the following sections.

Primary/Secondary Replication

In a primary/secondary replication cluster, a copy of the database is maintained at each participating server. One server is designated as the primary or master server, and all requests that write data to the database are routed through this server. The primary server updates its local copy of the database and then indicates that the write request is complete. The application issuing this request can continue in the knowledge that the change has been saved. At some point, the primary server propagates the same changes to each of the other servers, referred to as secondary servers or subordinates.

To ease the load on the primary server, many implementations of this scheme enable any server (primary or secondary) to respond to read requests.

Read requests made by client applications should be spread across the servers to maximize throughput, possibly taking into account factors such as the location of the client application relative to each server to reduce latency. The routing logic for requests should be hidden from the client application, possibly inside a library that implements the connectivity to the database, often provided by the NoSQL vendor.

Primary/secondary replication can help to maximize read performance at the cost of consistency. I It is possible for an application to save data to the primary server, and then attempt to read the same data back from a secondary server and get an older version of the data if the change has not yet been fully propagated. Equally, other applications may see different results when they query the same data, depending upon which servers they access. However, all changes will, at some point, find their way across all servers, and the system will eventually become consistent again.

The section, "Improving Consistency" later in this chapter describes some mechanisms that many NoSQL databases implement to reduce the possibility of inconsistent data.

Figure 11 illustrates the simplified structure of a primary/secondary replication cluster.

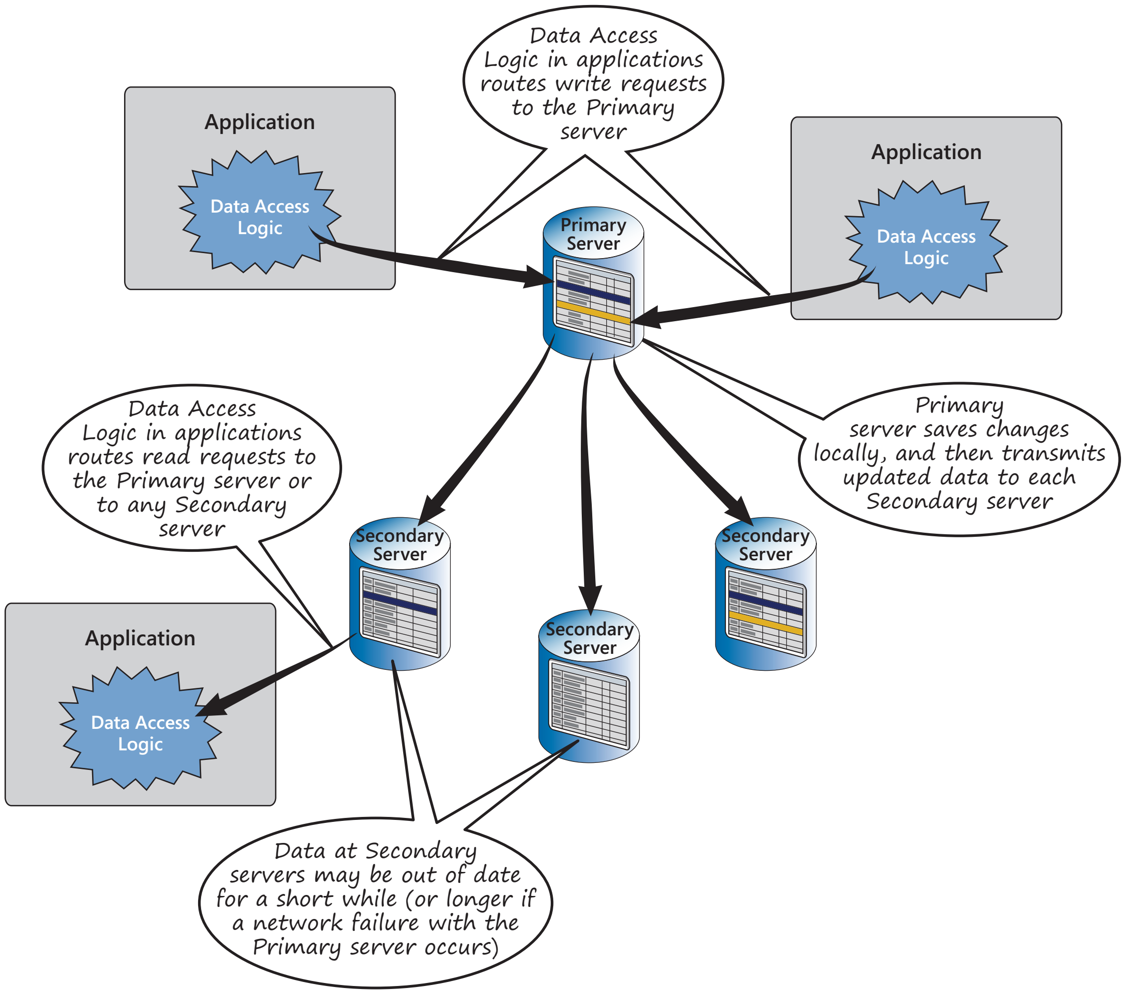

Data Access Logic in applications routes write requests to the Primary server
Application
Data Access Logic
Primary Server
Application
Data Access Logic
Primary server saves changes locally, and then transmits updated data to each Secondary server
Data Access Logic in applications routes read requests to the Primary server or to any Secondary server
Secondary Server
Secondary Server
Secondary Server
Application
Data Access Logic
Data at Secondary servers may be out of date for a short while (or longer if a network failure with the Primary server occurs)

Figure 11
A simplified primary/secondary replication cluster

A simple primary/secondary cluster exposes three principal possible failure points:

- The primary server could fail. In this case, one of the secondary servers can be promoted to the role of the primary server while the original primary server is recovered. When the original primary server comes back online, it can act as a secondary server and be brought up to date with changes from the new primary server.
- A secondary server could fail. This is not necessarily a big problem because any read requests made by applications can be directed towards working server while the failed server is recovered and the changes it has missed in the meantime are applied.
- The network connecting the primary server to one or more secondary servers could fail. In this situation, one or more secondary servers will be isolated and will not receive changes from the primary server. The data stored by the secondary servers will become stale, and applications issuing read requests to these servers will be presented with out-of-date information. When connectivity is restored, the changes from the primary server can be applied.

Peer to Peer Replication

A primary/secondary cluster can help to balance the load in a system that is read-intensive, but the primary server can act as a bottleneck if the system performs a large number of write operations. In this situation peer-to-peer replication might be more beneficial.

In a peer-to-peer cluster all servers support read and write operations. An application connects any of the servers when it wants to write data. The server saves the changes locally and indicates to the application that the write operation is complete. The data is subsequently transmitted to all other servers.

This approach spreads the load, but can result in conflicting changes if two concurrent applications change the same data simultaneously at different replication servers. In this situation, the system needs to resolve the conflicting data. The section "Improving Consistency" later in this chapter provides more information.

Figure 12 shows the structure of a peer-to-peer replication cluster.

Many peer-to-peer systems implement the Gossip Protocol to exchange information about updates between replication servers. However, the more servers that a cluster contains, the longer it can take for updates to propagate, increasing the possibility of conflicting updates occurring at different servers.

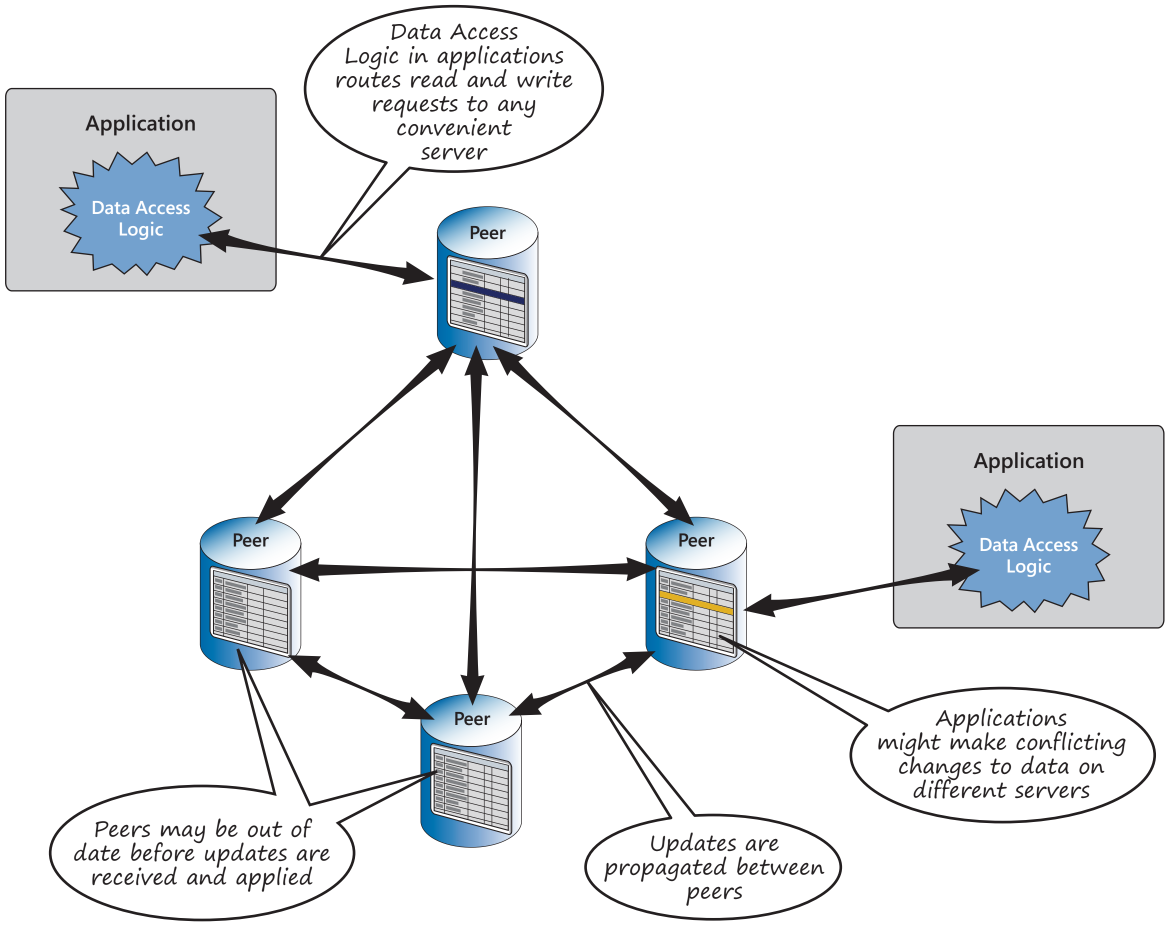

Application
Data Access Logic
Data Access Logic in applications routes read and write requests to any convenient server
Peer
Peer
Peer
Application
Data Access Logic
Peer
Applications might make conflicting changes to data on different servers
Peers may be out of date before updates are received and applied
Updates are propagated between peers

Figure 12
A peer-to-peer replication cluster

Improving Scalability and Reducing Network Latency

Scalability is a primary objective of many NoSQL databases. The most common strategy used is sharding; data is evenly partitioned amongst separate databases, and each database is stored on a separate server. As the workload increases over time, you can add further servers and repartition the data across the new servers to handle the additional load.

Sharding is not so useful for graph databases. The highly connected nature of nodes and edges in a typical graph database can make it difficult to partition the data effectively. Many graph databases do not provide facilities for edges to reference nodes in different databases. For these databases, scaling up rather than scaling out may be a better option.

Different NoSQL databases provide varying implementations of sharding; a few adopt a *Shared Nothing* approach where each shard is an autonomous database and the sharding logic is implemented by the applications themselves. Many provide a more federated approach, also known as *Auto Sharding* in some NoSQL databases, where the database software itself takes on responsibility for directing data to the appropriate shard.

It is always better to explicitly design a NoSQL database to take advantage of sharding rather than try and shard an existing database, otherwise you may lose the benefits of sharding by requiring that your applications retrieve data from multiple servers. You cannot simply bolt scalability onto a NoSQL database!

When you shard data, it is helpful to try and keep data that is accessed together in the same shard. In NoSQL terms, these means that you should design your aggregates with sharding in mind and understand the relationships that might occur between aggregated data to avoid splitting information that is frequently accessed as part of a single logical operation across different servers.

Sharding can also help to decrease latency. If you have a widely dispersed set of users, you can host the shards containing the data that users are most likely to access at sites geographically close to those users. For example, to reduce the network latency that occurs when a user logs in and provides their credentials, aggregates holding the details of users located in Boston could be hosted at a server in New England, while aggregates containing the details of users who live in San Francisco could be stored on a server in California. Again, this strategy requires that you pay close attention to the design of your aggregates so that you can more easily partition data in this manner.

It is important to understand that while sharding can improve scalability and reduce network latency, it does not improve availability. For this reason, many NoSQL databases combine sharding with replication; each shard can be replicated following the primary/secondary or peer-to-peer replication topologies described in the previous section.

Sharding and replication are integral to many NoSQL solutions, but an increasing number of relational database systems also provide these features. However, the implementations in relational solutions tend to be more complex (and expensive) than the equivalent functionality in most NoSQL systems.

Chapter 3, "Implementing a Relational Database," discusses how to apply sharding to a relational database. Chapter 4, "Implementing a Key-Value Data Store," Chapter 5, "Implementing a Document Database," and Chapter 6, "Implementing a Column-Family Database" describe possible sharding strategies for NoSQL databases.

Improving Consistency

Replication and sharding are important strategies that can help to ensure good performance and availability, but at the increased risk of inconsistency between replicated data. As described throughout this chapter, most NoSQL databases provide eventual rather than immediate consistency, and you should design your applications based around this model of working. However, many NoSQL databases include features that help to reduce the likelihood of inconsistent data being presented to an application. These strategies include defining quorums, and using data versioning.

Some NoSQL databases implement ACID transactions. In these cases, write consistency is guaranteed.

Defining Read and Write Quorums

Inconsistencies occur if data is replicated across several servers, but not all of those servers agree on the current value of that data. Quorums help to reduce inconsistencies by ensuring that a significant subset of the servers in a replication cluster (a *read quorum*) agree on the value of a data item. When an application performs a query, the data values stored by the read quorum are returned. However, for this strategy to be successful, when an application writes data to a database it must be sure that this data has been successfully stored by a *write quorum* of the servers in the cluster.

Ideally, the use of quorums, and the number of servers that constitute a read or write quorum in a cluster, should be totally transparent to applications reading and writing data; they should simply be able to write data to a cluster and read from a cluster without requiring any knowledge of how many servers are being updated or interrogated. Behind the scenes, a typical read operation will retrieve data from as many servers as necessary until the number specified by the read quorum can agree on a value. Similarly, a write operation will store the data to as many servers as are specified by the write quorum, and will not complete until all of these servers indicate that the data has been written successfully, or that the data cannot be saved due to a conflict.

Many NoSQL databases support such transparent configurations, and in many cases an administrator can silently add or remove servers from a quorum without interrupting applications. However, it is important to understand that the more servers you include in a write quorum the more servers will need to be updated before the database can indicate a successful write operation to an application, and this can in turn have an impact on the performance of the database and the latency of your application. The same is true for read operations; the bigger the read quorum, the more servers have to respond with the same data before the read operation is considered successful.

The number of servers in a quorum can vary between read operations and write operations, and a quorum does not have to constitute a majority of servers in a replication cluster. For example, in a peer-to-peer replication cluster, if you judge that inconsistencies are unlikely to occur (the chances of two applications updating the exactly same data within a short period of each other is small), you could specify that as long as two servers in the cluster agree on the value of a data item then that value should be returned. You can apply a similar principle to write operations.

Versioning Data and Resolving Conflicts

Even in a system that implements ACID transactions, or that provides read and write quorums to maximize data consistency, an application can still end up viewing or updating inconsistent data. For example, application A could retrieve a data value from a database, modify it, and then store it back in the database. However, between retrieving the data and storing the modified version, application B may have read the same data and changed it to a different value. If application A saves its data, it will have overwritten the change made by application B (it is not even aware that application B had changed the data). The result is a *lost update*.

A lost update may not be a problem to an application, depending on the nature of the application and the context in which it is running. In other cases, it may be critical. Many relational database management systems reduce the possibility of lost updates by implementing pessimistic locking; when an application reads data, that data is locked and remains locked until the end of the transaction. Another application attempting to read the same data is blocked until the lock is released, whereupon it will see any changes made to that data by the first transaction. However, pessimistic locking schemes can severely impact performance and throughput of an application, so most applications prefer to use an optimistic locking scheme, as follows:

1. When an application reads a data item it also retrieves a piece of version information.
2. If the application attempts to update the data item in the database, it re-reads the version information from the database to make sure it has not changed.
3. If the version information is unchanged, the application saves the modified data back to the database together with new version information.
4. If the version information is different, the application retrieves the latest value of the data from the database (another application must have changed it since it was last read), and then returns to step 2.

This strategy still has a small window of opportunity to lose an update (the data could be changed by another application between steps 2 and 3). Some NoSQL databases provide atomic compare-and-set operations that can eliminate this possibility; they momentarily lock data during steps 2 and 3 in the expectation that these tasks will be performed quickly.

The way in which different NoSQL databases format version information can vary significantly, and some implementations (mainly column-family databases) enable you to store multiple versions of the same data in an aggregate together with a timestamp indicating when that version was created. This approach enables you to retain a history of the modifications made to a data item, and if necessary an application can use this information to help resolve conflicts in inconsistent data (if an application reads different values for the same data from different servers in a replication cluster, it can use the history information to determine which is the more recent version, and possibly update the outdated information).

Timestamping depends on different servers in a cluster remaining time-synchronized with each other, and this strategy may not work well if they drift slightly or the servers are remote from each other (it takes time to transmit data over a network). An approach to conflict detection commonly implemented by peer-to-peer replication clusters is to use *vector clocks*. Each server in the cluster can maintain a count of the number of times it has updated an item, and include this information when it synchronizes updates with other servers in the cluster. For example, in a two server cluster comprising servers A and B, if server A updates an item it can store a *vector* containing the counter "A:1" locally with this item. When the update is synchronized with server B it will save the same vector with its local copy of the data. If server A makes another modification to the same data, it increments its counter in the vector for this item to "A:2." Meanwhile, if server B modifies the same information (before synchronizing with server A), it appends its own counter to the vector stored locally in server B to "A:1, B:1." The next time servers A and B attempt to synchronize, they should observe that the vectors for this data are now different ("A:2" in server A, and "A:1, B:1" in server B). This enables the two servers to discover that a conflict has occurred. However, it does not necessarily indicate how to resolve this conflict; this decision is often deferred customizable logic in the database servers, or to the applications that access the data.

Schemas and Non-Uniformity

When you design a relational database, you decide on a set of tables, and for each table you specify the columns it contains, and for each column you select the data type and range for the information it holds together with any other attributes, such as whether it allows NULL values. This is the schema of the database. Developers building applications that use a relational database have to have at least a passing understanding of the database schema, so that they don't try and store data of the wrong type, or omit mandatory information from an entity. Using an ORM can help to abstract some of this information, but it is unlikely to be able to completely isolate the structure of the database from the applications that use it. Some dependencies or features of the database always seep through and somewhere inside the application or the ORM there will be one or more SQL statements that reference the database schema directly. This coupling of an application to a database can lead to brittle solutions. If the database schema needs to be modified due to changed or new business requirements, it can be difficult to make such modifications without impacting every application that references that database. Some organizations have used strategies such as implementing additional views, triggers, and stored procedures that provide a mapping from the old schema to the new, enabling existing applications to continue functioning unchanged, but each of these items adds an extra level of indirection and they will inevitably have an impact on the performance of applications that use them.

NoSQL databases are frequently described as being schemaless. When you create a NoSQL database, you should pay attention to the structure of the data and the aggregates that will hold this data, but in most cases it is the applications that actually create these structures rather than a separate set of database utilities or schema editors. The simplified APIs exposed by most NoSQL databases typically enable an application to store and retrieve data, but rarely impose any restrictions on what this data actually is. This approach essentially gives applications free reign over the database, and as business requirements change applications can modify the structure of the data that they store.

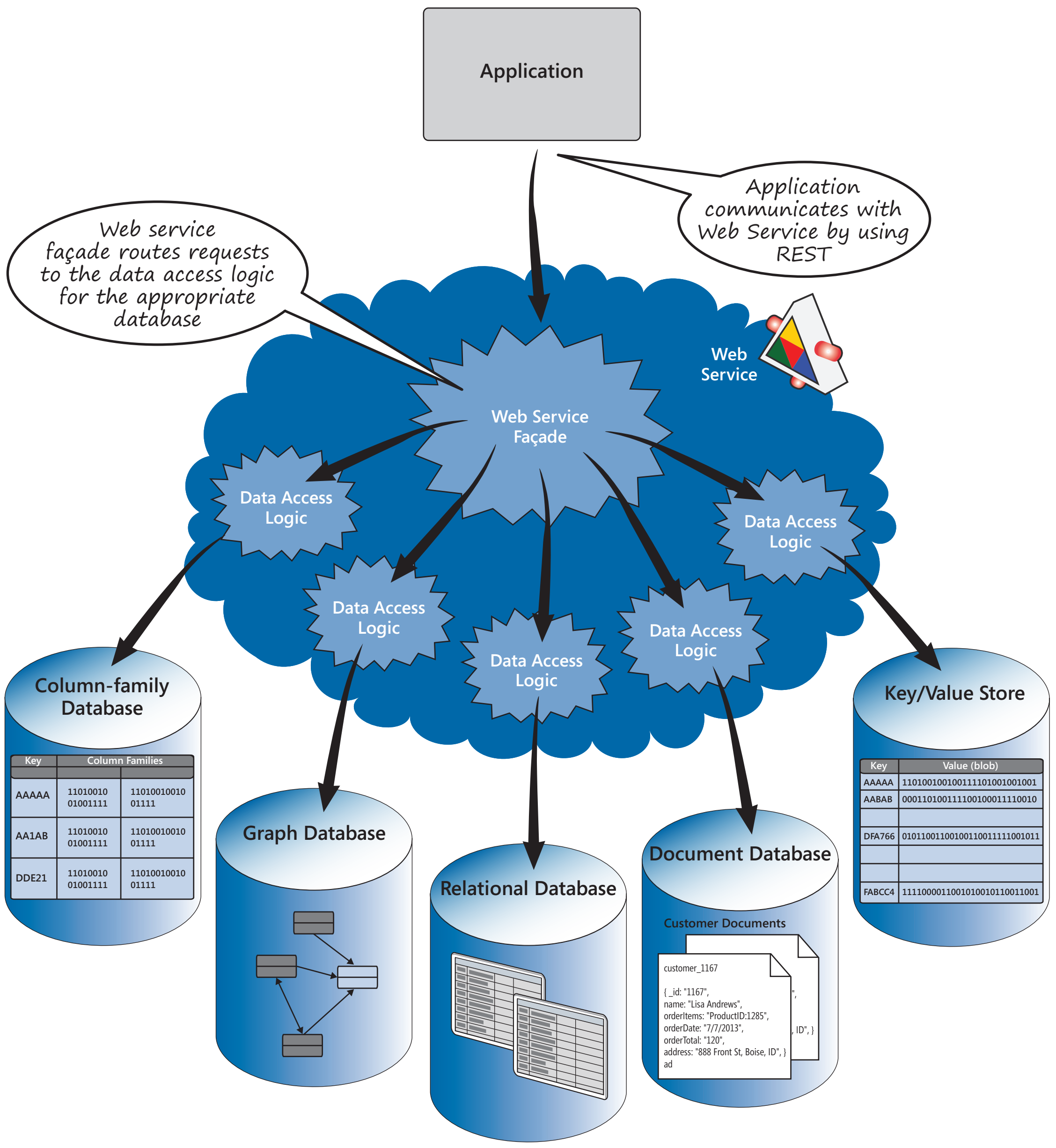
> *Most column-families databases require that you explicitly create a column-family before you can reference it in a row. However, the columns in the column-family do not have to be defined, and can vary from row to row.*

However, this freedom also brings with it a set of responsibilities and problems. If an application is modified to change the structure of the data that it stores, any existing data already in the database will not change, and the database ends up holding a non-uniform set of data. Therefore, applications must be prepared to handle varying data formats or they might risk losing information (or crashing). This approach can complicate the data access logic in an application, and requires a very disciplined approach to managing application development.

NoSQL databases are often described as being schemaless. What this really means is that the responsibility for managing the structure of data has moved from the databases to the applications that use them.

Integrating NoSQL Databases into a Polyglot Solution

The code for numerous NoSQL databases has been made available for public download, and the IT departments in many organizations are increasingly looking at these databases to resolve their own data storage problems. The major task is integrating this disparate software into a cohesive system. In the relational world, the database was viewed as the integration point for applications, but the dissimilarity of the various data structures means that this approach is clearly not viable for NoSQL solutions. Instead, a common technique is to implement a web service as a façade; the web service provides an interface that exposes business operations and converts requests into operations that communicate with the appropriate database. The web service can receive and send requests in a standard format, such as REST, and serialize data in a well-understood and portable format, such as JSON. Additionally, the web service can handle cross-cutting concerns such as security and inter-database integrity. Following this strategy, you can incorporate any number of databases, or switch to a different implementation should one database no longer meet the requirements of your organization. If you retain the same interface to the web service, applications that connect to the web service should be unaffected by such a change. Figure 13 shows the structure of this solution:



Key	Column Families	
AAAAA	11010010 01001111	11010010010 01111
AA1AB	11010010 01001111	11010010010 01111
DDE21	11010010 01001111	11010010010 01111

```
customer_1167

{ _id: "1167",
name: "Lisa Andrews",
orderItems: "ProductID:1285",
orderDate: "7/7/2013",
orderTotal: "120",
address: "888 Front St, Boise, ID", }
ad
```

Key	Value (blob)
AAAAA	1101001001001111010010010 01
AABAB	00011010011110010001111 0010
DFA766	0101100110010011001111001011
FABCC4	1111000011001010010110011001

FIGURE 13
Implementing a web service as the integration point for multiple NoSQL databases

However, although this model is conceptually simple, there are several challenges that you may need to resolve to construct a practical solution. For example, how do you maintain consistency and integrity across different databases if your application expects operations to be atomic? These issues are discussed in Chapter 8, "Building a Polyglot Solution."

Summary

Relational databases provide an excellent mechanism for storing and accessing data in a very generalized manner, but this generalization can also be a weakness. In some cases, applications need to perform very specific types of operations that require data to be accessed in certain ways, and the relational model might not always provide the most optimal way to meet these requirements. NoSQL databases are designed to handle these situations.

A variety of NoSQL databases are available, each intended to focus on a particular data storage and access strategy. While a typical NoSQL database might not be as comprehensive as a relational database, its focus on a well-defined range of tasks enables it to be highly optimized for those tasks. The key to success is to understand the features of different NoSQL databases, and then use these features to implement a repository that matches the specific requirements of your applications.

The white paper, "NoSQL and the Windows Azure Platform" provides a good comparison between NoSQL and relational databases, focusing on Microsoft technologies. You can download this white paper from the Microsoft Download Center.

The following table compares the common features and functionality of relational (SQL) and non-relational (NoSQL) databases, highlighting the strengths of the different approaches to structuring the data and the types of applications for which each technology is most appropriate.

Feature/Functionality	SQL	NoSQL
Standardization and interoperability	Mature technology, well understood. Subject to many ANSI and ISO standards, enabling interoperability between conforming implementations. Standard APIs (ODBC, SQL/CLI, and so on) enable applications to operate with databases from different vendors.	New technologies with no common overarching model or standardization. Each NoSQL database vendor defines their own APIs and data formats. Minimal interoperability between different vendors is possible at the database level. Application code written for one NoSQL database is unlikely to work against a NoSQL database from a different vendor.
Storing complex data	Data for complex objects is often normalized into several entities spanning multiple tables to reduce data redundancy. It may require intricate SQL queries to reconstruct the data for a single complex object from these tables. ORMs can abstract some of the details, but this extra layer can lead to inefficiencies.	Can store data for complex objects without splitting across aggregates, enabling a much simpler mapping between objects used by applications and the data stored in the database. This enables fast query access at the possible cost of additional complexity in application code when storing or updating denormalized data.
Performing queries	Relational model is very generalized. SQL supports ad-hoc queries by joining tables and using subqueries. Very good at summarizing and grouping relational data. Less good at handling complex non-relational queries.	Databases are designed to optimize specific queries, most commonly retrieving data from a single aggregate by using a key. Most NoSQL databases do not support data retrieval from multiple aggregates within the same query. Summarizing and grouping data may require implementing map/reduce functions to work efficiently. Graph databases can be excellent for handling complex non-relational queries.

Feature/Functionality	SQL	NoSQL
Scalability	Most suited to scale-up scenarios rather than scale out due to the performance of distributed transactions and cross-database queries. Some relational vendors support clustering and sharding as optional extensions.	Mostly designed with support for scaling out built-in. Many NoSQL databases provide seamless support for clustering and sharding data.
Performance with large datasets	Can require significant tuning to read from or write to large datasets efficiently.	Designed to be very efficient for handling large datasets.
Data consistency and transactions	Designed to be highly consistent (ACID), but at the cost of transactional performance. Transactional consistency can slow down operations in a distributed environment.	Designed to be eventually consistent (BASE) in a distributed environment. Some support for ACID properties for updates within an aggregate, but cross-aggregate operations are not guaranteed to be atomic. Careful use of quorums can help to reduce instances of inconsistency.
Integration	Relational databases can be easily shared between different applications. The database acts as the point of integration between applications.	Databases are usually designed specifically to support a single application. Application integration is usually handled by application code.

More Information

All links in this book are accessible from the book's online bibliography available at: *http://msdn.microsoft.com/en-us/library/dn320459.aspx*.

- You can download the guide "Building Hybrid Applications in the Cloud on Windows Azure" from the patterns & practices website at *http://msdn.microsoft.com/library/hh871440.aspx*.
- The guide "CQRS Journey" is available on the patterns & practices website at *http://msdn.microsoft.com/library/jj554200.aspx*.
- The article "Getting Acquainted with NoSQL on Windows Azure" is available on the ISV Developer Community blog at *http://blogs.msdn.com/b/usisvde/archive/2012/04/05/getting-acquainted-with-nosql-on-windows-azure.aspx*.
- You can download the white paper "NoSQL and the Windows Azure Platform" from the Microsoft Download Center at *http://download.microsoft.com/download/9/E/9/9E9F240D-0EB6-472E-B4DE-6D9FCBB505DD/Windows%20Azure%20No%20SQL%20White%20Paper.pdf*.
- You can find more information about NoSQL databases at *http://nosql-database.org/*.
- You can find detailed information about using map/reduce techniques to summarize data at *http://mapreduce.org/*.

2 The Adventure Works Scenario

Adventure Works is a large multinational manufacturing company that builds and sells metal and composite bicycles and associated products to the North American, European, and Asian commercial markets. The company is based in Bothell, Washington, with 290 employees and several regional sales teams located in various countries throughout their market base. To support these distributed sales teams, and also to enable customers to purchase items directly, Adventure Works developed an online Shopping application that allows users to browse the products that Adventure Works manufactures and place orders for these products.

This chapter describes the business requirements of the Adventure Works Shopping application, and summarizes the architecture of their solution.

How a Customer Uses the Shopping Application

The purpose of the Adventure Works Shopping application is to enable customers to view the items that the company has for sale, place orders for these items, and pay for them. The customer's experience is paramount, so the designers at Adventure Works chose to implement the system as an easy to use, forms-based web application. The Shopping application implements the features typical of many modern online commerce systems. It enables customers to browse products, add products to a shopping cart, and then place an order for these items. Customers can also view a list of all orders that they have placed.

When a customer has ordered and paid for the goods, they are picked from the Adventure Works warehouse, packaged, and dispatched. This part of the process is outside of the scope of the Shopping application and is handled by other systems internally inside Adventure Works.

The following sections describe the primary business use cases of the Shopping application in more detail.

Browsing Products

The products that Adventure Works sells are grouped into categories (Accessories, Bikes, Clothing, and Components), and each category is divided into a set of subcategories. For example, the Accessories category contains subcategories for Bike Racks, Bike Stands, Bottles and Cages, Lights, Locks, Helmets, and so on. Each subcategory contains one or more products. No products are held in more than one subcategory. The result is a tree-structured hierarchical relationship between categories, subcategories, and products. The Shopping application exploits this relationship to provide easy navigation of products.

When the user starts the application it displays a list of categories as shown by the screenshot in Figure 1.

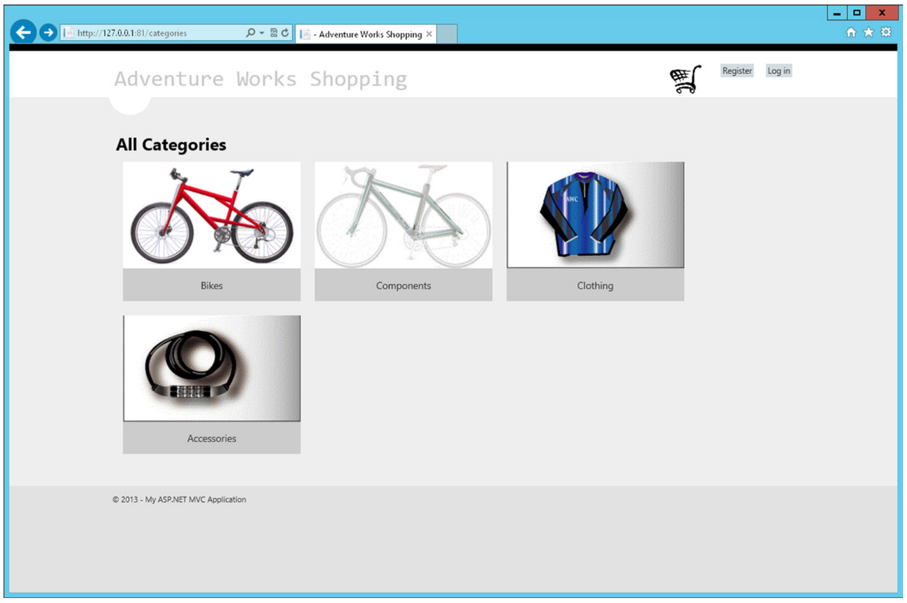

http://127.0.0.1:81/categories
- Adventure Works Shopping
Adventure Works Shopping
Register
Log in
All Categories
Bikes
Components
Clothing
Accessories
© 2013 - My ASP.NET MVC Application

FIGURE 1
The categories displayed when the Shopping application starts

A customer can click a category to display the subcategories for that category, and then click a subcategory to browse the products in that subcategory. Figure 2 shows the result when the user clicks the **Bikes** category and then clicks the **Road Bikes** subcategory.

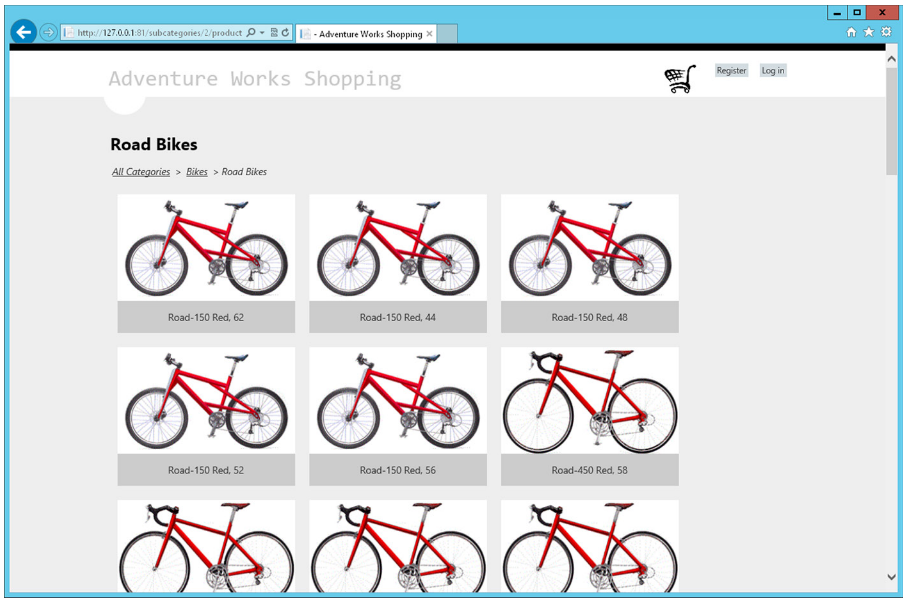
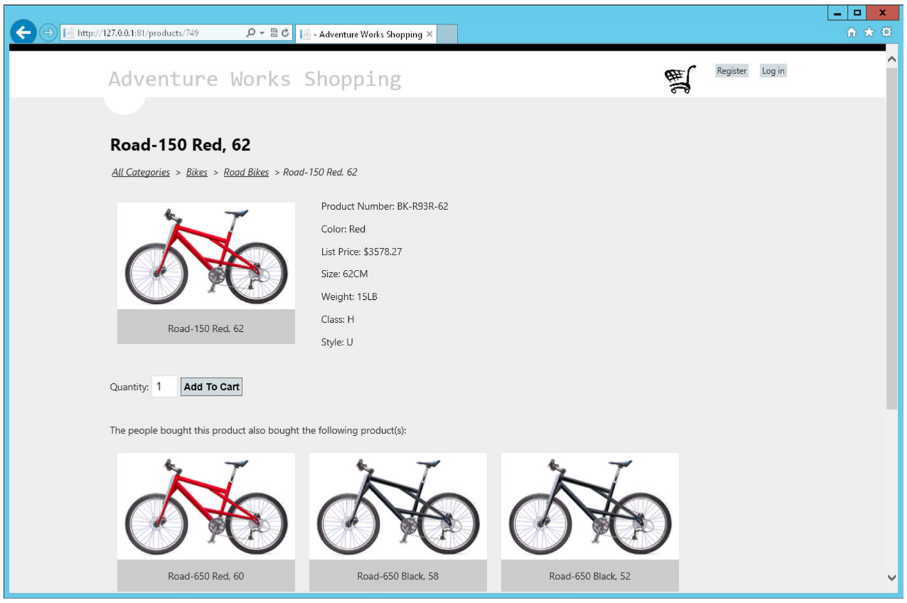


FIGURE 2
The subcategories in the Road Bikes category

Clicking a product displays the details for that product. Figure 3 shows the information displayed for the **Road-150 Red, 62** bicycle. The application also retrieves and displays information about related products that may be of interest to the customer, based on purchases made by other customers who also bought this bike:



Road-150 Red, 62

All Categories > Bikes > Road Bikes > Road-150 Red, 62

Road-150 Red, 62

Product Number: BK-R93R-62

Color: Red

List Price: $3578.27

Size: 62CM

Weight: 15LB

Class: H

Style: U

Quantity: 1 Add To Cart

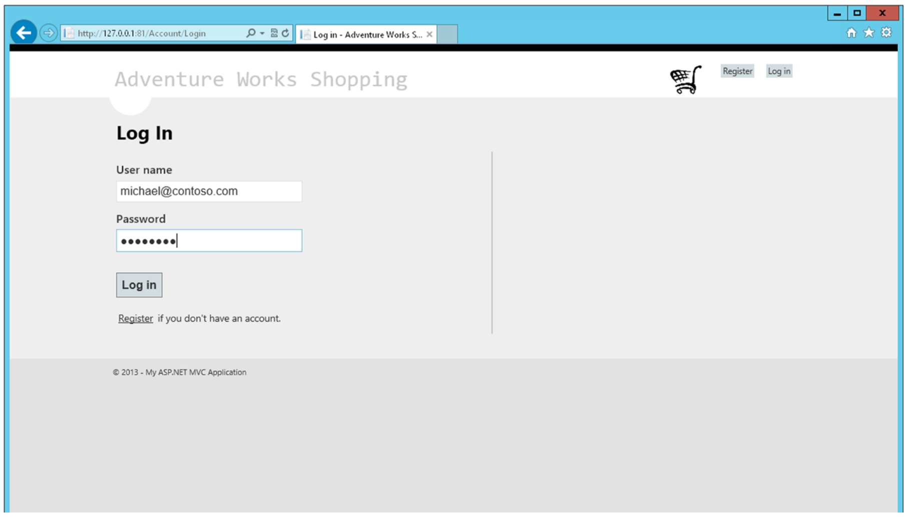
The people bought this product also bought the following product(s):

Road-650 Red, 60 | Road-650 Black, 58 | Road-650 Black, 52

Figure 3
The details displayed for a product

Logging in and Registering with the Shopping Application

The application allows customers to browse products anonymously. However, before customers can actually add items to their shopping cart and place an order they must log in. Customers can click the **Log in** button and provide their username and password to authenticate themselves, as shown in Figure 4.


http://127.0.0.1:81/Account/Login
Log in - Adventure Works S...
Adventure Works Shopping
Register
Log in

Log In

User name

michael@contoso.com

Password

••••••••

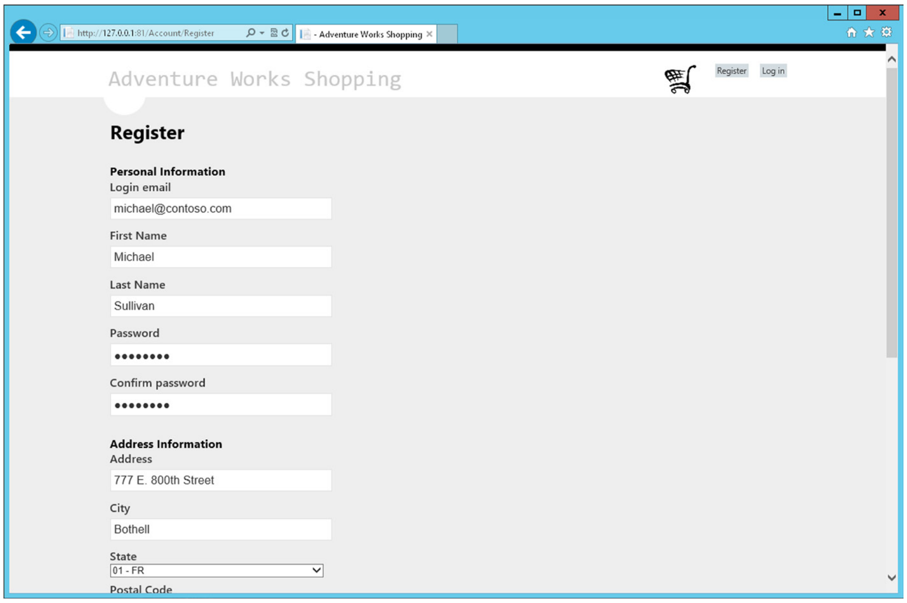
Log in

Register if you don't have an account.

© 2013 - My ASP.NET MVC Application

Figure 4
A customer logging in

If the customer is new and does not yet have an account, the customer can click the **Register** button. The Register screen prompts the customer to provide details such as an email address (used to identify the customer when they log in), a password, a billing address, and credit card information. Figure 5 illustrates the page that enables a customer to register their information.



Register

Personal Information

Login email	michael@contoso.com
First Name	Michael
Last Name	Sullivan
Password	••••••••
Confirm password	••••••••

Address Information

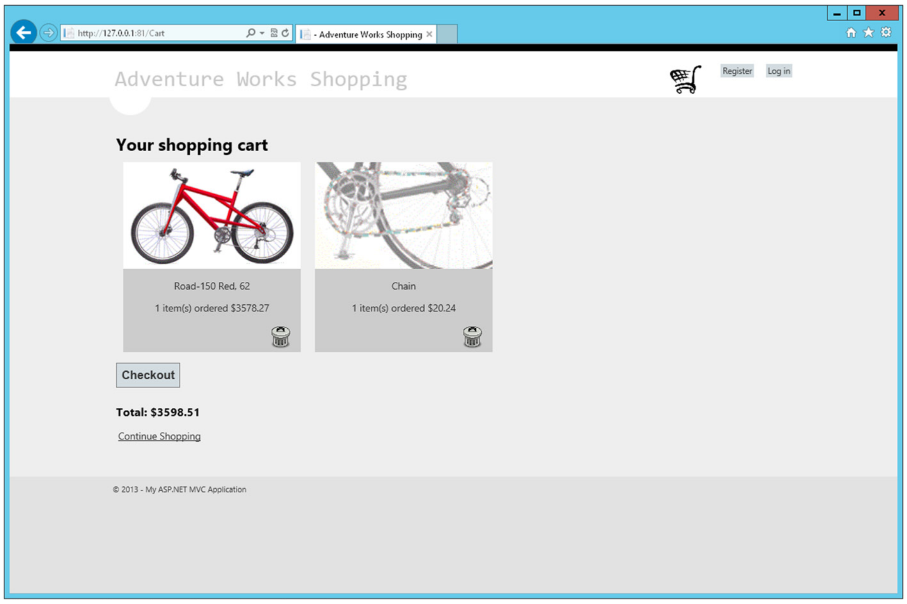
Address	777 E. 800th Street
City	Bothell
State	01 - FR
Postal Code	

FIGURE 5
Registering as a new customer

Adding Items to the Shopping Cart

Once a customer has logged in, the customer can add an item to the shopping cart by clicking **Add To Cart**. The customer can also specify the number of items to order on this page. The data in the shopping cart is saved to the database. This feature adds a degree of robustness to the application; if the user's browser should crash or lose connectivity to the Shopping application, the shopping cart is not lost. When the customer connects to the Shopping application again, they can carry on shopping and the customer can add further items to their cart.

The user can click the Shopping Cart icon at any time to review the contents of their shopping cart. This page also enables the customer to remove an item from their cart, or check out and place an order for the items in their cart. Figure 6 shows the shopping cart page in the Shopping application.


http://127.0.0.1:81/Cart

- Adventure Works Shopping

Adventure Works Shopping

Register Log in

Your shopping cart

Road-150 Red, 62
1 item(s) ordered $3578.27

Chain
1 item(s) ordered $20.24

Checkout

Total: $3598.51

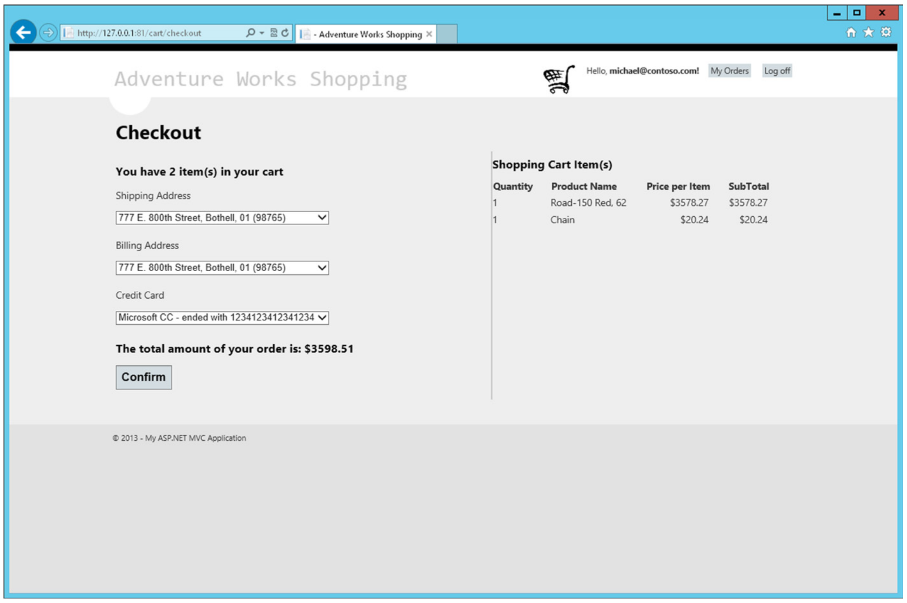
Continue Shopping

© 2013 - My ASP.NET MVC Application

Figure 6
The shopping cart for a customer

Placing an Order

To place an order, the customer can click **Checkout** on the shopping cart page. At this point, the customer must confirm information such as a billing address and a credit card. The image in Figure 7 shows this page.


http://127.0.0.1:81/cart/checkout

- Adventure Works Shopping

Adventure Works Shopping

Hello, **michael@contoso.com!** My Orders Log off

Checkout

You have 2 item(s) in your cart

Shipping Address
777 E. 800th Street, Bothell, 01 (98765)

Billing Address
777 E. 800th Street, Bothell, 01 (98765)

Credit Card
Microsoft CC - ended with 1234123412341234

The total amount of your order is: $3598.51

Confirm

Shopping Cart Item(s)

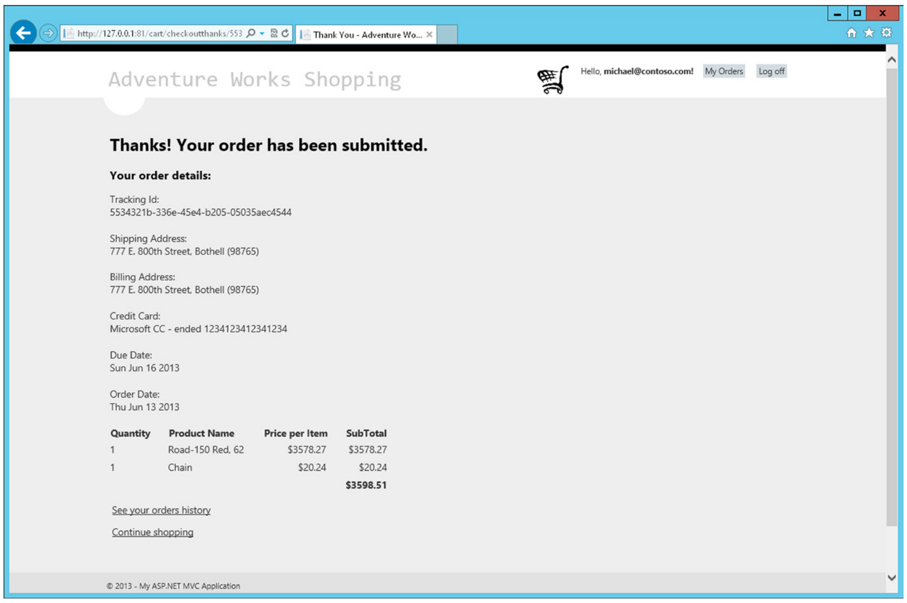
Quantity	Product Name	Price per Item	SubTotal
1	Road-150 Red, 62	$3578.27	$3578.27
1	Chain	$20.24	$20.24

© 2013 - My ASP.NET MVC Application

Figure 7
Specifying the billing information for an order

When the customer clicks **Confirm**, the order is placed and the customer receives confirmation of the order details, as shown in Figure 8.

In the real world, when the customer places an order, the items in the order are picked using a separate warehousing system and the order is then shipped using a dispatching system. The Shopping application has to integrate with these warehousing and dispatching systems although they are outside the scope of the Shopping application and they are not included in the sample solution provided with this guide.


http://127.0.0.1:81/cart/checkoutthanks/553
Thank You - Adventure Wo...
Adventure Works Shopping
Hello, michael@contoso.com!
My Orders
Log off

Thanks! Your order has been submitted.

Your order details:

Tracking Id:
5534321b-336e-45e4-b205-05035aec4544

Shipping Address:
777 E. 800th Street, Bothell (98765)

Billing Address:
777 E. 800th Street, Bothell (98765)

Credit Card:
Microsoft CC - ended 1234123412341234

Due Date:
Sun Jun 16 2013

Order Date:
Thu Jun 13 2013

Quantity	Product Name	Price per Item	SubTotal
1	Road-150 Red, 62	$3578.27	$3578.27
1	Chain	$20.24	$20.24
			$3598.51

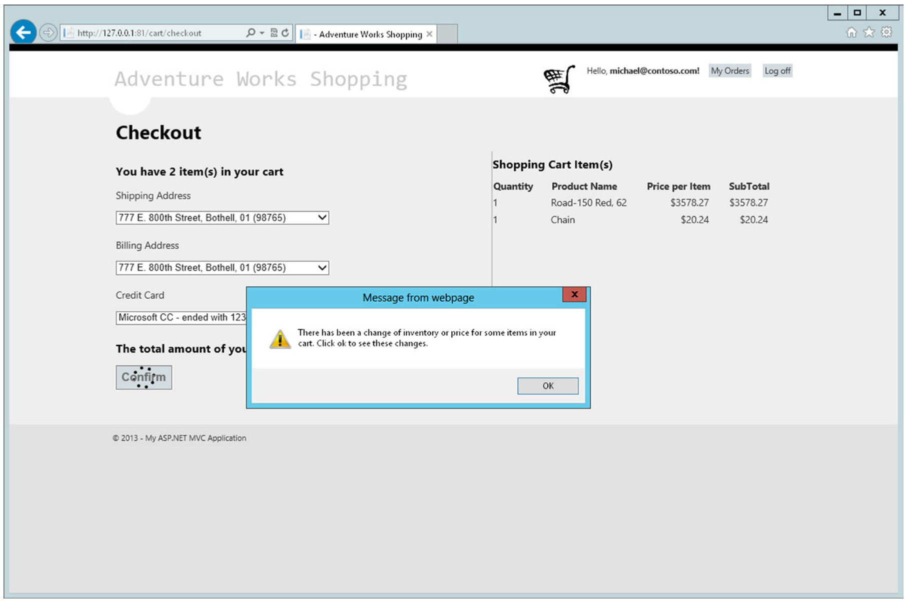
See your orders history

Continue shopping

© 2013 - My ASP.NET MVC Application

FIGURE 8
The confirmation page showing the details of an order

Note that when a customer places items in their shopping cart, the contents of the cart are saved, and the user can close the browser without placing an order. The next time the customer logs in, their cart is restored, and they can add further items or simply check out using the items currently in their cart. However, it is possible that the price of one or more items might have changed since the customer added them to the cart. When the customer clicks **Confirm** in the **Checkout** page, the Shopping application verifies each item to ensure that its price has not been updated. If there are any differences, the Shopping application displays the message shown in Figure 9.



http://127.0.0.1:81/cart/checkout

- Adventure Works Shopping

Adventure Works Shopping

Hello, michael@contoso.com! My Orders Log off

Checkout

You have 2 item(s) in your cart

Shipping Address

777 E. 800th Street, Bothell, 01 (98765)

Billing Address

777 E. 800th Street, Bothell, 01 (98765)

Credit Card

Microsoft CC - ended with 123

The total amount of you

Confirm

Shopping Cart Item(s)

Quantity	Product Name	Price per Item	SubTotal
1	Road-150 Red, 62	$3578.27	$3578.27
1	Chain	$20.24	$20.24

Message from webpage

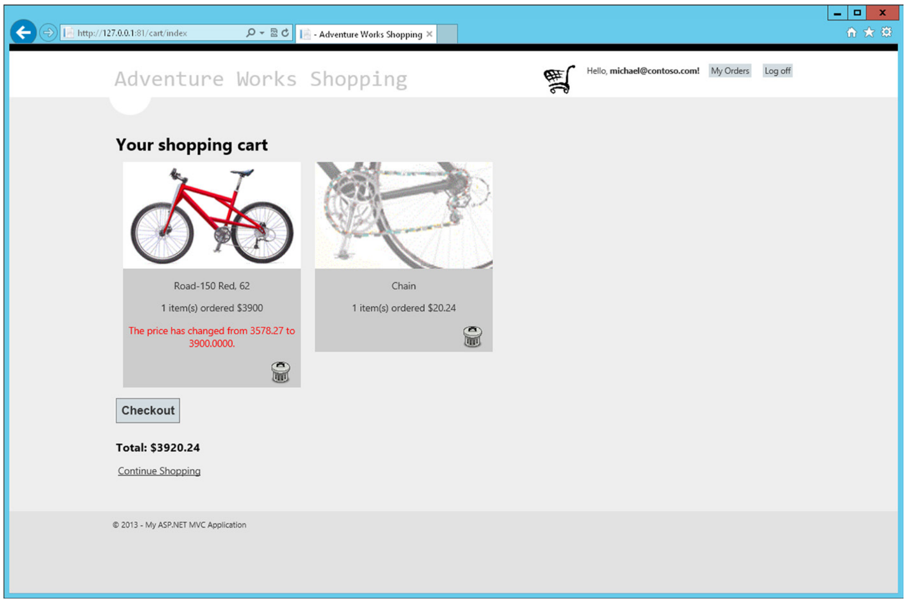
There has been a change of inventory or price for some items in your cart. Click ok to see these changes.

OK

© 2013 - My ASP.NET MVC Application

Figure 9
Informing the customer that the price of one or more items has changed

The customer can click OK to view their shopping cart, which indicated which items have changed, as shown in Figure 10.



http://127.0.0.1:81/cart/index

- Adventure Works Shopping

Adventure Works Shopping

Hello, michael@contoso.com! My Orders Log off

Your shopping cart

Road-150 Red, 62

1 item(s) ordered $3900

The price has changed from 3578.27 to 3900.0000.

Chain

1 item(s) ordered $20.24

Checkout

Total: $3920.24

Continue Shopping

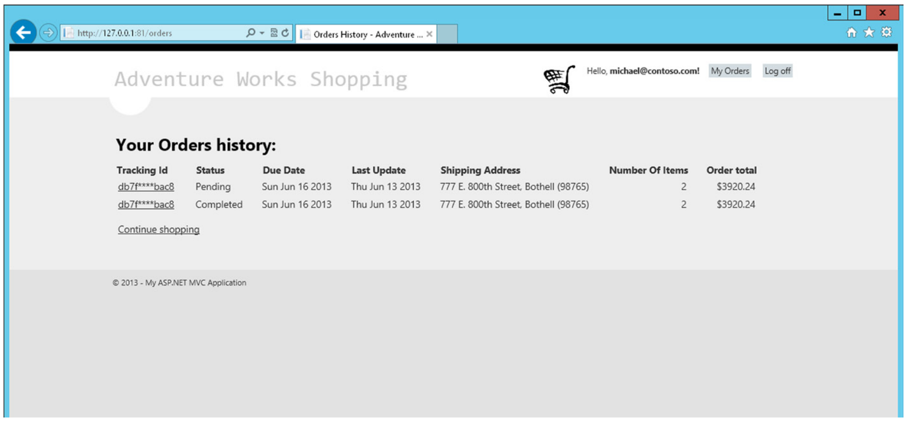
© 2013 - My ASP.NET MVC Application

Figure 10
The shopping cart indicating which items have a different price

If the customer is happy to accept the new price, they can click **Checkout**, and the process proceeds as described earlier.

Viewing Order History

A customer can view an audit trail listing any orders that they have previously placed, and any modifications made to those orders since they were placed, by clicking **My Orders** in the menu bar. Figure 11 shows the order history for a customer. Note that in the sample solution, when an order is created and added to the database, its status is set to **Pending**. Later, when the order has been processed its status is set to **Completed**. The order history for an order therefore displays at least two entries (one for each status). If an order is subjected to any further modifications, then the order history will contain entries for each of these changes as well.


http://127.0.0.1:81/orders
Orders History - Adventure ...
Adventure Works Shopping
Hello, michael@contoso.com!
My Orders
Log off

Your Orders history:

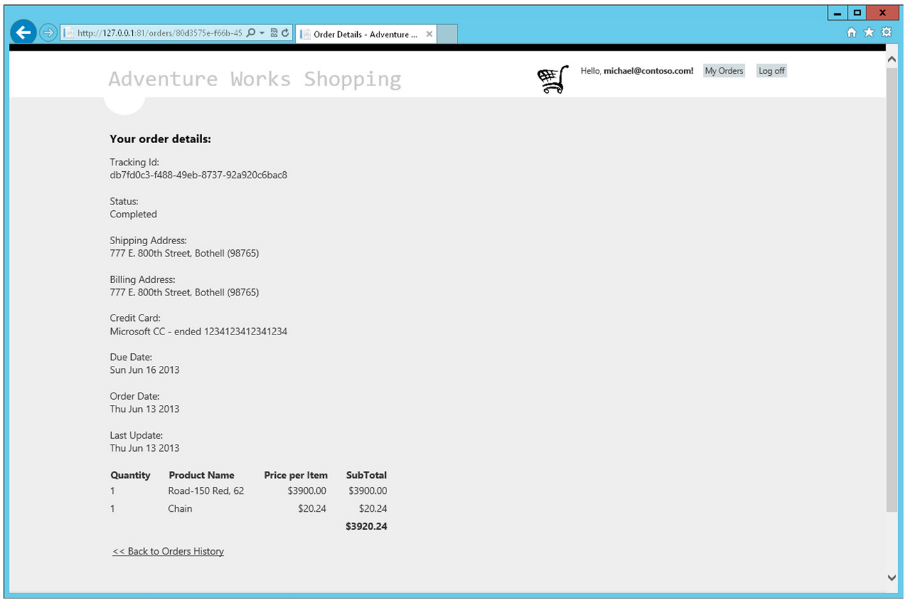
Tracking Id	Status	Due Date	Last Update	Shipping Address	Number Of Items	Order total
db7f****bac8	Pending	Sun Jun 16 2013	Thu Jun 13 2013	777 E. 800th Street, Bothell (98765)	2	$3920.24
db7f****bac8	Completed	Sun Jun 16 2013	Thu Jun 13 2013	777 E. 800th Street, Bothell (98765)	2	$3920.24

Continue shopping

© 2013 - My ASP.NET MVC Application

FIGURE 11
The order history for a customer

The customer can click the **Tracking Id** link for any order history record to view the details of that record, as shown in Figure 12.


http://127.0.0.1:81/orders/80d3575e-f66b-45
Order Details - Adventure ...
Adventure Works Shopping
Hello, michael@contoso.com!
My Orders
Log off

Your order details:

Tracking Id:
db7fd0c3-f488-49eb-8737-92a920c6bac8

Status:
Completed

Shipping Address:
777 E. 800th Street, Bothell (98765)

Billing Address:
777 E. 800th Street, Bothell (98765)

Credit Card:
Microsoft CC - ended 1234123412341234

Due Date:
Sun Jun 16 2013

Order Date:
Thu Jun 13 2013

Last Update:
Thu Jun 13 2013

Quantity	Product Name	Price per Item	SubTotal
1	Road-150 Red, 62	$3900.00	$3900.00
1	Chain	$20.24	$20.24
			$3920.24

<< Back to Orders History

FIGURE 12
An order history record

How the Shopping Application Stores Information

A traditional web application stores data in a relational database and the business logic of the web application retrieves, displays, and modifies this information as the user browses the data. However, the developers at Adventure Works felt that this approach might not scale well for the Shopping application; they anticipate that the application will attract many thousands of concurrent users each placing many orders. For this reason, the developers decided to optimize the application by using a combination of data storage technologies, each one tuned to meet the specific business requirements of the various use cases, as follows:

- Customer information is stored by using SQL Server. The application must maintain customer confidentiality and security. SQL Server easily integrates into the security framework provided by the technologies used to build the Shopping application, and so it provides the ideal repository for holding the details of customers and their passwords.
- Shopping cart information is saved in a simple key-value store that can be accessed by using the customer's unique identifier as the lookup key. This mechanism is very quick and highly scalable.
- The details of products do not change very often, and in the Shopping application this data is read-only, this information can be stored and retrieved efficiently by using a document database.
- Customers can place multiple orders, and each order can contain many line items. The warehousing and dispatching systems that actually arrange for goods to be picked and shipped to the customer frequently modify the status information and other details of an order. These systems, which are outside the scope of the Shopping application, use SQL Server to meet the OLTP requirements of the business. Therefore the developers decided to save the details of orders to the same database to enable the Shopping application to integrate seamlessly with these systems.
- Once an order has been placed, the details of the order constitute a historical document, and the system must record a full audit trail of any changes made to the order. Therefore, the developers decided to save order information in two places; the active orders are held in SQL Server as described previously, but a historical record of each order is stored in a document database as a series of order history documents. Each order history document is held in a fully denormalized structure to enable the details of the order to be retrieved quickly. As changes occur to an order, a new order history document containing the updated order information is added to the document database, together with a date stamp indicating when the change occurred. In this way, it is possible to obtain the complete history that lists all the changes that have been made to an order.
- Product recommendations require maintaining complex relationships between products, customers, and orders. This type of information is best stored in a graph database.

The chapters in this guide describe how the developers at Adventure Works designed and implemented these data stores. These chapters also illustrate good practice for building applications that utilize these data stores.

This guide concentrates on describing best practices for using SQL and NoSQL databases to store and retrieve data. It does not provide strategies for migrating data from a relational database to a NoSQL database.

The Architecture of the Shopping Application

The developers at Adventure Works are knowledgeable about various Microsoft products and technologies, including the .NET Framework, ASP.NET MVC and the ASP.NET Web API, SQL Server, and Visual Studio, so they decided to implement the solution by using these technologies.

The solution comprises three main components:

- The customer-facing user interface, implemented as an MVC4 web application. This application formats and displays information about products from the database, and enables the user to navigate through the site and place orders. Users can connect to this application using a web browser.

  In this guide, the user interface web application is provided simply as a means to demonstrate that the various data access mechanisms described are functioning correctly. It is possible to replace this user interface with any other application that can formulate the appropriate REST requests and consume REST responses.

- The business logic, implemented as a set of web services in an MVC4 Web API application. This application exposes the data and operations that it supports through a REST (Representational State Transfer) interface. Separating the business logic in this way decouples it from the user interface, minimizing the impact that any changes to the implementation of the user interface will have on this business logic.
- Data storage, implemented by using a combination of SQL Server and NoSQL databases. The designers at Adventure Works selected different repositories to match the business requirements of each of the use cases as described in the previous section. The web services access the data in these databases by using Repository pattern to isolate the database-specific code from the business logic.

Using REST enables Adventure Works to invoke operations and consume the responses by using any web-enabled system that can formulate REST queries, giving them great flexibility if they wish to integrate these services into other parts of their operations. Additionally, if Adventure Works decides to operate through partner organizations in different territories, those partners can implement their own localized user interfaces that generate REST requests and process REST responses.

Appendix A, "How the MvcWebApi Web Service Works," contains detailed information on the structure of the web service and the code that connects to the various repositories.

Building and hosting a web application for a multinational organization such as Adventure Works is a major undertaking and can require a significant investment in hardware, management, and other infrastructure resources. Connectivity and security are also major concerns; users require timely and responsive access, and at the same time the system must maintain the integrity of the data in the database and the privacy of users' information. To support a potentially large number of concurrent requests against an ever-expanding database, the designers at Adventure Works chose to implement the Shopping application in the cloud by using Windows Azure. The web application that implements the user interface and the web service that provides the business logic are deployed as web roles. Additionally the designers chose to deploy the various data stores that the application uses to the cloud. These data stores comprise a combination of Windows Azure SQL Database (*SQL Server in the cloud*), Windows Azure Table Storage, and other NoSQL databases running on virtual machines hosted in the cloud. This environment provides the necessary scalability, elasticity, reliability, security, and performance necessary for supporting a large number of concurrent, distributed users.

Deploying a solution such as this to the cloud minimizes the hardware and support investment that Adventure Works needs to make. They can monitor the volume of traffic to the web application and the web service, and if necessary simply scale the solution to run additional instances with more resources.

As an alternative to using Windows Azure SQL Database, the architects also considered hosting the SQL Server database in a Windows Azure VM. However, they came to the conclusion that Windows Azure SQL Database would provide the scalability that the application requires, but they might review this decision in the future. Chapter 3, "Implementing a Relational Database" provides more information about the benefits and tradeoffs of using Windows Azure SQL Database against running a SQL Server database in a Windows Azure VM.

Figure 13 shows a high-level view of the Shopping application running in the cloud. Customers can connect to the user interface web application over HTTP from any device that provides a web browser.

IMPORTANT

The UI web application generates anti-forgery tokens that help to protect critical operations such as registering as a new customer, logging in, and placing an order (other operations, such as viewing the product catalog, expose data that is in the public domain and are not protected). However, the MvcWebApi web service currently implements only minimal security measures. Any application that knows the URL of the web service can connect to it and send requests. A real-world implementation of this system would include more robust authentication and prevent unauthorized use, but in this reference implementation the additional code would obscure the data access guidance that this application is designed to convey.

Additionally, the application does not include any logging capabilities. If you wish to capture information about events such as failed and successful login attempts, the dates and times when customers placed orders, and other significant actions, you can incorporate this functionality into the web service by using the Semantic Logging Application Block. The *Semantic Logging Application Block* is part of the Microsoft Enterprise Library available from the patterns & practices website.

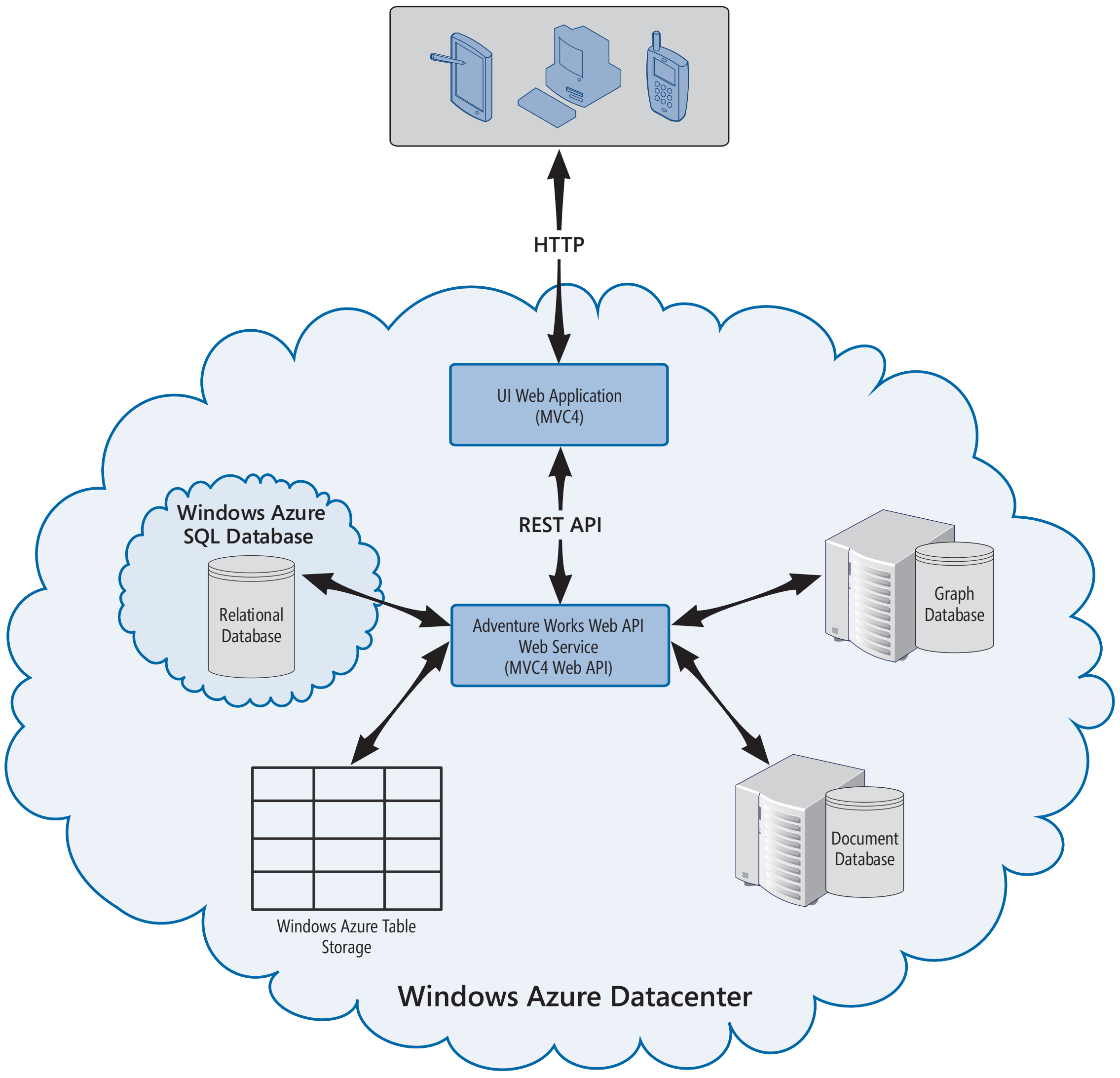

HTTP
UI Web Application (MVC4)
REST API
Windows Azure SQL Database
Relational Database
Adventure Works Web API Web Service (MVC4 Web API)
Graph Database
Windows Azure Table Storage
Document Database
Windows Azure Datacenter

Figure 13

High-level overview of the Adventure Works Shopping application running in the cloud

Summary

For more information about building and deploying applications to the cloud by using Windows Azure, see the patterns & practices guide "*Developing Multi-tenant Applications for the Cloud, 3rd Edition*" available from MSDN.

This chapter has introduced the Shopping application built by Adventure Works. The application consists of an MVC4 web application that acts as the user interface, a web service that provides a REST interface to the web application, and a collection of SQL and NoSQL databases that store the data for the web application.

In the following chapters, you will see how the developers at Adventure Works designed the data stores to optimize access to the data required by the application, and how they implemented the code that accesses this data.

More Information

All links in this book are accessible from the book's online bibliography available at: *http://msdn.microsoft.com/en-us/library/dn320459.aspx*.

- The community site for this guide at *http://dataguidance.codeplex.com/* provides links to online resources The sample code can be downloaded from the Microsoft Download Center at *http://www.microsoft.com/en-us/download/details.aspx?id=40294*.
- The patterns & practices guide "Developing Multi-tenant Applications for the Cloud, 3rd Edition" is available on MSDN at *http://msdn.microsoft.com/en-us/library/ff966499.aspx*.
- The portal with information about Microsoft Windows Azure is at *http://www.microsoft.com/windowsazure/*. It has links to white papers, tools, and many other resources. You can also sign up for a Windows Azure account here.
- You can download the Microsoft Enterprise Library from the patterns & practices website at *http://msdn.microsoft.com/entlib*.

3 Implementing a Relational Database

The Shopping application stores the details of customers by using SQL Server. As described in Chapter 2, *"The Adventure Works Scenario,"* the developers also decided to use SQL Server to store order information, at least initially.

This chapter describes the concerns that Adventure Works addressed in order to design the SQL Server database for the Shopping application. It summarizes the decisions that they made in order to optimize the database to support the business functions of the application, where they deployed the database, and how they designed the code that accesses the database.

Chapter 2, "The Adventure Works Scenario," describes the primary business functions of the Shopping application.

Designing a Relational Database to Support Fast Transactions

SQL Server is an example of a relational database management system (RDBMS). Relational databases first emerged in the early 1970s as a direct result of the research performed by Edgar Codd. Since that time, RDBMSs have become a mainstream technology. The simple-to-understand constructs and the mathematical underpinning of relational theory have caused RDBMSs to be viewed as the repository of choice for many systems. However, it is important to understand how to apply the relational model to your data, and how you application accesses this data, otherwise the result can be a database that performs very poorly and that contains inconsistent information.

A common scenario for using a relational database is as the repository behind on-line transaction processing (OLTP) systems. A well-designed relational database enables your application to insert and update information quickly and easily. Most modern RDBMSs implement fast and efficient strategies that ensure the isolation of transactions and the integrity of the data that participates in these transactions.

This section summarizes some of the strategies that Adventure Works considered to maximize the transactional throughput of the Shopping application.

Normalizing Tables

An important principle that underpins the relational model is that a single piece of information, or fact, should be stored only once. For example, in the Adventure Works database, the fact that the sales tax charged on order SO43659 was 1971.51 is recorded in a single row in the **SalesOrderHeader** table and nowhere else in the database. Additionally, each fact about an item should be independent of any other facts about the same item. Taking order SO43659 as the example again, this order is recorded as having been due to arrive with the customer by July 13, 2005. This information is independent of the sales tax amount, and the due date can change without affecting the sales tax amount.

In a strict implementation of the relational model, independent facts should be recorded in separate tables. In the Adventure Works database, this would mean having a table that holds information about the sales tax amounts (**SalesOrderHeaderTaxAmount**), and another table that stores order due dates (**SalesOrderHeader-DueDate**). Further tables can hold other information; **SalesOrderHeaderOrderDate** could record the date on which each order was placed, and **SalesOrderHeaderTotalDue** could store the total value of the order, for example. A schema that holds such a collection of tables is referred to as being normalized.

The primary advantage of a normalized structure is that it can help to optimize operations that create, update, and delete data in an OLTP-intensive environment, for the following reasons.

- Each piece of information is held only once. There is no need to implement complex, time-consuming logic to search for and modify every copy of the same data.
- A modification only affects a row holding a single piece of data. If a row in a table contains multiple data items, as occurs in a denormalized table, then an update to a single column may block access to other unrelated columns in the same row by other concurrent operations.

  Many RDBMSs implement row versioning to enable multiple concurrent operations to update different columns in the same row in a table. However, this mechanism comes at the cost of additional processing and memory requirements, which can become significant in a high-volume transaction processing environment.

- You have complete flexibility in the schema of your data. If you need to record additional information about an entity, such as the sales person that handled any enquiries about the order, you can create a table to hold this information. Existing orders will be unaffected, and there is no need to add columns to an existing table that allows null values.

Figure 1 illustrates a normalized schema for holding sales order header information. For contrast, Figure 1 also depicts the existing denormalized **SalesOrderHeader** table. Notice that order 44544 does not have a sales tax value recorded; in the normalized schema, there is no row in the **SalesOrderHeaderTaxAmount** table for this order, but in the denormalized **SalesOrderHeader** table the same order has a null value for this data.

> *The SalesOrderHeader table is actually in Third Normal Form (3NF), and depicts the level of partial denormalization commonly implemented by many relational database designers. The section "*Denormalizing Tables*" later in this chapter provides more information about 3NF.*

Normalized Tables

SalesOrderHeaderTaxAmount Table

SalesOrderID	TaxAmount
1	15.28
44280	18.97

SalesOrderHeaderDueDate Table

SalesOrderID	DueDate
1	08/07/2013
44280	08/10/2013
44544	08/11/2013

SalesOrderHeaderOrderDate Table

SalesOrderID	OrderDate
1	01/07/2013
44280	01/10/2013
44544	01/11/2013

SalesOrderHeaderTotalDate Table

SalesOrderID	TotalDate
1	279.99
44280	282.50
44544	155.25

SalesOrderHeader Table (Denormalized)

SalesOrderID	TaxAmount	DueDate	OrderDate	TotalDue
1	15.28	08/07/2013	01/07/2013	279.99
44280	18.97	08/10/2013	01/10/2013	282.50
44544	*NULL*	08/11/2013	01/11/2013	155.25

Figure 1
A fully normalized set of tables holding sales tax amount, order due date, order date, and total due information compared to the partially denormalized SalesOrderHeader table

Allowing a table to include null values in a column adds to the complexity of the business logic (and code) that manipulates or queries the data in that table. This is because you may need to check for null values explicitly every time you query or modify data in this table. Failure to handle null values correctly can result in subtle bugs in your application code. Eliminating null values can help to simplify your business logic, and reduce the chances of unexpected errors.

You can find more information on the benefits of normalizing a SQL Server database on the "Normalization" page in the SQL Server 2008 R2 documentation on the MSDN web site. This page describes normalization in SQL Server 2008 R2, but the same principles apply to SQL Server 2012 and Windows Azure SQL Database.

You can create views as an abstraction of complex, multi-table entities to simplify the logical queries that applications perform. An application can query a view in the same way as a table. However, the RDBMS still needs to perform the various join operations to construct the data for a view when it is queried.

Implementing small discrete tables can enable fast updates of individual data items, but the resultant schema naturally consists of a large number of narrow tables, and the data can easily become fragmented. The information that comprises a single logical entity, such as an order, is split up into multiple tables. Reconstructing the data for an entity requires joining the data in these tables back together, and this process requires processing power. Additionally, the data might be spread unevenly across the physical disks that provide the data storage, so joining data from several tables can incur a noticeable I/O overhead while disk heads have to seek from one location to another. As the number of requests increases, contention can cause the performance of the database to suffer, leading to a lack of scalability. Many modern RDBMSs attempt to counter these overheads by implementing in-memory caching to eliminate as much I/O as possible, and partitioning of data to reduce contention and maximize distribution.

Partial denormalization of tables (combining them together) can help to alleviate some of these concerns, and you should seek to balance the requirement for performing fast data updates against the need to retrieve data efficiently in your applications. Data that is modified rarely but read often will benefit from being denormalized, while data that is subject to frequent changes may be better left normalized.

Implementing Efficient Transactions

An OLTP scenario is characterized by a large number of concurrent operations that create, update, and delete data, packaged up as transactions. Most modern RDBMSs implement locking and logging strategies to ensure that the ACID (Atomicity, Consistency, Isolation, and Durability) properties of transactions are maintained. These features aim to guarantee the integrity of the data, but they necessarily have an impact on the performance of your transactions, and you should try and minimize their negative effects wherever possible. The following list provides some suggestions:

- **Keep transactions short.** A long-running transaction can lock data for an extended period of time, increasing the chances that it will block operations being performed by other concurrent transactions. Therefore, to maximize throughput, it is important to design the business logic carefully, and only perform the operations that are absolutely necessary within the bounds of a transaction.

  Many RDBMSs support the notion of isolation levels, which can affect the amount of blocking that a transaction experiences at the expense of consistency. For example, SQL Server implements the ***READ UNCOMITTED*** *isolation level that enables transaction A to read the uncommitted changes made by transaction B, bypassing any locks that might be held over the changed data. If the change is subsequently rolled back by transaction B, the data read by transaction A will be inconsistent with the values in the database. The most restrictive isolation level is* ***SERIALIZABLE****, which blocks if transaction A attempts to read data that has been changed by transaction B, but also prevents transaction B from subsequently changing any data that has been read by transaction A. This isolation level guarantees that if an application reads the same data item more than once during a transaction, it will see the same value each time. SQL Server also supports other isolation levels that fall between these two extremes.*

- **Avoid repeating the same work**. Poorly designed transactions can lead to deadlock, resulting in operations being undone. Your applications have to detect this situation and may need to repeat the transaction, reducing the performance of the system still further. Design your transactions to minimize this possibility. For example, always access tables and other resources in the same sequence in all operations to avoid the "deadly embrace" form of deadlock.
- **Avoid implementing database triggers over data that is updated frequently**. Many RDBMSs support triggers that run automatically when specified data is inserted, updated, or deleted. These triggers run as part of the same transaction that fired them, and they add complexity to the transaction. The developer writing the application code to implement the transaction might not be aware that the triggers exist, and might attempt to duplicate their work, possibly resulting in deadlock.
- **Do not include interactivity or other actions that might take an indeterminate period of time**. If your transactions depend upon input from a user, or data retrieved from a remote source, then gather this data before initiating the transaction. Users may take a long time to provide data, and information received from a remote source may take a long time to arrive (especially if the remote data source is some distant site being accessed across the Internet), give rise to the same consequences as a long-running transaction.
- **Implement transactions locally in the database**. Many RDBMSs support the concept of stored procedures or other code that is controlled and run by the database management system. You can define the operations that implement a transaction by using a stored procedure, and then simply invoke this stored procedure from your application code. Most RDBMSs are more easily able to refactor and optimize the operations in a stored procedure than they are the individual statements for a transaction implemented by application code that runs outside of the database.

  This approach reduces the dependency that the application has on a particular database schema but it might introduce a dependency on the database technology. If you switch to a different type of database, you might need to completely reimplement this aspect of your system.

Triggers are useful in query-intensive databases containing denormalized tables. In a denormalized database that contains duplicated data, you can use triggers to ensure that each copy of the data is updated when one instance changes.

For more information about triggers in SQL Server 2012, see the "DML Triggers" topic on MSDN.

For information about the benefits of using stored procedures in SQL Server 2012, see the "Stored Procedures (Database Engine)" topic on MSDN.

Implementing Distributed Transactions

A transaction can involve data held in multiple databases that may be dispersed geographically.

The architecture of an application may initially appear to require a single database held in a central location. However, the global nature of many modern customer-facing applications can often necessitate splitting a single logical database out into many distributed pieces, to facilitate scalability and availability, and reduce latency of data access. For example, you might find it beneficial to partition a database to ensure that the majority of the data that a customer accesses is stored geographically close to that customer. The section "Implementing a Relational Database to Maximize Concurrency, Scalability, and Availability" later in this chapter provides more information.

Solutions based on distributed synchronous transactions, such as those that implement ACID semantics, tend not to scale well. This is because the network latency of such a system can lead to long-running transactions that lock resources for an extended period, or that fail frequently, especially if network connectivity is unreliable. In these situations, you can follow a BASE (Basic Availability, Soft-state, Eventual consistency) approach.

For more information about implementing BASE transactions by using Windows Azure Service Bus Topics and Subscriptions, see Appendix A, "Replicating, Distributing, and Synchronizing Data" in the guide "Building Hybrid Applications in the Cloud on Windows Azure," available on MSDN.

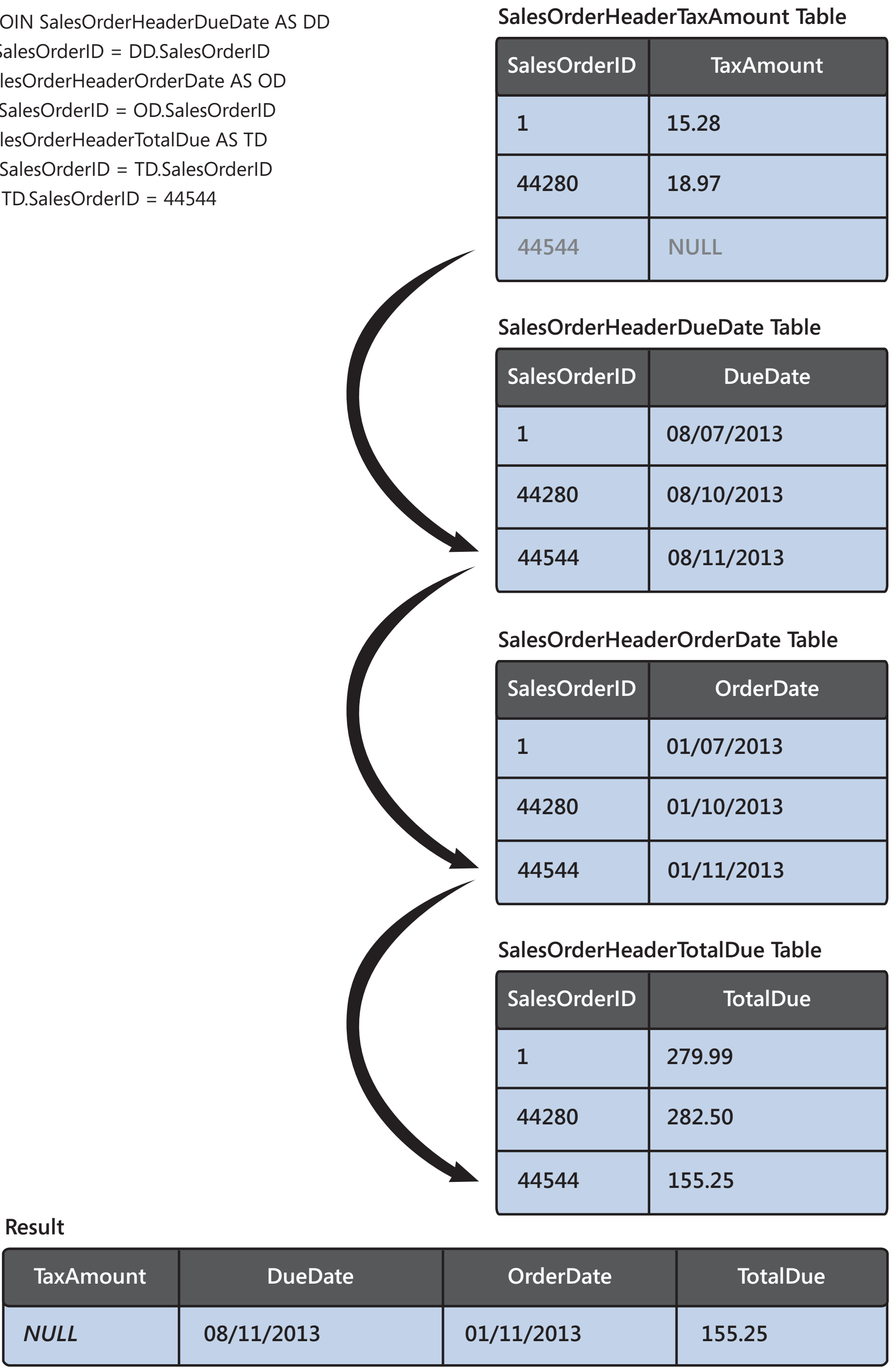
In a BASE transaction, information is updated separately at each site, and the data is propagated to each participating database. While the updates are in-flight, the state of overall system is inconsistent, but when all updates are complete then consistency is achieved again. The important point is to ensure that the appropriate changes are eventually made at all sites that participate in the transaction, and this necessitates that the information about these changes is not lost. This strategy requires additional infrastructure. If you are using Windows Azure, you can implement BASE transactions by using Windows Azure Service Bus Topics and Subscriptions. Application code can post information about a transaction to a Service Bus Topic, and subscribers at each site can retrieve this information and make the necessary updates locally.

Designing a Relational Database to Optimize Queries

One of the prime strengths of the relational model is the ability to retrieve and format the data to generate an almost infinite variety of reports. A normalized database as described in the previous section can be highly efficient for implementing OLTP, but the fragmentation of data into multiple tables can make queries slower due to the additional processing required to retrieve and join the data. In general, the more normalized the tables in a database are, the more tables you need to join together to reconstruct complete logical entities. For example, if you normalize the order data in the Adventure Works database as described in the previous section, then an application that reports or displays the details of orders must perform queries that reconstruct each order from these tables. Figure 2 shows an example of just such a query that retrieves the data for order 44544. As an added complication, there is no row in the **SalesOrderHeader-TaxAmount** table for this order, so it is necessary to phrase the query by using a right outer join which generates a temporary ghost row:

BASE transactions are most appropriate in situations where strict consistency is not required immediately, as long as data is not lost and the system eventually becomes consistent. Common scenarios include inventory and reservation systems. Many online banking systems also implement BASE transactions. If you pay for an item in a supermarket and then view your account online, the details of the transaction are only likely to appear some time later although the money will have left your account immediately.

```
SELECT TA.TaxAmount, DD.DueDate, OD.OrderDate, TD.TotalDue
FROM SalesOrderHeaderTaxAmount AS TA
RIGHT JOIN SalesOrderHeaderDueDate AS DD
ON TA.SalesOrderID = DD.SalesOrderID
JOIN SalesOrderHeaderOrderDate AS OD
ON DD.SalesOrderID = OD.SalesOrderID
JOIN SalesOrderHeaderTotalDue AS TD
ON OD.SalesOrderID = TD.SalesOrderID
WHERE TD.SalesOrderID = 44544
```



SalesOrderHeaderTaxAmount Table

SalesOrderID	TaxAmount
1	15.28
44280	18.97
44544	NULL

SalesOrderHeaderDueDate Table

SalesOrderID	DueDate
1	08/07/2013
44280	08/10/2013
44544	08/11/2013

SalesOrderHeaderOrderDate Table

SalesOrderID	OrderDate
1	01/07/2013
44280	01/10/2013
44544	01/11/2013

SalesOrderHeaderTotalDue Table

SalesOrderID	TotalDue
1	279.99
44280	282.50
44544	155.25

Result

TaxAmount	DueDate	OrderDate	TotalDue
NULL	08/11/2013	01/11/2013	155.25

Figure 2
Reconstructing the details of an order from the normalized tables holding order information

This section describes some techniques that you can follow to optimize your databases for query access.

Denormalizing Tables

If the database contains the data for a large number of entities and the application has to scale to support a large number of concurrent users, then it is important for you to reduce the overhead of performing queries.

The natural solution is to denormalize the database. However you must be prepared to balance the query requirements of your solution against the performance of the transactions that maintain the data in the database. Most organizations design the majority of their tables to follow 3NF. This level of normalization ensures that the database does not contain any duplicate or redundant data, while at the same time storing the data for most entities as a single row.

In a decision support system or data warehouse, you can go further and denormalize down to Second Normal Form (2NF), First Normal Form (1NF), or even denormalize the data completely. However, tables that follow these structures frequently contain large amounts of duplicated data, and can impose structural limitations on the data that you can store. For example, in the Adventure Works database, order information is divided into two tables: **SalesOrderHeader** that contains information such as the shipping details and payment details for customer that placed the order, and **SalesOrderDetail** that contains information about each line item in the order. Any query that has to find the details of an order joins these two tables together across the **SalesOrderID** column in both tables (this column is the primary key in the **SalesOrderHeader** table, and a foreign key in the **SalesOrderDetail** table). Figure 3 illustrates the structure of these two tables (not all columns are shown):

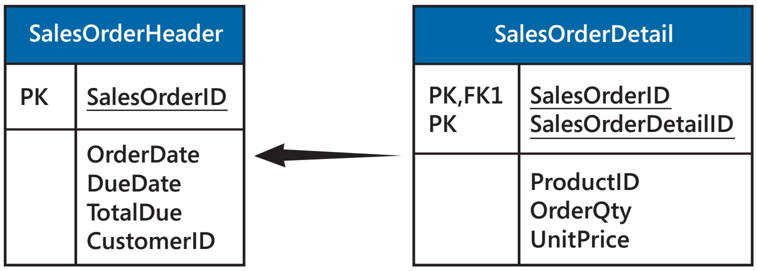
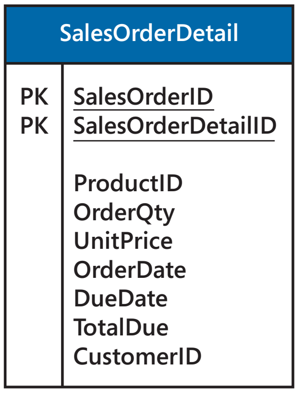
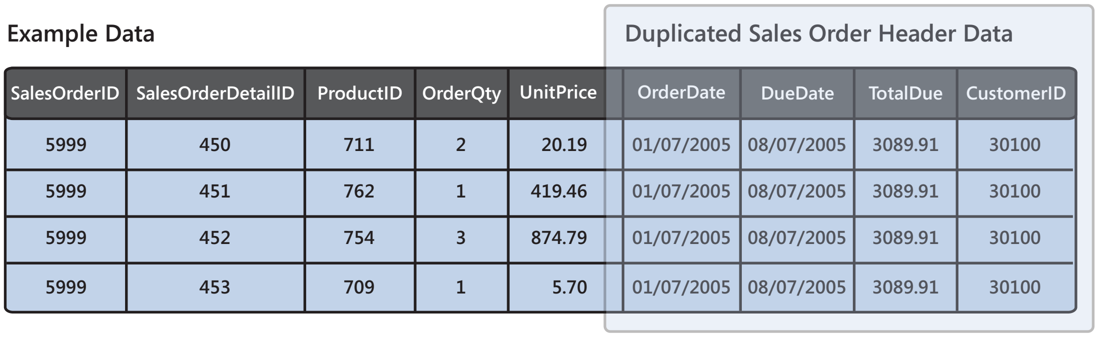


FIGURE 3
The structure of the SalesOrderHeader and SalesOrderDetail tables

There are at least two ways that enable you to eliminate this join:

- Add the columns from the **SalesOrderHeader** table to the **SalesOrderDetail** table, and then remove the **SalesOrderHeader** table. This design results in a table in 1NF, with duplicated sales order header data, as shown highlighted in Figure 4.





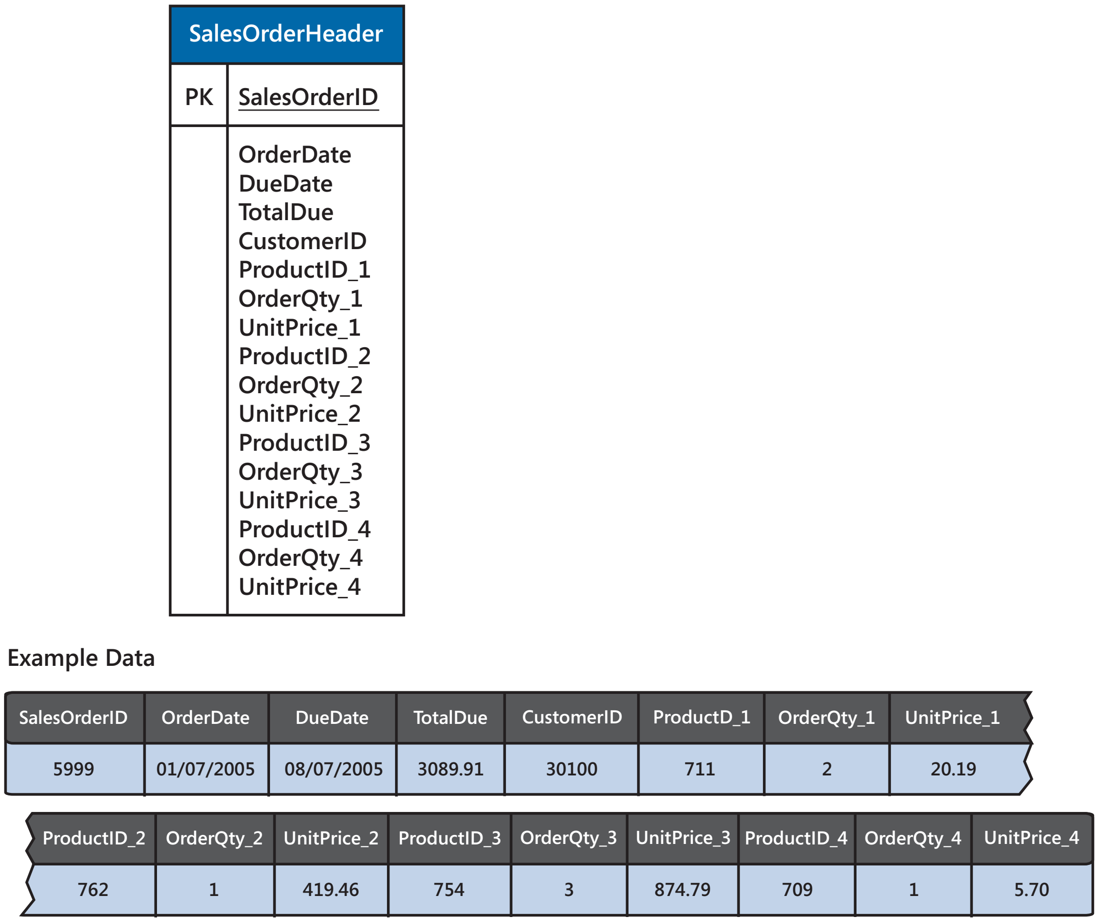
Example Data

Duplicated Sales Order Header Data

SalesOrderID	SalesOrderDetailID	ProductID	OrderQty	UnitPrice	OrderDate	DueDate	TotalDue	CustomerID
5999	450	711	2	20.19	01/07/2005	08/07/2005	3089.91	30100
5999	451	762	1	419.46	01/07/2005	08/07/2005	3089.91	30100
5999	452	754	3	874.79	01/07/2005	08/07/2005	3089.91	30100
5999	453	709	1	5.70	01/07/2005	08/07/2005	3089.91	30100

Figure 4
SalesOrder data structured as a table in 1NF

- Add the columns from the **SalesOrderDetail** table to the **SalesOrderHeader** table, and repeat these columns for each line item in an order. The result of this approach is a fully denormalized table. It does not contain any duplicated data, but you have to decide in advance the maximum number of line items that an order can contain. This approach can be too inflexible if the volume of line items can vary significantly between orders. Figure 5 shows the order data as a denormalized table.



SalesOrderID	OrderDate	DueDate	TotalDue	CustomerID	ProductD_1	OrderQty_1	UnitPrice_1
5999	01/07/2005	08/07/2005	3089.91	30100	711	2	20.19

ProductID_2	OrderQty_2	UnitPrice_2	ProductID_3	OrderQty_3	UnitPrice_3	ProductID_4	OrderQty_4	UnitPrice_4
762	1	419.46	754	3	874.79	709	1	5.70

FIGURE 5
SalesOrder data structured as a fully denormalized table

Additionally, a table such as this can make queries very complicated; to find all orders for product 711 you would have to examine the **ProductID_1**, **ProductID_2**, **ProductID_3**, and **ProductID_4** columns in each row.

Maintaining Query-Only Data

If you denormalize tables, you must ensure that the data that is duplicated between rows remains consistent. In the orders example, this means that each row for the same order must contain the same order header information. When an application creates a new order it can copy the duplicated data to the appropriate columns in all the rows for that order.

The main challenge occurs if the application allows a user to modify the data for an order. For example, if the shipping address of the customer changes, then the application must update this address for every row that comprises the order. This additional processing has an impact on the performance of the operation, and the extra business logic required can lead to complexity in the application code, and the corresponding increased possibility of bugs. You could use triggers (if your RDBMS supports them) to help reduce the processing overhead and move the logic that maintains the duplicate information in the database away from your application code. For example, a trigger that is fired when the shipping address for an order row is modified could find all other rows for the same order and replicate the change. However, you should avoid placing too much processing in a trigger because it can have a detrimental effect on the performance of the RDBMS as a whole, especially if the same trigger is fired frequently as the result of interactions from a large number of concurrent users.

If your system must support efficient queries over a large data set, and also allow fast updates to that data while minimizing the processing impact on the RDBMS, you can maintain two versions of the database; one optimized for queries and the other optimized for OLTP. You can periodically transfer the modified data from the OLTP database, transform it into the structure required by the query database, and then update the query database with this data. This approach is useful for DSS solutions that do not require up to the minute information. You can arrange for a regular batch process to perform the updates in bulk during off-peak hours. Many RDBMS vendors supply tools that can assist with this process. For example, Microsoft SQL Server provides SQL Server Integration Services (SSIS) to perform tasks such as this.

You can find more information about SQL Server Information Services in the "SQL Server Integration Services" section in SQL Server 2012 Books online.

Designing Indexes to Support Efficient Queries

Searching through a table to find rows that match specific criteria can be a time-consuming and I/O intensive process, especially if the table contains many thousands (or millions) of rows. Most RDBMSs enable you to define indexes to help speed this process up, and provide more direct access to data without the necessity of performing a linear search.

Most RDBMSs implement unique indexes (indexes that disallow duplicate values) over the columns that comprise the primary key of a table. The rationale behind this strategy is that most queries that join tables together are based on primary key/foreign key relationships, so it is important to be able to quickly find a row by specifying its primary key value. If your application poses queries that search for data based on the values in other columns, you can create secondary indexes for these columns. You can also create composite indexes that span multiple columns, and in some cases the RDBMS may be able to resolve a query simply by using data held in an index rather than having to fetch it from the underlying table. For example, if your application frequently fetches the first name and last name of a customer from a customers table by querying the customer ID, consider defining a composite index that spans the customer ID, first name, and last name columns.

Using indexes can speed up queries, but that do have a cost. The RDBMS has to maintain them as data is added, modified, and deleted. If the indexed data is subject to frequent updates, then this maintenance can add a considerable overhead to the performance of these operations. The more indexes that you create, the greater this overhead. Furthermore, indexes require additional storage resources, such as disk space and cache memory. Creating a composite index over a very large table might easily add 30% (or more) to the space requirements for that table.

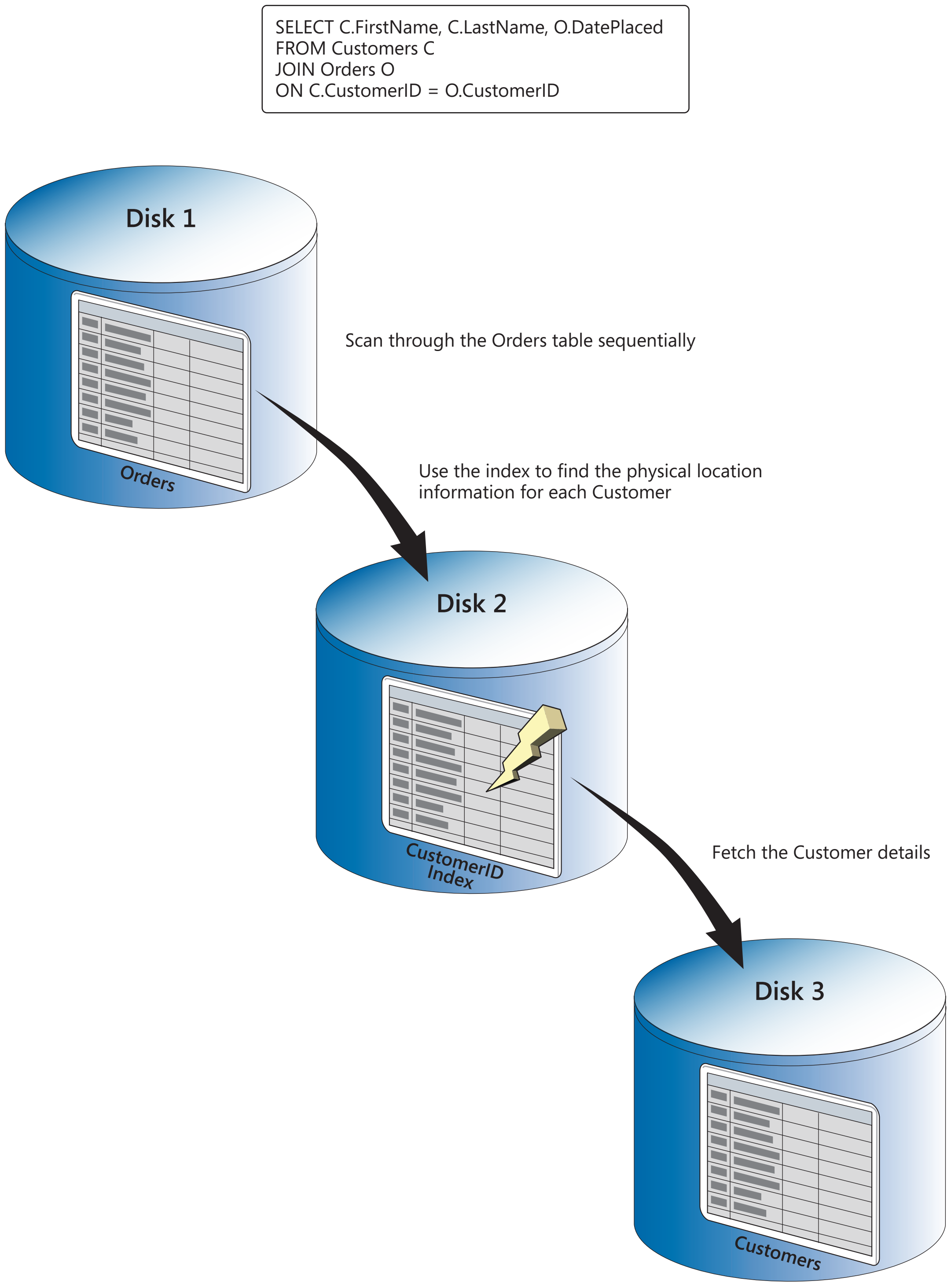
For more information about implementing indexes in a SQL Server database, read the "Indexes" section in Books Online for SQL Server 2012.

If you are maintaining two versions of a database (one for OLTP and one for DSS) as described in the section "Maintaining Query-Only Data" earlier in this chapter, then drop all of the indexes in the DSS database before you perform the bulk transfer of data from the OLTP database, and then rebuild them when the transfer is complete.

It is common to minimize the number of indexes in an OLTP-oriented database, but implement a comprehensive indexing strategy in a database used by DSS applications.

Partitioning Data

At the lowest level, a relational database is simply a file on disk. A common cause of poor performance when querying tables in a relational database is contention resulting from multiple I/O requests to the same physical disk. To counter this issue, many modern RDBMSs enable you to implement a relational database as a collection of files (each on a separate disk), and direct the data for different tables to a specific file or physical device. For example, if your application frequently runs queries that join information from customer and order tables, you could place the data for the customer table on one disk, and the data for the order table on another. You could also arrange for specific indexes to be stored on disks separate from the tables that they reference. Figure 6 shows an example (the tables are fictitious and are not part of the Adventure Works database). The query joins the **Customers** and **Orders** table over the **CustomerID** column to find the details of customers and the orders that they have placed. The **Customers** and **Orders** tables are placed on separate physical disks, and the **CustomerID** index (the primary key index for the **Customers** table) is placed on a third disk. This arrangement enables the RDBMS to take advantage of parallel I/O operations; the RDBMS can scan through the **Orders** table sequentially to find the **CustomerID** for each order, and use the **CustomerID** index to lookup the location of the customer details.



```
SELECT C.FirstName, C.LastName, O.DatePlaced
FROM Customers C
JOIN Orders O
ON C.CustomerID = O.CustomerID
```

Disk 1
Orders
Scan through the Orders table sequentially
Use the index to find the physical location information for each Customer
Disk 2
CustomerID Index
Fetch the Customer details
Disk 3
Customers

Figure 6
Maximizing parallel I/O by placing tables and indexes on separate physical disks

You can combine this technique with partial normalization of a table to place commonly accessed data on one device, and less frequently accessed data on another. This approach, known as vertical partitioning, reduces the row size for a table referenced by the most common queries, and therefore enables the RDBMS to retrieve more rows from disk in each I/O operation. If the RDBMS implements caching, then it can store more rows in memory, optimizing the process still further.

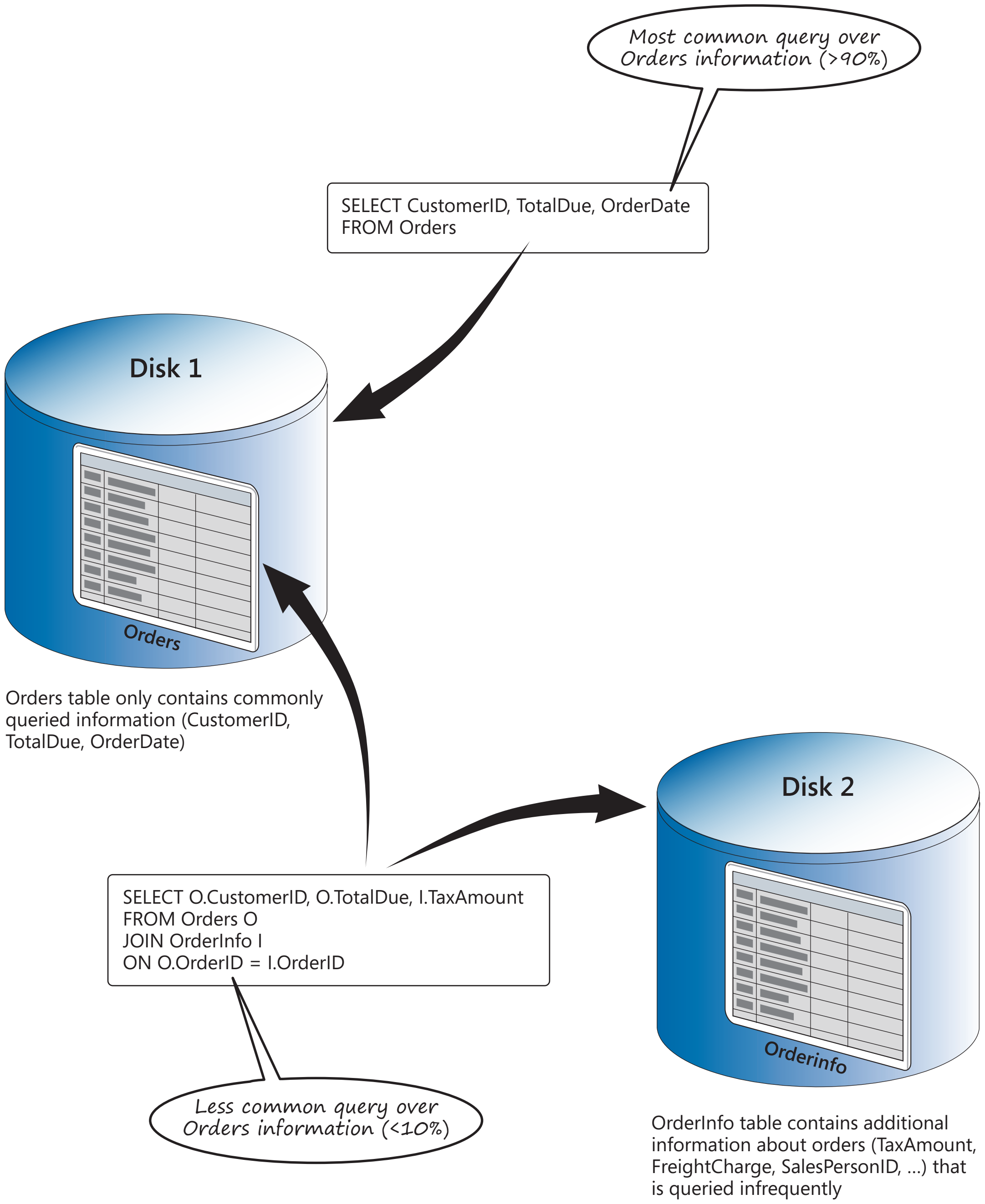
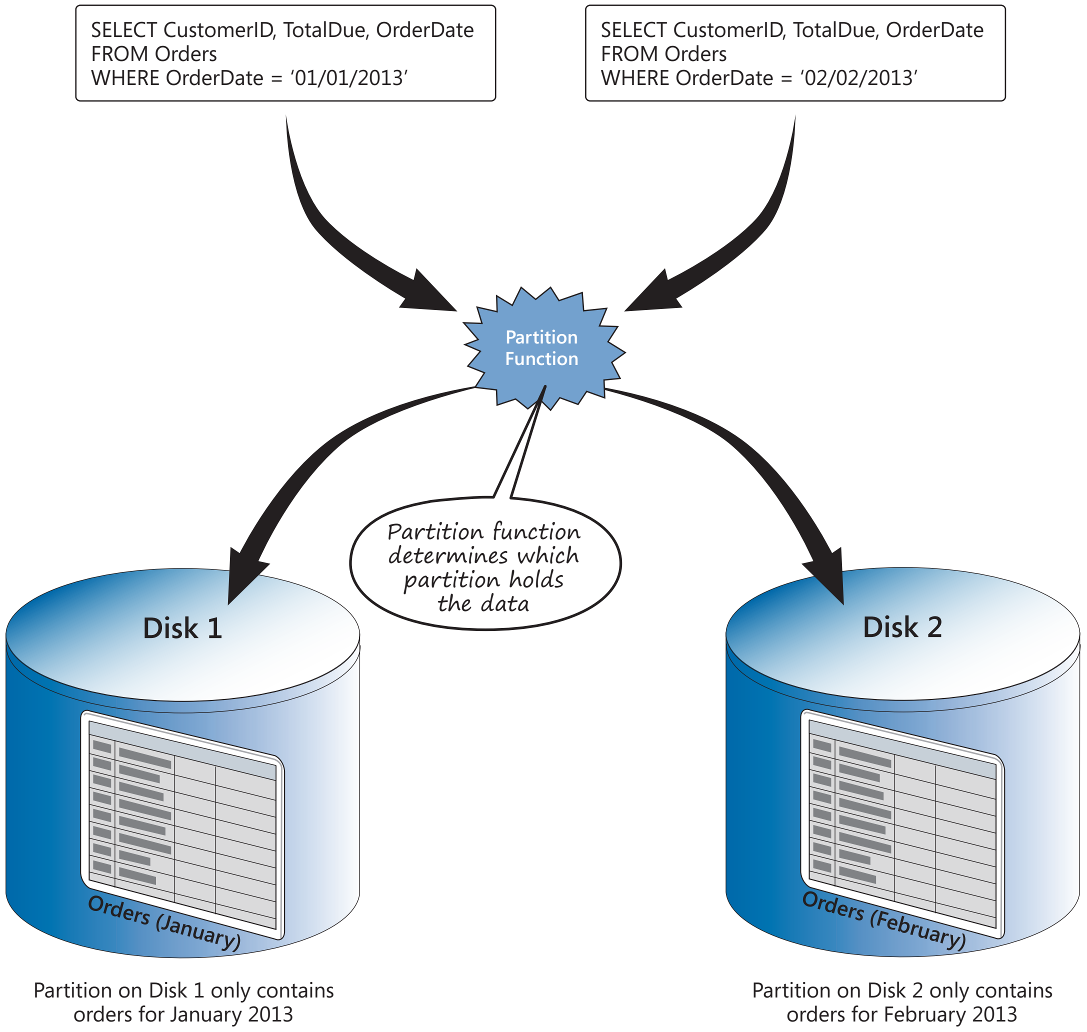

Most common query over Orders information (>90%)
SELECT CustomerID, TotalDue, OrderDate FROM Orders
Disk 1
Orders
Orders table only contains commonly queried information (CustomerID, TotalDue, OrderDate)
Disk 2
OrderInfo
SELECT O.CustomerID, O.TotalDue, I.TaxAmount FROM Orders O JOIN OrderInfo I ON O.OrderID = I.OrderID
Less common query over Orders information (<10%)
OrderInfo table contains additional information about orders (TaxAmount, FreightCharge, SalesPersonID, ...) that is queried infrequently

FIGURE 7
Partially normalizing a table and implementing vertical partitioning

Many RDBMSs also support horizontal partitioning. Horizontal partitioning enables you to divide the data for a single table up into sets of rows according to a partition function, and arrange for each set of rows to be stored in a separate file on a different disk.

You decide how to partition data based on how it is most frequently accessed. For example, you could partition customer order information by the month and year in which the order was placed; orders for December 2012 could be written to one partition, orders for January 2013 to another, orders for February 2013 to a third partition, and so on. In this way, queries that retrieve orders based on their date are quickly directed to a specific partition, and the data can be retrieved without having to scan through data or indexes relating to different months and years.



```
SELECT CustomerID, TotalDue, OrderDate
FROM Orders
WHERE OrderDate = '01/01/2013'
```

```
SELECT CustomerID, TotalDue, OrderDate
FROM Orders
WHERE OrderDate = '02/02/2013'
```

Partition Function
Partition function determines which partition holds the data
Disk 1
Orders (January)
Disk 2
Orders (February)
Partition on Disk 1 only contains orders for January 2013
Partition on Disk 2 only contains orders for February 2013

Figure 8
Implementing horizontal partitioning

As well as improving the performance of specific queries, horizontal partitioning can also speed up operations that transfer data from an OLTP system into a DSS database, and you can perform maintenance operations such as reindexing data more quickly. This is because these operations typically only target contiguous subsets of the data rather than the entire table, and data held in unaffected partitions does not have to be updated.

For more information about implementing horizontal partitioning with SQL Server, read the "Partitioned Tables and Indexes" section in Books Online for SQL Server 2012. For information on how SQL Server queries can take advantage of horizontal partitioning, see the "Query Processing Enhancements on Partitioned Tables and Indexes" page in SQL Server Books Online.

The Microsoft SQL Server Customer Advisory Team (SQLCAT) has published a list of techniques that you can adopt for building a large-scale database that supports efficient queries. This list is available on the "Top 10 Best Practices for Building a Large Scale Relational Data Warehouse" page on the SQLCAT website.

Horizontal partitioning is not so efficient for queries that retrieve data from multiple partitions. In the orders example, queries that need to fetch the set of orders for an entire year, or a period that spans a month boundary, may need to perform additional I/O to locate the various partitions. If the partitions are stored on different disks, the RDBMS may be able to offset this overhead by parallelizing the I/O operations.

How Adventure Works Designed the Database for the Shopping Application

The Shopping application supports a small number of business scenarios, as described in Chapter 2. The features that utilize the customer and order information are those concerned with registering a customer, logging in, and placing an order. The developers at Adventure Works examined the usage patterns for the various tables in the database for each of the scenarios, and they came to the following conclusions about how they should structure the data required to support these scenarios.

Registering a New Customer and Logging In

This functionality is arguably the most complex and sensitive part of the system. The developers had to design a schema that supported the following business requirements:

- Every customer must have an account that is identified by their email address and protected by using a password.
- All customers must pay for their orders by providing the details of a valid credit card.
- All customers must provide a billing address and a shipping address. These addresses can be the same, but they can also be different.

The dynamic nature of this data led the developers to implement the customer, credit card, and address information as a series of tables in 3NF. This structure helps to reduce the probability of duplicate information, while optimizing many of the common queries performed by the application. Figure 9 shows these tables and the relevant columns:

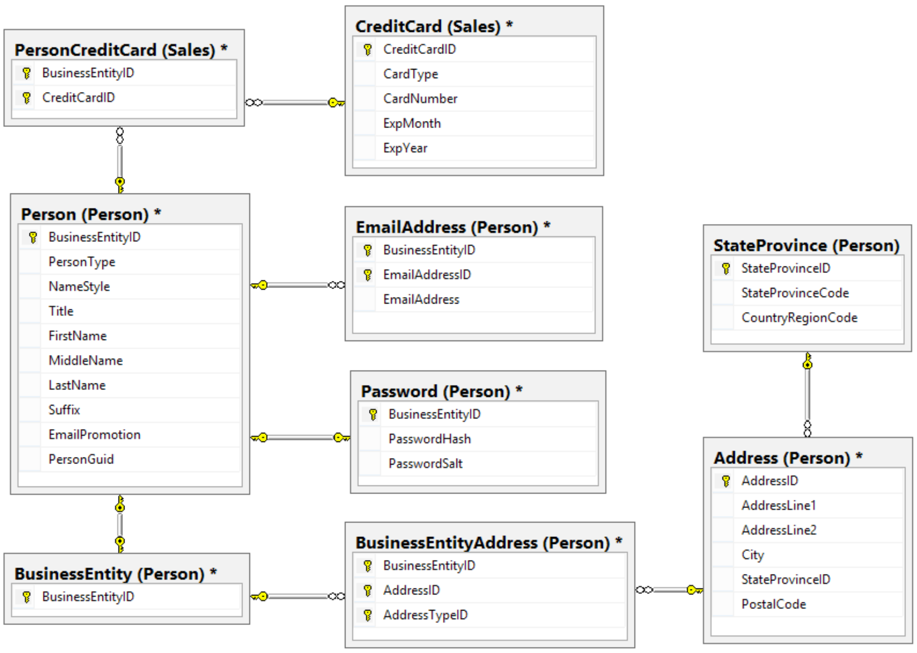


Figure 9
Tables used by the Shopping application to store customer information

The tables in the AdventureWorks2012 database contain additional columns, not shown in Figure 9, that are not relevant to the Shopping application.

All primary key columns in the Adventure Works database are implemented by using SQL Server clustered indexes. All foreign key columns have a secondary, non-clustered index to improve the performance of join operations.

Placing an Order

When the customer clicks **Checkout** on the shopping cart page in the Shopping application, the products that constitute the order are taken from the customer's shopping cart and used to create a new order. The customer can, in theory, place any number of items in their shopping cart.

The designers at Adventure Works considered storing the details of orders in a single denormalized table, but as described in Chapter 2, the warehousing and dispatch systems that arrange for goods to be picked and shipped to the customer modify the details held in an order (these systems are OLTP-intensive but are outside the scope of the Shopping application.) Therefore, the designers chose to implement the database schema for orders by using two tables:

- The **SalesOrderHeader** table holds information about the order (such as the identity of the customer, the billing address, the shipping address, the credit card that the customer used, and the value of the order).
- The **SalesOrderDetails** table holds the individual line items for each order.

Figure 10 shows the **SalesOrderHeader** and **SalesOrderDetail** tables:

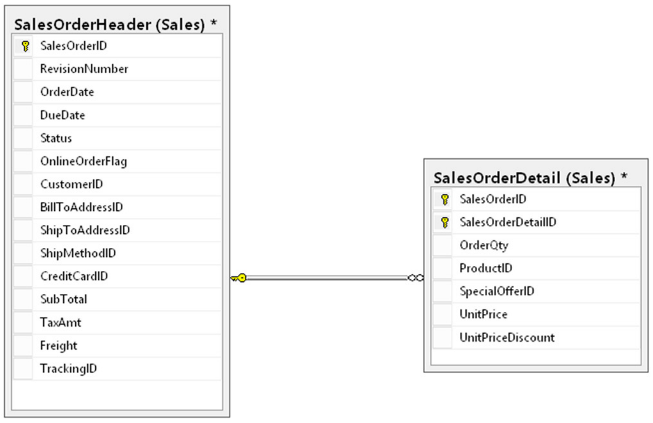


Figure 10
Tables used by the Shopping application to record the details of an order

This diagram only shows the columns in these tables that are relevant to the Shopping application. The warehousing and dispatch systems also use these tables and maintain additional information about orders (such as whether the items have been picked, the order has been dispatched, and so on) which are not included here.

Verifying the Price of an Item

The Shopping application also maintains a full audit trail of any changes made to orders after they have been placed. To support this functionality and to enable order history information to be retrieved quickly, the Shopping application copies the important details of an order to a separate document database when the order is placed, and each time the order is updated. The order history records for each order in the document database are identified by an order code rather than the ***SalesOrderID*** *used by the SQL Server database. This order code is a copy of the* ***TrackingID*** *field stored in the SQL Server database. See Chapter 5, "Implementing a Document Database," for more information about how order history information is stored and managed.*

Before an order can actually be placed and the details stored in the database, the Shopping application checks to see whether the price of any items have changed since the customer first placed them in their shopping cart (a customer's shopping cart is a long-lived entity that survives after the customer has logged out, and is restored when the customer logs back in again, possibly at a much later date).

Most of the details of each product are stored in the Product Catalog. This is a document database described in Chapter 5. However, the inventory management functionality of the warehousing system inside Adventure Works maintains product information in a separate SQL Server database, and a separate batch system (that is outside the scope of the Shopping application) periodically updates the product catalog with the latest inventory information. The product inventory information in the SQL Server database is stored in a single table name **Product**. The Shopping application checks the prices of items against this table rather than the product catalog. The **Product** table contains a number of columns that are required by the warehousing system, but the only information used by the Shopping application comprises the **ProductID** and the **ListPrice** columns shown in Figure 11.

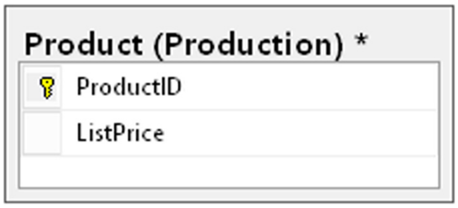


Figure 11
The columns in the Product table used by the Shopping application to verify the current price of an item

Maintaining Data Integrity

The Shopping application implements the following transactional operations.

- When a new customer registers with the Shopping application, the details of the customer and their credentials are added to the **Person**, **EmailAddress**, and **Password** tables. At the same time, their address is stored in the **Address** table, and this also requires creating the appropriate rows in the **Business-Entity** and **BusinessEntityAddress** tables to link the address back to the **Person** table. Finally, the credit card details are added to the **CreditCard** table which is linked back to **Person** by adding a new row to the **Person-CreditCard** table.
- When the customer places an order, the contents of the shopping cart are used to construct a sales order. This action requires that a new **SalesOrder-Header** row is created together with a **SalesOrderDetail** row for each item in the shopping cart, and then the rows in the **ShoppingCartItem** table must be deleted.

These operations are critical to the Shopping application. Failure to implement either of these as atomic processes could result in orders being lost or in incomplete details for a customer being added to the database, which in turn could cause problems when an order is charged or shipped. Because the designers knew that the structure of the database might change and that parts of the system such as the Shopping Cart and Order History will use a different type of database, they chose to implement these operations by writing code in the Shopping application rather than attempt to locate this functionality in the database in the form of stored procedures.

Chapter 8, "Building a Polyglot Solution" describes strategies for implementing atomic operations that span different types of databases, including NoSQL databases.

Implementing a Relational Database to Maximize Concurrency, Scalability, and Availability

As more and more users access a database, issues of scalability and availability become increasingly important. In a system intended to be used by a widely distributed audience, you also need to consider where data will be located in order to minimize network latency. To help ensure that a database remains available and responsive, many RDBMSs support redundancy and partitioning to spread the data processing load across multiple physical servers. You can also address scalability by using RDBMSs that run in the cloud and that provide the elasticity necessary to scale up and down as data processing requirements dictate. This section examines these issues, and describes some solutions that enable you to resolve them.

Scaling Out and Sharding

As the number of concurrent users and the volume of requests escalate, you need to ensure that the database scales to meet demand. You can either scale up the hardware, or scale out. Scaling up typically means purchasing, configuring, and maintaining a single unified hardware platform that provides enough resources to handle the peak workload. This approach can be expensive and often requires considerable administrative overhead if you need to upgrade the system to more powerful machinery. For these reasons, a scale-out approach is usually preferred.

Scaling out spreads the logical database out across multiple physical databases, each located on a separate node. As the workload increases over time, you can add nodes. However, the structure of the database and the tables that it contains are more complex, and understanding how to partition the data is crucial for implementing this strategy successfully.

As described in Chapter 1, "Data Storage for Modern High-Performance Business Applications," the most common pattern for designing a database intended to scale out in this way is to use horizontal partitioning, or *sharding*, at the database level. Each partition can reside on a separate node and can contain information from several tables. As the size of the database and the number of requests increase, you can spread the load across additional nodes, and performance can improve in a near-linear manner as you add more nodes. This approach can also be more cost-effective than scaling up because each node can be based on readily-available commodity hardware.

Sharding requires that you divide your database up into logical subsets of data, and deploy each subset to a specific node. Each node is a database in its own right, managed by an RDBMS.

> *Sharding a database necessarily distributes data across multiple nodes. While this can improve scalability and can help optimize applications that query data, it can also have a detrimental effect on the performance of any transactions that update data spread across multiple shards. In these situations, you may find it better to trade consistency for performance and implement updates as BASE operations rather than distributed ACID transactions.*

You can implement sharding in many ways, but the following two strategies illustrate how to apply the most common patterns to a relational system:

- **The Shared Nothing Pattern**. In this pattern, each database runs as an autonomous unit, and the sharding logic that determines where to store and retrieve data is implemented by the application that users run. This model is called the *Shared Nothing* pattern because no data is shared or replicated between nodes. Figure 12 shows this model.

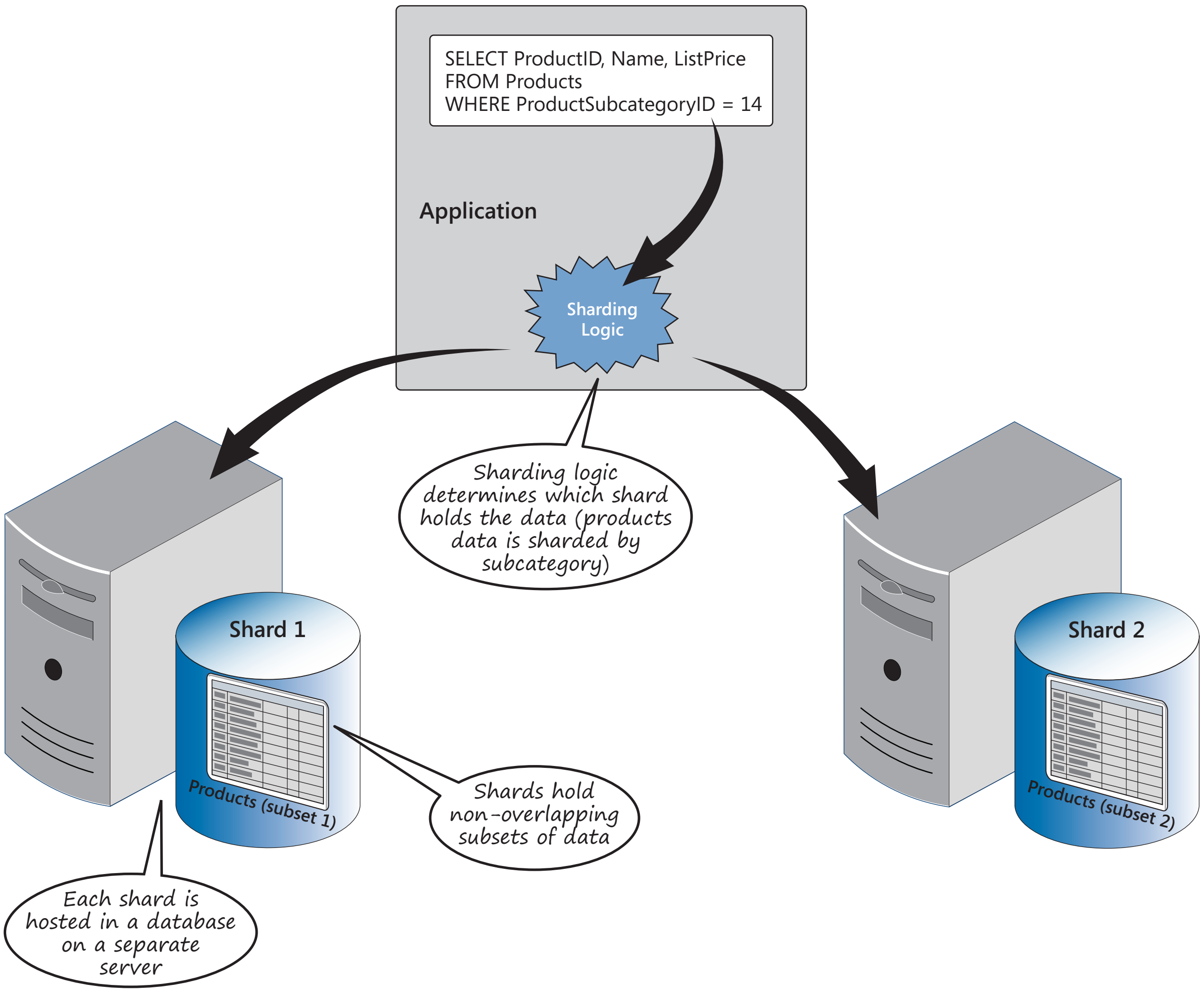

SELECT ProductID, Name, ListPrice
FROM Products
WHERE ProductSubcategoryID = 14
Application
Sharding Logic
Sharding logic determines which shard holds the data (products data is sharded by subcategory)
Shard 1
Products (subset 1)
Shard 2
Products (subset 2)
Shards hold non-overlapping subsets of data
Each shard is hosted in a database on a separate server

Figure 12
Scaling out by implementing the *Shared Nothing* approach

In this example, the sharding logic determines which shard contains the details of the specified product based on the subcategory (different shards hold data for different subcategories.) This pattern does not require any special support from the RDBMS, but the disadvantage is that shards can become unbalanced if one subcategory contains far more data than another. If this is likely, then you should implement sharding logic that more evenly distributes data, such as using a hash of the primary key. Redistributing data across existing unbalanced shards and modifying the sharding logic can be a complex, manual process that may require an administrator to take the application temporarily offline. This may not be acceptable if the system has to be available 24 hours a day.

Additionally, the sharding logic has to have enough information to know which shard to access for any given query. For example, if the data is sharded based on a hash of the primary key, but a query does not specify which key values to look for, then the sharding logic may need to interrogate every shard to find all matching data. This is not an efficient strategy.

- **The Federation Pattern**. In this pattern, the RDBMSs take on the responsibility for managing the location of data and balancing the shards. One database acts as the *federation root*, and stores the metadata that describes the location of the different shards and how data is partitioned across these shards, as shown in Figure 13.

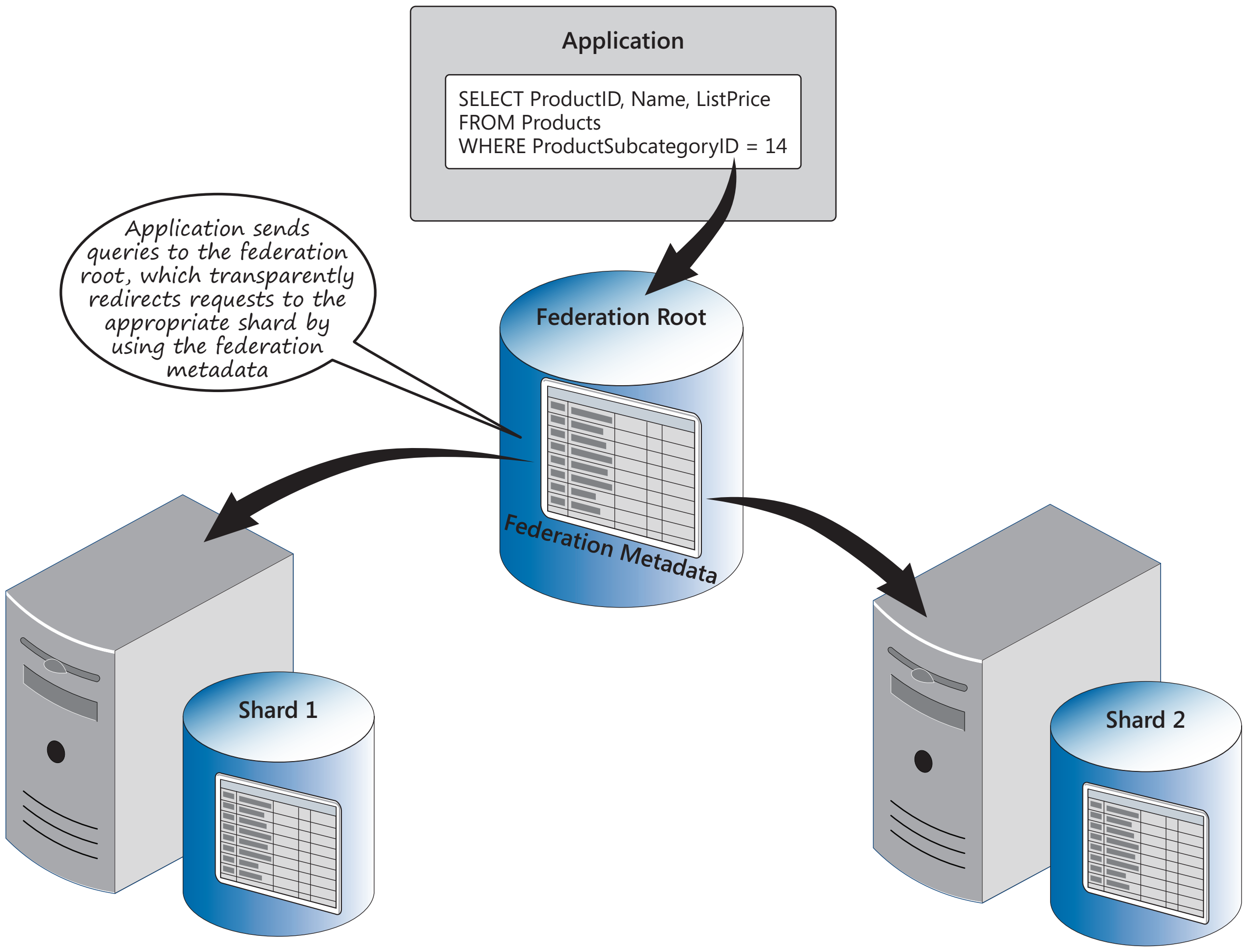

Application
SELECT ProductID, Name, ListPrice
FROM Products
WHERE ProductSubcategoryID = 14
Application sends queries to the federation root, which transparently redirects requests to the appropriate shard by using the federation metadata
Federation Root
Federation Metadata
Shard 1
Shard 2

Figure 13
Scaling out by implementing the Federation pattern

Applications connect to the federation root, but requests are transparently directed towards the database holding the appropriate shard. As the data changes, it can be relocated from one shard to another to rebalance the partitions.

This pattern requires support from the RDBMS itself to implement the federation root. An increasing number of RDBMS vendors are implementing this pattern, including Microsoft SQL Server and Windows Azure SQL Database.

Sharding can provide fast, direct access to data, but if you need to join this data with information held in tables located on other nodes then this advantage can disappear. You can combine sharding with replication (described in the section "Minimizing Network Latency" below) if you regularly need to perform queries that join data in a shard with relatively static information.

Minimizing Network Latency

If your database has a large number of users that are dispersed geographically, you can also use sharding to minimize the latency of data access. In many cases, the data that users require follows a pattern that mirrors the location of the users themselves. For example, in the Shopping application, customers located in the Western United States are more likely to query customer and order information for that same region (their own data and orders). Therefore, it would be beneficial to store that data in a shard that is physically located in the same region and reduce the distance that it has to travel. Data stored in other shards will still be available, but it may take longer to retrieve.

Using sharding to minimize latency requires that the application knows in which shard to find the data. If you are using an RDBMS that implements federation, the application will need to connect to the federation root to discover this information, and this will increase the latency of requests. In this case, it may be more beneficial to incorporate the sharding logic in the application and revert to the Shared Nothing pattern.

Not all data will fit this pattern. In the Shopping application, all customers are likely to query the same category, subcategory, and product information regardless of their location. One solution to reduce the latency of data access for this information is to replicate it in each region. As data changes, these changes will need to be propagated to each replica. This approach is most beneficial for relatively static data, because updates will be infrequent and the resulting overhead of synchronizing multiple copies of data is small. For data that does change relatively often, you can still maintain multiple replicas, but you should consider whether all users require up to the minute consistency of this data. If not, then you can implement synchronization as a periodic process that occurs every few hours.

If you are using Windows Azure SQL Database to store your data in the cloud, you can replicate tables and synchronize databases by using SQL Data Sync.

Appendix A, "Replicating, Distributing, and Synchronizing Data" in the guide "Building Hybrid Applications in the Cloud on Windows Azure," available on MSDN contains guidance on using Windows Azure SQL Database and SQL Data Synchronization to distribute and replicate data held in SQL Server databases.

Improving Availability

In many systems, the database is a critical part of the infrastructure, and the system may not function correctly (or at all) if the database is not available. A database might be unavailable for a number of reasons, but the most common causes include:

- **Failure of the server hosting the database**. Hardware failure is always a possibility, and if you need to ensure that your database is available at all times you may need to implement some form of hardware redundancy. Alternatively, you can arrange to maintain a copy of the data on another server, but this will require that the second server is kept up to date. If data changes frequently, this may require implementing a failover solution that duplicates the effects of all transactions as they occur.

SQL Server supports failover clusters that can provide high availability by maintaining multiple local instances of SQL Server that contain the same data. For more information, review the section "AlwaysOn Failover Cluster Instances (SQL Server)" in Books Online for SQL Server 2012.

If you have implemented a distributed database solution as described in the section "Minimizing Network Latency," a second possibility is to replicate the data in all shards as well as the common data required by all users. To retain the performance of operations that create, update, and delete data it may not be possible to implement up to the minute transactional integrity across all replicas, so you may need to implement additional infrastructure such as reliable message queues to ensure that updates are not lost and that all replicas become consistent eventually. Appendix A in the guide *"Building Hybrid Applications in the Cloud on Windows Azure"* shows how you can achieve this by using Windows Azure.

- **Loss of connectivity to the server hosting the database**. All networks, whether they are wired or wireless, are prone to failure. Even if you implement hardware redundancy, if the database server constitutes a single node then you have the potential for loss of connectivity. Replicating the database across multiple nodes can mitigate this possibility, but your application must be able to quickly switch to a node that has connectivity. This may necessitate incorporating additional logic into your application to detect whether a it can connect to the database, or you might need to reconfigure the application manually to reference a different node.
- **Overloading the server hosting the database, causing requests to timeout**. If a database server is heavily loaded, it may be slow in responding to requests, resulting in timeout failures, other errors, or just poor performance in your application while it waits for a response. As with loss of connectivity, a common solution is to replicate data and fall back to a responsive server.

If you are using Windows Azure to host services as part of your solution, you can use Windows Azure Traffic Manager to manage connectivity to these services and transparently redirect requests if an instance of a service should fail or a server become unresponsive. For more information, read Appendix E, "Maximizing Scalability, Availability, and Performance" in the guide "Building Hybrid Applications in the Cloud on Windows Azure."

Distributed databases exhibit behavior that can be summarized by the CAP theorem. This theorem states that it is impossible for a distributed system to guarantee data consistency (all nodes see exactly the same data at the same time), availability (all requests will receive a response), and partition tolerance (the system can continue to function even if connectivity to part of it is lost) at the same time. You can only ever meet two of these guarantees. Many designers choose to compromise on data consistency and implement BASE semantics; updates may take their time to propagate across all sites in the system, but data will eventually become consistent at every site.

Implementing a Relational Database in the Cloud by Using Microsoft Windows Azure

Cloud-based database servers are becoming increasingly popular because they remove the need for an organization to maintain its own infrastructure for hosting a database. They can prove extremely cost effective, especially if they offer elasticity that can enable a system to scale quickly and easily as the number of requests and volume of work increases. Windows Azure implements a range of services that can help you to build just such a solution, and you have at least two options that you can choose for hosting a database:

- **Use Windows Azure SQL Database**. This is a version of SQL Server designed specifically to run in the cloud. It offers compatibility with SQL Server running on-premises within an organization, and you can use many of the same tools to connect to it and manage data. Windows Azure SQL Database provides enterprise-class availability, scalability, and security, with the benefits of built-in data protection and self-healing.

- **Create a virtual machine to run your database server, and then deploy this virtual machine to the cloud**. This approach gives you complete flexibility and control over the database management system, and you can implement almost any RDBMS solution, whether it is SQL Server 2012, MySQL, or another technology.

If you need to run a database management system other than SQL Server, you can create a virtual machine that hosts this software and deploy it to Windows Azure.

Each of these strategies has its own advantages and disadvantages over the other. The following list summarizes some of the key points:

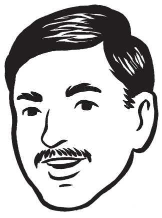
- **Pricing**. The cost of running a virtual machine depends on the number of resources it requires, and is charged on an hourly basis. Windows Azure SQL Database is priced according to the number of databases and the size of each database.
- **Scalability**. If you run SQL Server in a virtual machine, you can scale up to the maximum size allowed for a virtual machine; 8 virtual CPUs, 14GB of RAM, 16TB of disk storage, and 800MB/s bandwidth.

  If you are using Windows Azure SQL Database, the environment in which the database server runs is controlled by the datacenter. The host environment may be shared with other database servers, and the datacenter tries to balance resource usage so that no single application or database server dominates any resource, throttling a server if necessary to ensure fairness. The maximum size of a single database is 150GB. However, you can create multiple databases on a single server, and you can also create additional servers. Windows Azure SQL Database implements SQL Database Federation, providing a transparent sharding mechanism that enables you to scale out across multiple servers very easily.

  You can find detailed information describing federation with Windows Azure SQL Database at "Federations in Windows Azure SQL Database," on MSDN.

- **Availability**. SQL Server running in a virtual machine supports SQL Server AlwaysOn availability groups, read-only secondaries, scalable shared databases, peer-to-peer replication, distributed partitioned views, and data-dependent routing. Additionally, the virtual machine itself is guaranteed to be available 99.9% of the time, although this guarantee does not include the database management system running in the virtual machine.

  Windows Azure SQL Database comes with the same high availability guarantees (99.9% uptime) as a Windows Azure virtual machine. When you create a new database, Windows Azure SQL Database transparently implements multiple replicas across additional nodes. If the primary site fails, all requests are automatically directed to a secondary node with no application downtime. This feature comes at no extra charge.



Windows Azure SQL Database maintains multiple copies of a database running on different servers. If the primary server fails, then all requests are transparently switched to another server.

- **Compatibility**. If you are running SQL Server in a virtual machine, you have complete access to the tools and other features provided by the software, such as SQL Server Integration Services. Similarly, you can choose to run a completely different database management system in your virtual machine. The virtual machine itself can be running Windows, or Linux, and you have full control over any other software running in the virtual machine that integrates with your database management system.

  If you are using Windows Azure SQL Database, you have access to a large but specific subset of the functionality available in the complete SQL Server product. For example, Windows Azure SQL Database does not support tables without clustered indexes.

A complete list of the features available in Windows Azure SQL Database, is available at "General Guidelines and Limitations (Windows Azure SQL Database)," on MSDN.

- **Administration and configuration**. If you are using a virtual machine to host your database server, you have complete control over the database software, but you are responsible for installing, configuring, managing, and maintaining this software.

  If you are using Windows Azure SQL Database, the datacenter configures and manages the software on your behalf, but you have little or no control over any specific customizations that you might require.
- **Security and Connectivity**. You can easily integrate a Windows Azure virtual machine into your own corporate network by using Windows Azure Virtual Network. This feature provides a safe mechanism that can make the virtual machine an authenticated member of your corporate network and participate in a Windows domain. Services such as the database management system running on the virtual machine can authenticate requests by using Windows Authentication.

  The blog post "Data Series: SQL Server in Windows Azure Virtual Machine vs. SQL Database" contains more information comparing the features of SQL Server running in a virtual machine with Windows Azure SQL Database. The blog post "Choosing between SQL Server in Windows Azure VM & Windows Azure SQL Database," describes how you can select whether to use SQL Server running in a virtual machine or Windows Azure SQL Database to implement your database.

  Windows Azure SQL Database runs in a shared environment that does not integrate directly with your on-premises servers. Windows Authentication is not supported. Additionally, there are restrictions on the log-ins that you can use (you cannot log in as **sa**, **guest**, or **administrator**, for example).

Why Adventure Works Used Windows Azure SQL Database for the Shopping Application

The designers at Adventure Works deployed the database in the cloud by using Windows Azure SQL Database for the following reasons:

- The database had to be accessible to the Shopping application, which is hosted using Windows Azure. For security reasons, the IT services team at Adventure Works was unwilling to open up external access to SQL Server running on Adventure Works own in-house infrastructure.
- Windows Azure SQL Database provides the scalability, elasticity, and connectivity required to support an unpredictable volume of requests from anywhere in the world.
- Microsoft guarantees that a database implemented by using Windows Azure SQL Database will be available 99.9% of the time. This is important because if the database is unavailable then customers cannot place orders.

The developers at Adventure Works also considered using a series of dedicated virtual machines running SQL Server in the cloud. However, they did not require the infrastructure isolation or level of configuration control available by following this approach. In fact, they felt that running SQL Server in a virtual machine might place too much of a burden on the in-house administrators who would be responsible for monitoring and maintaining the health of a collection of RDBMSs.

To reduce latency for customers running the Shopping application, Adventure Works deployed an instance of the application to multiple Windows Azure datacenters, and configured Windows Azure Traffic Manager to route users to the nearest instance of the Shopping application. Each datacenter also hosts an instance of the database, and the application simply uses the local instance of the database to query customer details and store information about orders. The designers configured Windows Azure SQL Data Sync to synchronize data updates on a daily basis so information about all orders eventually propagates across all sites. One site was selected to act as the synchronization hub, and the others are members that synchronize with this hub. In the event of failure or loss of connectivity to the hub, it is possible to reconfigure Windows Azure SQL Data Sync and select a different synchronization hub, or even force a manual synchronization, without modifying any application code.

The warehousing and dispatch systems inside Adventure Works use its own SQL Server database running on-premises within the datacenter of the organization. This database was included as a member database in a separate synchronization group that synchronizes with the hub every hour, and all orders placed by customers are transmitted to this database by the synchronization process.

Figure 14 shows the high-level structure of this solution.

The sample application provided with this guide does not implement replication by default, but you can configure this option manually. For more information about Windows Azure SQL Data Sync, see the "SQL Data Sync" topic on MSDN.

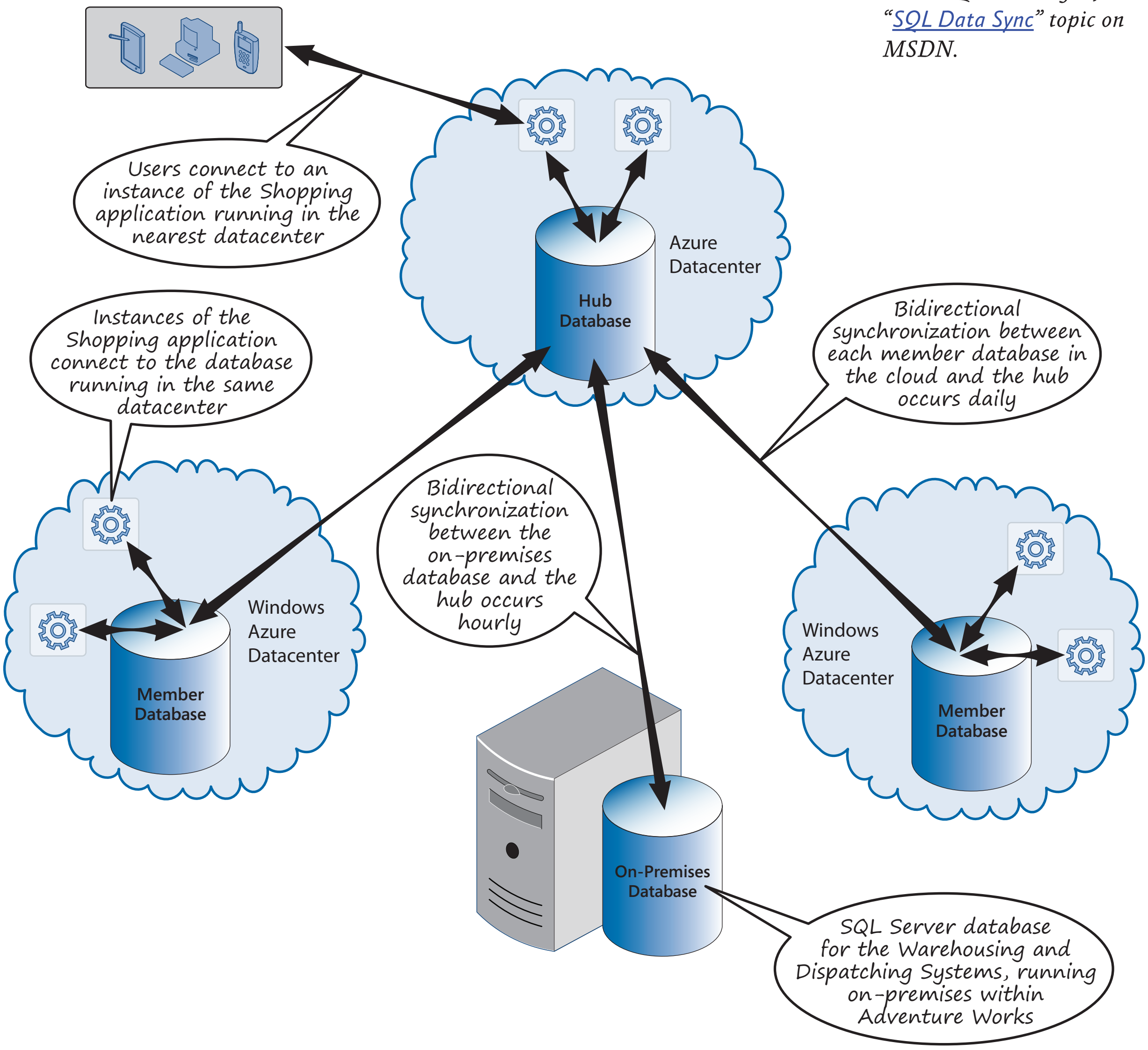

Users connect to an instance of the Shopping application running in the nearest datacenter
Azure Datacenter
Hub Database
Instances of the Shopping application connect to the database running in the same datacenter
Bidirectional synchronization between each member database in the cloud and the hub occurs daily
Bidirectional synchronization between the on-premises database and the hub occurs hourly
Windows Azure Datacenter
Member Database
Windows Azure Datacenter
Member Database
On-Premises Database
SQL Server database for the Warehousing and Dispatching Systems, running on-premises within Adventure Works

Figure 14
How Adventure Works deployed the databases and synchronize the data

Accessing Data in a Relational Database from an Application

You need to give careful attention to the code that you write to access the data in a relational database. This code can have an impact on the performance, not just of a single instance of the application, but on the system as a whole, especially if you implement transactional exchanges with the database in a suboptimal manner. Additionally, you should avoid tying the structure of your code too closely to the database technology otherwise you may be faced with making major changes to your application should you need to switch to a different type of data store.

Connecting to a Relational Database

An application typically interacts with a relational database by using SQL commands. However, SQL is a language rather than an API or network protocol, so RDBMS vendors provide tools and libraries that enable an application to create SQL commands and package them up in a vendor-specific format that can be transported from the application to the database server. The database server processes the commands, performs the appropriate work, and then sends results back by using another vendor-specific format. The application uses the vendor-provided library to unpack the response and implement the necessary logic to process this response.

The tools and libraries available to an application are dependent on the RDBMS that the system is using. For example, Microsoft provides ActiveX Data Objects (ADO.NET), which enable an application built using the .NET Framework to connect directly to Microsoft SQL Server. Other vendors supply their own libraries. However, ADO.NET implements a layered model that decouples the API exposed to applications from the underlying format of the data that is transported to the database server. This design has enabled Microsoft and other third-party vendors to develop database drivers that enable applications to connect to different RDBMSs through ADO.NET.

The original ActiveX Data Objects library predates the .NET Framework. This library is still available if you are building applications that do not use the .NET Framework.

There have been attempts to develop a standard library to enable applications to have a degree of independence from the RDBMS, Open Database Connectivity (ODBC) which was first proposed by the SQL Access Group in 1992 being a prime example. ADO.NET supports ODBC, enabling you to build applications that can connect to any RDBMS that has an ODBC driver.

Despite its age, ODBC is an important data integration technology. You can use ODBC to connect to an RDBMS from Microsoft Word if you need to include information from a database in a document, for example. Additionally, ODBC drivers are available for several data sources that are not RDBMSs. For example, Microsoft supplies an ODBC driver for Excel, which enables you to retrieve data from an Excel spreadsheet by performing SQL queries.

You can also use the patterns & practices Data Access Block (part of Enterprise Library) to perform common database operations by writing database-neutral code. The purpose of the Data Access Block is to abstract the details of the RDBMS that you are using from the code that interacts with the database. The Data Access Block enables you to switch to a different RDBMS and minimize the impact that such a change might have on your code. You need to provide configuration information that specifies which RDBMS to connect to, and parameters that contain additional connection information, but other than that if you are careful you can build applications that have minimal dependencies on the database technologies that they use.

You can find more information about the latest version of Enterprise Library on MSDN.

Abstracting the Database Structure from Application Code

A relational database stores data as a collection of tables. However, a typical application processes data in the form of entity objects. The data for an entity object might be constructed from one or more rows in one or more tables in the database. In an application, the business logic that displays or manipulates an object should be independent of the format of the data for that object in the database so that you can modify and optimize the structure of the database without affecting the code in the application, and vice versa.

Using an Object-Relational Mapping Layer

Libraries such as ADO.NET, ODBC, and the Data Access Block typically provide access to data in a structure that mirrors its tabular form in the database. To decouple this structure from the object model required by an application, you can implement an object-relational mapping layer, or ORM. The purpose of an ORM is to act as an abstraction of the underlying database. The application creates and uses objects, and the ORM exposes methods that can take these objects and use them to generate relational CRUD (create, retrieve, update, and delete) operations, which it then sends to the database server by using a suitable database driver. Tabular data returned by the database is converted into a set of objects by the ORM. If the structure of the database changes, you modify the mappings implemented by the ORM, but leave the business logic in the application unchanged.

ORMs follow the **Data Mapper** pattern to move data between objects and the database, but keep them independent of each other. Data Mapper is a actually a meta-pattern that comprises a number of other lower-level patterns that ORMs implement, typically including:

- **Metadata Mapping**. An ORM uses this pattern to define the mappings between in-memory objects and fields in database tables. The ORM can then use these mappings to generate code that inserts, updates, and deletes information in the database based on modifications that an application makes to the in-memory objects.
- **Interpreter**. The Interpreter pattern is used to convert application-specific code that retrieves data into the appropriate SQL **SELECT** statements that the RDBMS understands. ORMs based on the .NET Framework frequently use this pattern to convert LINQ queries over entity collections into the corresponding SQL requests.

- **Unit of Work**. An ORM typically uses this pattern to maintain a list of changes (insert, update, and delete operations) over in-memory objects, and then batch these operations up into transactions comprising one or more SQL operations (generated by using the metadata mapping) to save the changes. If the transaction fails, the reason for the failure is captured and the state of the objects in-memory is preserved. The application can use this information to rectify the failure and attempt to save the changes again.

Microsoft provides the Entity Framework as an ORM for .NET Framework applications. Other popular ORMs available for the Microsoft platform include NHibernate and nHydrate.

Using the Entity Framework

The Entity Framework is integrated into Visual Studio, and it enables you to build applications that can interact with RDBMSs from a variety of vendors, including Oracle, IBM, Sybase, and MySQL. You can work with the Entity Framework by following a database-first or code-first approach, depending on whether you have an existing database that you wish to use, or you want to generate a database schema from an existing set of objects, as follows:

- If you have an existing database, the Entity Framework can generate an object model that directly mirrors the structure of the database. Rows in tables are mapped to collections of objects, and relationships between tables are implemented as properties and validation logic. You can also create mappings to data returned by stored procedures and views. Visual Studio 2012 provides the ADO.NET Entity Data Model template to help you perform this task. This template runs a wizard that enables you to connect to the database, and select the tables, views, and stored procedures that your application will use. The wizard constructs an entity model and can generate a collection of entity classes that correspond to each of the tables and views that you selected. You can use the Entity Model Designer to amend this model, add or remove tables, modify the relationships between tables, and update the generated code. Figure 15 shows an example:

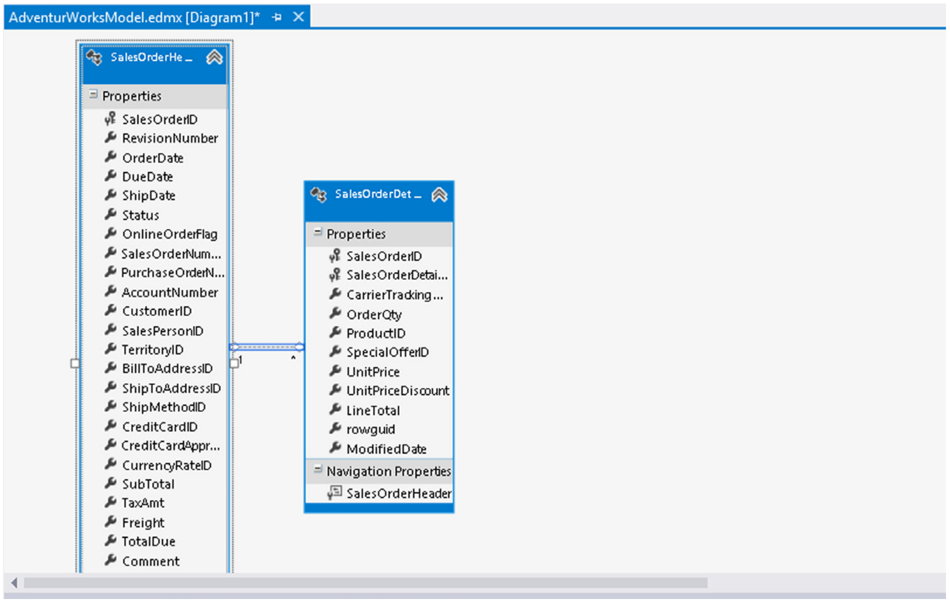


FIGURE 15
The Entity Model Designer in Visual Studio 2012

You can also reverse engineer a database from your code. You can generate a SQL script that will create the tables and relationships that correspond to the entities and mappings in the entity model.

> *In the Entity Framework version 5.0, the Entity Data Model Wizard does not generate the entity classes by default. You must set the **Code Generation Property** of the entity model to **Default** by using the Entity Model Designer.*

- If you have an existing object model, the Entity Framework can generate a new database schema from the types in this model. The Entity Framework includes attributes that enable you to annotate classes and indicate which fields constitute primary and foreign keys, and the relationships between objects. The Entity Framework also includes a set of attributes that you can use to specify validation rules for data (whether a field allows null values, the maximum length of data in a field, and so on).

If you don't have access to the source code for the classes in the object model, or you do not wish to annotate them with Entity Framework attributes, you can use the Fluent API to specify how objects map to entities, how to validate the properties of entities, and how to specify the relationships between entities.

Using the Fluent API decouples the classes in your object model from the Entity Framework. You can quickly switch to a different ORM or implement an alternative mapping strategy (for example, if you need to change to a different type of database) without modifying the classes in your object model.

In the Entity Framework, you interact with the database through a *Context* object. The context object provides the connection to the database and implements the logic perform CRUD operations on the data in the database. The context object also performs the mapping between the object model of your application and the tables defined in the database. To achieve this, the context object exposes each table as a collection of objects either using the types that you specified (if you are following a code-first approach), or the types that the Entity Framework Data Model wizard generated from the database (if you are following a database-first approach).

To retrieve objects from the database, you create an instance of the context object, and then iterate through the corresponding collection. You can limit the volume of data fetched and specify filters to apply to the data in the form of LINQ to SQL queries. Behind the scenes, the context object generates an SQL **SELECT** query a **WHERE** clause that includes the appropriate conditions.

You can find detailed information about LINQ to SQL on MSDN.

Each collection provides **Add**, and **Remove** methods that an application can use to add an instance of an entity object to an entity collection, and remove an object from an entity collection. You update an object in an entity collection simply by modifying its property values. The context object tracks all changes made to the objects held in an entity collection. Any new objects that you add to a collection, objects that you remove from a collection, or changes that you make to objects, are sent to the database as SQL **INSERT**, **DELETE**, and **UPDATE** statements when you execute the **SaveChanges** method of the context object. The **SaveChanges** method performs these SQL operations as a transaction, so they will either all be applied, or if an error occurs they will all be undone. If an error occurs, you can write code to determine the cause of the error, correct it, and call **SaveChanges** again. If you have made changes to objects in other entity collections that are associated with the same context object, these changes will also be saved as part of the same transaction.

The Entity Framework abstracts the details of the connection to the database by using a configurable connection string, typically stored in the configuration file of the application. This string specifies the information needed to locate and access the database, together with the RDBMS-specific driver (or provider) to use. The following example shows a typical connection string for accessing the AdventureWorks2012 database running under Visual Studio LocalDB on the local machine (LocalDB is a development version of SQL Server provided with Visual Studio 2012). The **System.Data.SqlClient** provider implements the connection and communication logic for SQL Server.

```
<connectionStrings>
    ...
    <add name="AdventureWorksContext" connectionString="Data
Source=(localdb)\V11.0;Initial Catalog=AdventureWorks2012; Integrated
Security=SSPI" providerName="System.Data.SqlClient" />
</connectionStrings>
```

For detailed information about the Entity Framework, see "Entity Framework" page in the Data Developer Center, on MSDN.

If you need to connect to a different database but continue to use the same RDBMS provider (for example, if you wish to change from using SQL Server Express on your local computer to Windows Azure SQL Database in the cloud), you can change the parameters in the *name* property of the connection string. If you need to switch to a different RDBMS, install the assembly that implements the provider code for this RDBMS and then change the *providerName* property of the connection string to reference this assembly.

Using a Micro-ORM

ORMs implement a comprehensive mechanism for mapping tables in a database to collections of objects and tracking changes to these objects, but they can generate inefficient code if the developer implementing the mapping is not careful. For example, it is very easy to use an ORM to implement a mapping layer that retrieves all the columns for each row from a wide table, even if your code only requires the data from one or two of these columns. Micro-ORMs take a more minimalist approach, enabling you to fine tune the SQL operations that your code requires and improve efficiency. The purpose of a micro-ORM is simply to provide a fast, lightweight mapping between objects and entities.

A micro-ORM typically runs an SQL query, and converts the rows in the result to a collection of objects. You must define the types used to create these objects yourself. To perform create, update, and delete operations you compose the appropriate **INSERT**, **UPDATE**, and **DELETE** SQL statements in your application and execute them through the micro-ORM.

The primary advantage of a micro-ORM compared to an ORM is speed. A micro-ORM acts as a very lightweight wrapper around a set of SQL statements and does not attempt to track the state of objects that it retrieves. It does not implement the functionality typically provided by the Metadata Mapper, Interpreter, or Unit of Work patterns. The main disadvantage is that your application has to do this work itself, and you have to write the code!

The following example is based on Dapper, a popular micro-ORM that you can add to a Visual Studio 2012 project by using NuGet (install the "Dapper dot net" package). The example uses the SQL Server ADO.NET provider to connect to the AdventureWorks2012 database and retrieve product information. The Dapper library adds the generic **Query<T>** extension method to the connection class. The **Query** method takes a string containing an SQL **SELECT** statement, and returns an **IEnumerable<T>** result. The type parameter, **T**, specifies the type to which the **Query** method maps each row returned; in this case, it is the local **Address** class. Columns in the **SELECT** statement run by the query are mapped to fields with the same name in this class:

```
public class Address
{
    public int AddressId { get; set; }
    public string AddressLine1 { get; set; }
    public string AddressLine2 { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
}

...
string connectionString =
    @"Data Source=(local)\SQLExpress;Initial Catalog=AdventureWorks2012;
Integrated Security=SSPI";

using (System.Data.SqlClient.SqlConnection connection =
    new System.Data.SqlClient.SqlConnection(connectionString))
{
    connection.Open();
    var addresses = connection.Query<Address>(
        @"SELECT AddressId, AddressLine1, AddressLine2, City, PostalCode
          FROM Person.Address");

foreach(var a in addresses)
{
    // Process each row returned
}
...
```

The Data Access Block provides the SQL String Accessor, which also acts as a micro-ORM.

How the Shopping Application Accesses the SQL Server Database

In the Shopping application, the MvcWebApi web service receives REST requests sent from the user interface web application, validates these requests, and then converts them into the corresponding CRUD operations against the appropriate database. All incoming REST requests are routed to a controller based on the URI that the client application specifies. The controllers that handle the business logic for registering customers, logging in, and placing orders, indirectly use the Microsoft Entity Framework 5.0 to connect to the Adventure Works database and retrieve, create, update, and delete data. The designers implemented the Repository pattern to minimize dependencies that the controllers have on the Entity Framework.

Avoid building dependencies on a specific data access technology into the business logic of an application. Using the Repository pattern can help to reduce the chances of this occurring.

The purpose of the Repository pattern is to act as an intermediary between the object-relational mapping layer (implemented by the Entity Framework) and the data mapping layer that provides the objects for the controller classes. In the Shopping application, each repository class provides a set of APIs that enable a controller class to retrieve a database-neutral object from the repository, modify it, and store it back in the repository. The repository class has the responsibility for converting all the requests made by a controller into commands that it can pass to the underlying data store; in this case the Entity Framework. As well as removing any database-specific dependencies from the business logic in the controllers, this approach provides flexibility. If the designers decide to switch to a different data store, such as a document database, they can provide an alternative implementation of the repository classes that expose the same APIs to the controller classes.

Retrieving Data from the SQL Server Database

The application implements four repository classes, **PersonRepository**, **SalesOrderRepository**, **InventoryProductRepository**, and **StateProvinceRepository** that it uses to retrieve and manage customer, order, product inventory, and address information in the SQL Server database. There is not a one-to-one relationship between the repository classes and the tables in the database because some repository classes handle data from more than one table.

> *Each repository class handles the logic for a specific and discrete set of business data, sometimes referred to as a* Bounded Context. *For example, the* ***SalesOrderRepository*** *handles all the functionality associated with placing and maintaining an order, while the* ***PersonRepository*** *is focused on the logic for managing the details of customers. This approach isolates the data access functionality for sets of business operations and helps to reduce the impact that a change in the implementation of one repository class may have on the others.*

The section "Decoupling Entities from the Data Access Technology" in Appendix A, "How the MvcWebApi Web Service Works" describes how the repository classes use AutoMapper to create database-neutral domain objects from the database-specific entity objects.

The methods in each repository class receive and return database-neutral domain objects to the controllers that call them.

The repository classes connect to the database by using context objects (described later in this section). The following code example shows how methods in the **PersonRepository** class fetch **Person**, information from the database, either by specifying the ID of the person or their email address, reformat this data, and then return it to a controller:

```
public class PersonRepository : BaseRepository, IPersonRepository
{
    public DE.Person GetPerson(int personId)
    {
        using (var context = new PersonContext())
        {
            Person person = null;

            using (var transactionScope = this.GetTransactionScope())
            {
                person = context.Persons
                    .Include(p => p.Addresses)
                    .Include(p => p.CreditCards)
                    .Include(p => p.EmailAddresses)
                    .Include(p => p.Password)
                    .SingleOrDefault(p => p.BusinessEntityId == personId);

                transactionScope.Complete();
            }

            if (person == null)
            {
                return null;
            }

            var result = new DE.Person();
            var addresses = new List<DE.Address>();
            var creditCards = new List<DE.CreditCard>();

            Mapper.Map(person.Addresses, addresses);
            Mapper.Map(person.CreditCards, creditCards);
            Mapper.Map(person, result);

            addresses.ForEach(a => result.AddAddress(a));
            creditCards.ForEach(c => result.AddCreditCard(c));
            person.EmailAddresses.ToList().ForEach(
                e => result.AddEmailAddress(e.EmailAddress));

            return result;
        }
    }

    public DE.Person GetPersonByEmail(string emailAddress)
    {
        using (var context = new PersonContext())
        {
            PersonEmailAddress personEmail = null;
            using (var transactionScope = this.GetTransactionScope())
            {
                personEmail = context.EmailAddresses
                    .Include(pe => pe.Person)
                    .FirstOrDefault(ea => ea.EmailAddress.Equals(emailAddress));

                transactionScope.Complete();
            }

            if (personEmail == null)
            {
                return null;
            }

            var result = new DE.Person();
            Mapper.Map(personEmail.Person, result);
            return result;
        }
    }
    ...
}
```

All read operations are performed within a ***TransactionScope*** *object that specifies the* ***ReadCommitted*** *isolation level. This isolation level ensures that the data retrieved from the database is transactionally consistent. The* ***BaseRepository*** *class from which the repository classes inherit provides the* ***GetTransactionScope*** *method that creates this* ***TransactionScope*** *object.*

The **AccountController** class creates an instance of the **PersonRepository** class to retrieve and manage the details of customer accounts. The methods in this controller call the appropriate methods in a **PersonRepository** object to retrieve the details for a customer. For example, the **Get** method of the **AccountController** class invokes the **GetPerson** method of a **PersonRepository** object to fetch the details of a customer, as highlighted in the following code sample:

```
public class AccountController : ApiController
{
    private IPersonRepository personRepository;

    public AccountController(IPersonRepository personRepository, ...)
    {
        this.personRepository = personRepository;
        ...
    }
    ...
    public HttpResponseMessage Get(string id)
    {
        Guid guid;
        if (!Guid.TryParse(id, out guid))
        {
            ...
        }

        var person = this.personRepository.GetPerson(guid);
        ...
    }
    ...
}
```

As a second example of how the system queries information in the SQL Server database, when the **Shopping** application creates a new order, the **Post** method in the **OrdersController** class calls the **InventoryAndPrice-Check** method in the **InventoryService** class to verify the price of items in the order and also check whether Adventure Works still stocks these items.

The ***InventoryService*** *class simulates part of the functionality normally exposed by the warehousing and inventory systems inside Adventure Works. It is provided for illustrative purposes only.*

The **InventoryAndPriceCheck** method interacts with the database through an **InventoryProductRepository** object to check the price of an item. The following code highlights the relevant parts of the **InventoryAnd-PriceCheck** method:

```
public class InventoryService : IInventoryService
{
    ...
    private readonly IInventoryProductRepository inventoryProductRepository;

    ...
    public bool InventoryAndPriceCheck(ShoppingCart shoppingCart)
    {
        ...
        foreach (var shoppingCartItem in shoppingCart.ShoppingCartItems)
        {
            var inventoryProduct = this.inventoryProductRepository.
                GetInventoryProduct(shoppingCartItem.ProductId);

            ...
        }
        ...
    }
    ...
}
```

*The controller classes and the **InventoryService** class use the Unity Application Block to instantiate the repository objects. The code in each controller class references a repository by using an interface, such as **IInventoryProductRepository** in the example code shown above. The developers configured the Unity Application Block to inject a reference to the **InventoryProductRepository** class when this interface is referenced. The section "Instantiating Repository Objects" in Appendix A provides more information.*

You can download the Unity Application Block from MSDN, or you can add it to a Visual Studio 2012 project by using NuGet.

The repository classes connect to the SQL Server database by using a set of custom Entity Framework 5.0 context classes named **PersonContext**, **SalesOrderContext**, **InventoryProductContext**, and **StateProvinceContext**. These custom context classes expose the data from the database through groups of public properties that contain collections of entity objects.

For detailed information on how the repository classes connect to SQL Server through the Entity Framework by using the context classes, see the section "How the Entity Framework Repository Classes Work" in Appendix A.

To help making unit testing easier, the developers at Adventure Works followed a code-first approach and the entity classes are essentially stand-alone types that do not directly reference the Entity Framework. Instead, the context classes use the Entity Framework Fluent API to map the entity classes to tables in the SQL Server database. For example, the **Person** entity class referenced by the **PersonContext** class looks like this:

```
public class Person : ...
{
    public string PersonType { get; set; }
    public bool NameStyle { get; set; }
    public string Title { get; set; }
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string Suffix { get; set; }
    public int EmailPromotion { get; set; }
    public Guid PersonGuid { get; set; }
    public virtual PersonPassword Password { get; set; }
    public virtual ICollection<PersonEmailAddress> EmailAddresses { get; set; }
    public virtual ICollection<PersonBusinessEntityAddress> Addresses {get; set;}
    public virtual ICollection<PersonCreditCard> CreditCards { get; set; }
}
```

The **PersonContext** class maps this class to the **Person** table in the **Person** schema in the SQL Server database by creating an instance of the **PersonMap** class. This code also configures relationships with other objects, such as the **PersonCreditCard** class, that mirror the relationships in the database:

```
public class PersonMap : EntityTypeConfiguration<Person>
{
    public PersonMap()
        : base()
    {
        this.ToTable("Person", "Person");
        this.HasKey(p => p.BusinessEntityId);

        this.HasRequired(p => p.Password)
            .WithRequiredPrincipal(p => p.Person)
            .WillCascadeOnDelete(true);

        this.HasMany(p => p.EmailAddresses)
            .WithRequired(e => e.Person)
            .WillCascadeOnDelete(true);

        this.HasMany(p => p.CreditCards)
            .WithMany(c => c.Persons)
            .Map(map => map.ToTable("PersonCreditCard", "Sales")
                .MapLeftKey("BusinessEntityID")
                .MapRightKey("CreditCardID"));
    }
}
```

Inserting, Updating, and Deleting Data in the SQL Server Database

The repository classes that insert, update, and delete data use the **SaveChanges** method of the Entity Framework to perform these operations. For example, the **SalesOrderRepository** class exposes the **SaveOrder** method that the **Orders-Controller** uses to save the details of a new order. This method takes an **Order** domain object that contains the details of the order and uses the information in this object to populate a **SalesOrderHeader** object (by using AutoMapper). The method then adds this object to the **SalesOrderHeaders** collection in the context object before calling **SaveChanges**. The data is saved to the database within the scope of a transaction that uses the **ReadCommitted** isolation level (the **GetTransactionScope** method inherited from the **BaseRepository** class creates this transaction scope):

```
public class SalesOrderRepository : BaseRepository, ISalesOrderRepository
{
    public DE.Order SaveOrder(DE.Order order)
    {
        var salesOrderHeader = new SalesOrderHeader();
        Mapper.Map(order, salesOrderHeader);

        using (var transactionScope = this.GetTransactionScope())
        {
            context.SalesOrderHeaders.Add(salesOrderHeader);
            context.SaveChanges();

            transactionScope.Complete();
        }

        return order;
    }
    ...
}
```

Appendix A, "How the MvcWebApi Web Service Works," contains more detailed information on the structure of the web service and how the web service implements the Repository pattern to provide access to the SQL Server database through the Entity Framework.

Summary

This chapter has described the primary concerns that you should consider when you store the data for an application in a relational database. In this chapter, you saw how to support highvolume transaction throughput by normalizing a database and ensuring that transactions do not lock resources for an extended period. You also saw how to design a database to support query operations efficiently. However, in the real world, a minority of systems are OLTP-only or query-only. Most applications require databases that support a mixture of transactional and query operations. The key is to balance the design of the database to hit the sweet spot that maximizes performance for the majority of the time.

Scalability, availability, and performance are also important issues for any commercial system. Many RDBMSs include technologies that enable you to partition data and spread the load across multiple servers. In a system that supports geographically dispersed users, you can use this same strategy to place the data that a user commonly requires close to that user, reducing the latency of the application and improving the average response time of the system.

This chapter also discussed the decisions that Adventure Works made, why they decided to structure the data for customers and orders in the way that they did, and why they chose to store the database in the cloud by using Windows Azure SQL Database. This chapter also summarized the way in which the application connects to the database by using the Entity Framework, and how it uses the Repository pattern to abstract the details of the Entity Framework from the business logic of the system.

While the relational model has its undoubted strengths, it also has limitations, the most common being it can be difficult to handle non-relational data. Most modern RDBMS solutions have had to add non-standard features to optimize the way in which non-relational data is stored, and extend the way in which relationships that involve this data are handled. In many cases, it may be better to store non-relational data in a database that does not enforce a relational structure. Additionally, the inherent nature of relational databases can limit their scalability. This chapter has looked at some of the ways in which modern RDBMSs address this issue, but sometimes it is better to use a data storage technology that is naturally scalable. The remaining chapters in this book look at some common options.

More Information

All links in this book are accessible from the book's online bibliography available at: *http://msdn.microsoft.com/en-us/library/dn320459.aspx*.

- The "*Normalization*" page describing the benefits of normalizing a SQL Server database is available on MSDN at *http://msdn.microsoft.com/library/ms191178(v=sql.105).aspx*.
- The "*DML Triggers*" page describing how to implement database triggers in SQL Server 2012 is available on MSDN at *http://msdn.microsoft.com/library/ms178110.aspx*.
- The "*Stored Procedures (Database Engine)*" page describing the advantages of using stored procedures in SQL Server 2012 is available on MSDN at *http://msdn.microsoft.com/library/ms190782.aspx*.
- The page "*Online Transaction Processing vs. Decision Support*," available on MSDN at *http://msdn.microsoft.com/library/ms187669(v=sql.105).aspx*, summarizes common design decisions for meeting the requirements of databases that support OLTP and DSS operations.
- You can download the patterns & practices guide "*Building Hybrid Applications in the Cloud on Windows Azure*" from MSDN at *http://msdn.microsoft.com/library/hh871440.aspx*.
- The "*SQL Server Integration Services*" section in Microsoft SQL Server 2012 Books Online, is available at *http://msdn.microsoft.com/library/ms141026.aspx*.
- You can find detailed information on SQL Server indexes in Books Online for SQL Server 2012. The "*Indexes*" section is available on MSDN at *http://msdn.microsoft.com/library/ms175049.aspx*.
- The "*Partitioned Tables and Indexes*" section of Books Online for SQL Server 2012 is available on MSDN at *http://msdn.microsoft.com/library/ms190787.aspx*.
- You can find the page "*Query Processing Enhancements on Partitioned Tables and Indexes*" describing how SQL Server queries can take advantage of horizontal partitioning at *http://msdn.microsoft.com/library/ms345599.aspx*.
- The "*Top 10 Best Practices for Building a Large Scale Relational Data Warehouse*" page that summarizes best practices for building query intensive SQL Server databases is available on the SQLCAT website at *http://sqlcat.com/sqlcat/b/top10lists/archive/2008/02/06/top-10-best-practices-for-building-a-large-scale-relational-data-warehouse.aspx*.
- The section "*AlwaysOn Failover Cluster Instances (SQL Server)*" in Books Online for SQL Server 2012 is available at *http://msdn.microsoft.com/library/ms189134.aspx*.
- The Windows Azure calculator is available online at *http://www.windowsazure.com/en-us/pricing/calculator*.
- The page "*Federations in Windows Azure SQL Database*," describing how to use federations to scale out a database is available on MSDN at: *http://msdn.microsoft.com/en-us/library/windowsazure/hh597452.aspx*.

- You can find the "*General Guidelines and Limitations (Windows Azure SQL Database)*" page, on MSDN at *http://msdn.microsoft.com/library/windowsazure/ee336245.aspx*.
- The article "*Data Series: SQL Server in Windows Azure Virtual Machine vs. SQL Database*" is available at *http://blogs.msdn.com/b/windowsazure/archive/2012/06/26/data-series-sql-server-in-windows-azure-virtual-machine-vs-sql-database.aspx*.
- The article "*Choosing between SQL Server in Windows Azure VM & Windows Azure SQL Database*" is available at *http://blogs.msdn.com/b/windowsazure/archive/2013/02/14/choosing-between-sql-server-in-windows-azure-vm-amp-windows-azure-sql-database.aspx*.
- The "*SQL Data Sync*" page, which provides information about Windows Azure SQL Data Sync, is available on MSDN at *http://msdn.microsoft.com/library/hh456371.aspx*.
- Information about the *Enterprise Library* is available on MSDN at *http://msdn.microsoft.com/library/ff648951.aspx*.
- You can read about LINQ to SQL on the "*LINQ to SQL: .NET Language-Integrated Query for Relational Data*" page on MSDN at *http://msdn.microsoft.com/library/bb425822.aspx*.
- You can find information about the Entity Framework in the Data Developer Center, available online at *http://msdn.microsoft.com/data/ef.aspx*.
- You can download the *Unity Application Block* from MSDN at *http://msdn.microsoft.com/en-us/library/dn170416.aspx*.
- The Repository pattern is described on MSDN, at *http://msdn.microsoft.com/library/ff649690.aspx*.
- Information about the Dapper dot net package is available on NuGet *http://www.nuget.org/packages/Dapper/*.

4 Implementing a Key/Value Store

A relational database is an excellent storehouse for applications that need to perform ad hoc queries or perform complex analyses over data and their relationships; the generalized nature of the relational model makes it extremely flexible. However, in many cases, all an application requires of a database is to store and retrieve information quickly and efficiently, without the overhead of combining data from multiple tables. Additionally, the relational model is not good at handling non-uniform data, where entities might have different properties or attributes depending on their context. You can apply Entity-Attribute-Value modeling to a relational database to implement a flexible schema, but the result is that queries are often very time-consuming to perform, and even simple insert, update, and delete operations can involve maintaining integrity across several tables. In these situations, a key/value store is a better choice than a relational database for storing the data.

A key/value store focusses on the ability to store and retrieve data rather than the structure of that data; that is the concern of the application rather than the database. Consequently, most implementations are very quick and efficient, lending themselves to fast scalable applications that need to read and write large amounts of data.

This chapter describes the principles that underpin most large-scale key/value stores, and summarizes the concerns that you should address to use a key/value store for saving and querying data quickly and efficiently. This chapter also describes how Adventure Works used a key/value store to implement the shopping cart functionality for the Shopping application.

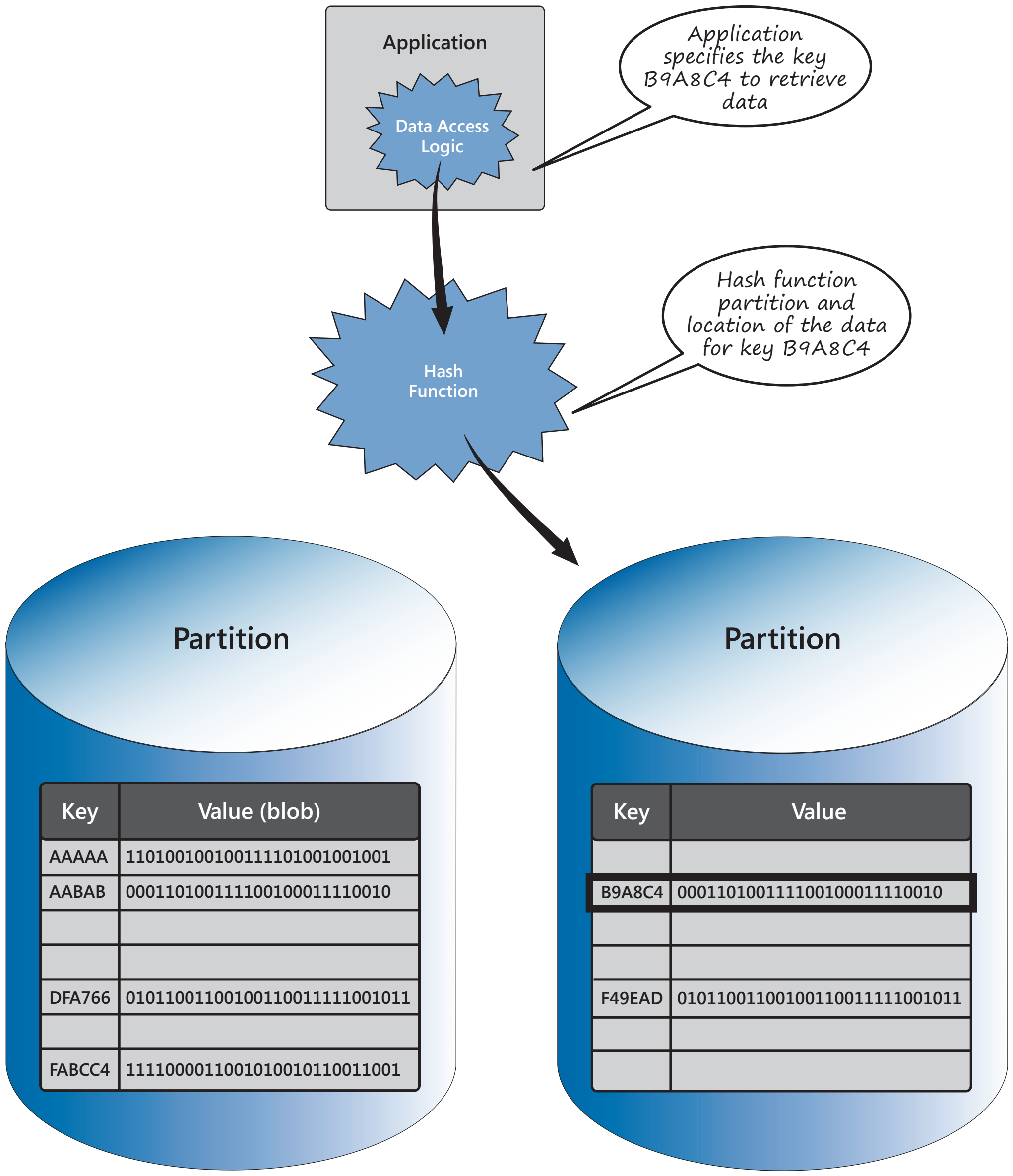
Designing a Key/Value Store

A key/value store is the most succinct form of the NoSQL database technologies, implementing a conceptually simple model for saving and fetching data.

Each piece of data in a key/value store consists of a pair; a unique key that can be used to identify the data, and a value. To save data, an application generates a key and associates it with a value, and submits the key/value pair to the key/value store. The key/value store writes the value to the database by using the key to determine the location for the value. In a large-scale NoSQL database, key/value storage is likely to be partitioned across distributed servers. To help ensure that the data is spread evenly across partitions, many key/value stores hash the key value to determine the storage location (partition and position within partition), as shown in Figure 1.

> *Some key/value stores are optimized to support queries that fetch contiguous sets of data items rather than individual values. These key/value stores frequently store data in a partition in key order rather than computing a location by hashing the key.*

When an application retrieves data, it provides the key to the key/value store. The key/value store hashes the key and accesses the resulting location to fetch the data.


Application
Data Access Logic
Application specifies the key B9A8C4 to retrieve data
Hash Function
Hash function partition and location of the data for key B9A8C4
Partition
Partition

Key	Value (blob)
AAAAA	11010010010011110100100100 1
AABAB	0001101001111001000111 10010
DFA766	0101100110010011001111 1001011
FABCC4	1111000011001010010110011001

Key	Value
B9A8C4	000110100111100100011110010
F49EAD	01011001100100110011111001011

Figure 1
Storing and retrieving data in a partitioned key/value store

If the key/value store that you are using does not support partitioning, you can implement the same functionality by creating multiple databases and implementing the partitioning logic on the client side as part of the application code.

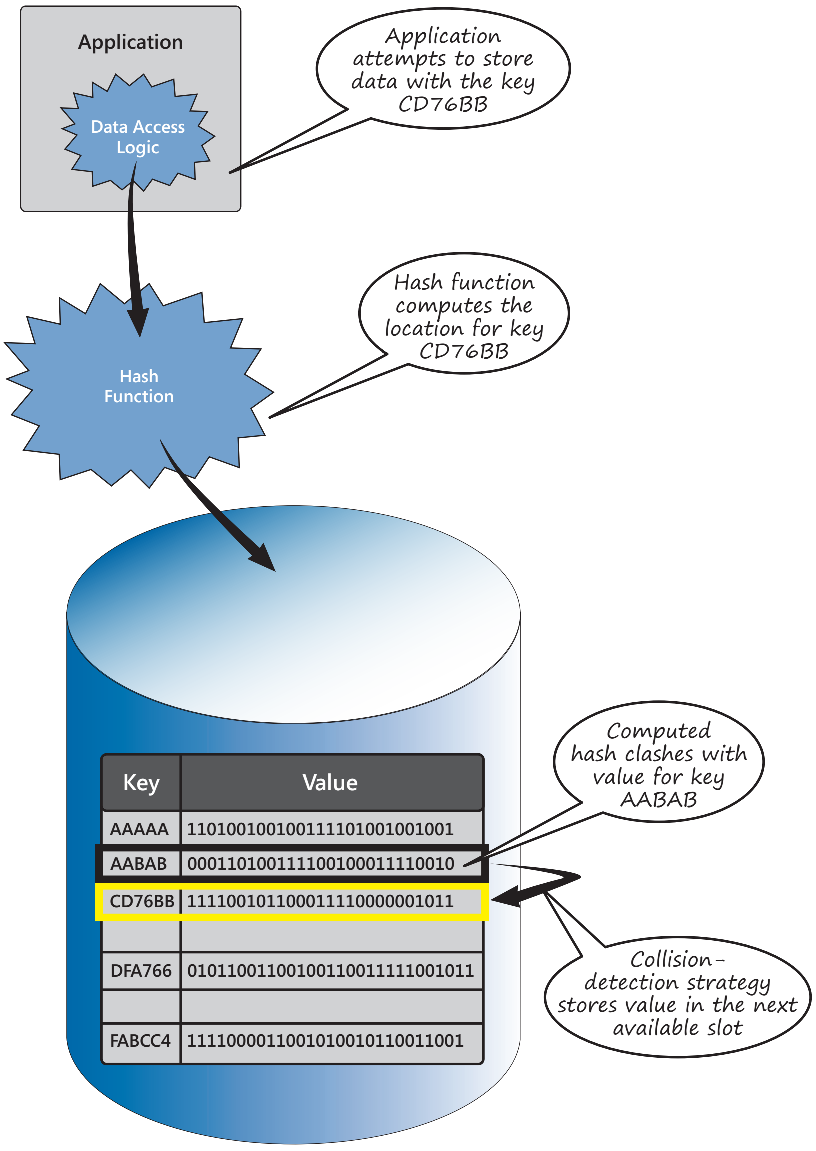
There are several factors that can affect the performance of a key/value store and the applications that use them, including:

- The efficiency of the hashing function.
- The design of the keys and the size of the values being stored and retrieved.
- The distribution of data across partitions.
- The functional patterns that applications follow to store and retrieve data.

The following sections discuss each of these factors in more detail.

Managing the Efficiency and Consistency of Hashing Functions

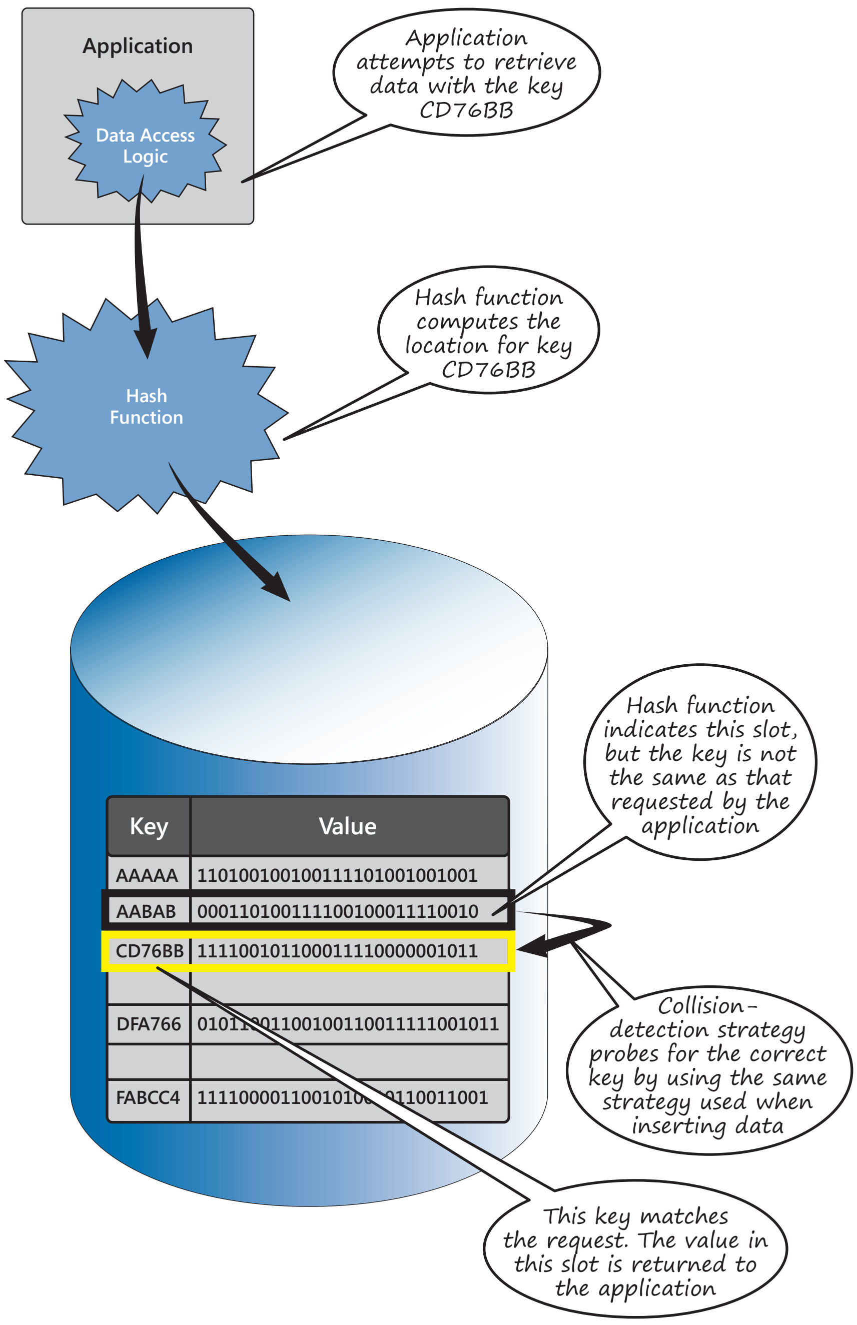
In a key/value store that uses hashing, the way in which the hash function calculates the location of a value based on its key is crucial to the performance of the database. If two keys hash to the same location, a collision occurs. A key/value store has to be prepared to detect these collisions and resolve them. Key/value databases can implement a variety of strategies to handle this scenario, such as computing a secondary hash (with a different function) to determine a new location, or performing a linear search for the next available slot in the partition starting at the location that was previously calculated. In either case, the additional work can decrease the responsiveness of the database. Figure 2 shows the process for detecting and handling a collision when adding a new item to a key/value store. In this example, the collision-detection strategy is to store the value in the next available slot in the partition.


Application
Data Access Logic
Application attempts to store data with the key CD76BB
Hash Function
Hash function computes the location for key CD76BB
Computed hash clashes with value for key AABAB
Collision-detection strategy stores value in the next available slot

Key	Value
AAAAA	1101001001001111010010010 01
AABAB	0001101001111001000111100 10
CD76BB	1111001011000111100000010 11
DFA766	0101100110010011001111110 01011
FABCC4	1111000011001010010110011 001

FIGURE 2
Detecting and handling collisions on insert

When an application retrieves data, the key/value store must examine the key held at the location calculated by using the hash function before returning it to the application in case a collision had previously occurred. If the key is not correct, then the key/value store must probe for the correct data using the same strategy as that used when inserting data. Figure 3 illustrates this process.


Application
Data Access Logic
Application attempts to retrieve data with the key CD76BB
Hash Function
Hash function computes the location for key CD76BB
Hash function indicates this slot, but the key is not the same as that requested by the application
Collision-detection strategy probes for the correct key by using the same strategy used when inserting data
This key matches the request. The value in this slot is returned to the application

Key	Value
AAAAA	110100100100111101001001001
AABAB	000110100111100100011110010
CD76BB	111100101100011110000001011
DFA766	01011[illegible]011001001100111111001011
FABCC4	11110000110010101[illegible]0110011001

FIGURE 3
Detecting and handling collisions when retrieving data

A few key/value stores expose the hashing functions that they use, and they enable an administrator to replace these functions with an implementation optimized for their applications. For key/value stores that do not support customization of the hash functions or collision-detection strategies that they use, you can reduce the likelihood of collisions by ensuring that the keyspace (the number of unique keys available to the database) greatly exceeds the volume of data items that you are likely to store. This may require you to monitor the space utilization within partitions very closely, and create new partitions if necessary, as more data is stored.

Several NoSQL key/value stores automatically resize and repartition themselves as data is loaded, up to some maximum number of partitions, to minimize the chances of collisions. If your data is likely to exceed this maximum value, then you may need to divide your data manually across a number of databases and implement additional logic in your application code to determine which database to use to store and retrieve an item with a given key. This approach is similar to implementing client-side partitioning described earlier.

If you are implementing client-side partitioning in your application code, make sure that your hashing function is consistent and independent of the number of partitions. If the number of partitions changes, it is important that your applications can continue to use the same hashing function to find all existing keys. For example, don't use modulus arithmetic on integer key values to determine the partition holding an item.

Designing Keys and Values

The data values stored in a key/value store are opaque to the database management system, and in most cases the keys provide the only means of access to the data values. You cannot easily retrieve data based on information held in the data values, so you should design your keys to support the most frequently performed queries. For example, in a personnel system, if you nearly always retrieve employee information by using the social security number, design the key around this element. Remember that you must ensure that keys are unique to avoid two data items having the same key value.

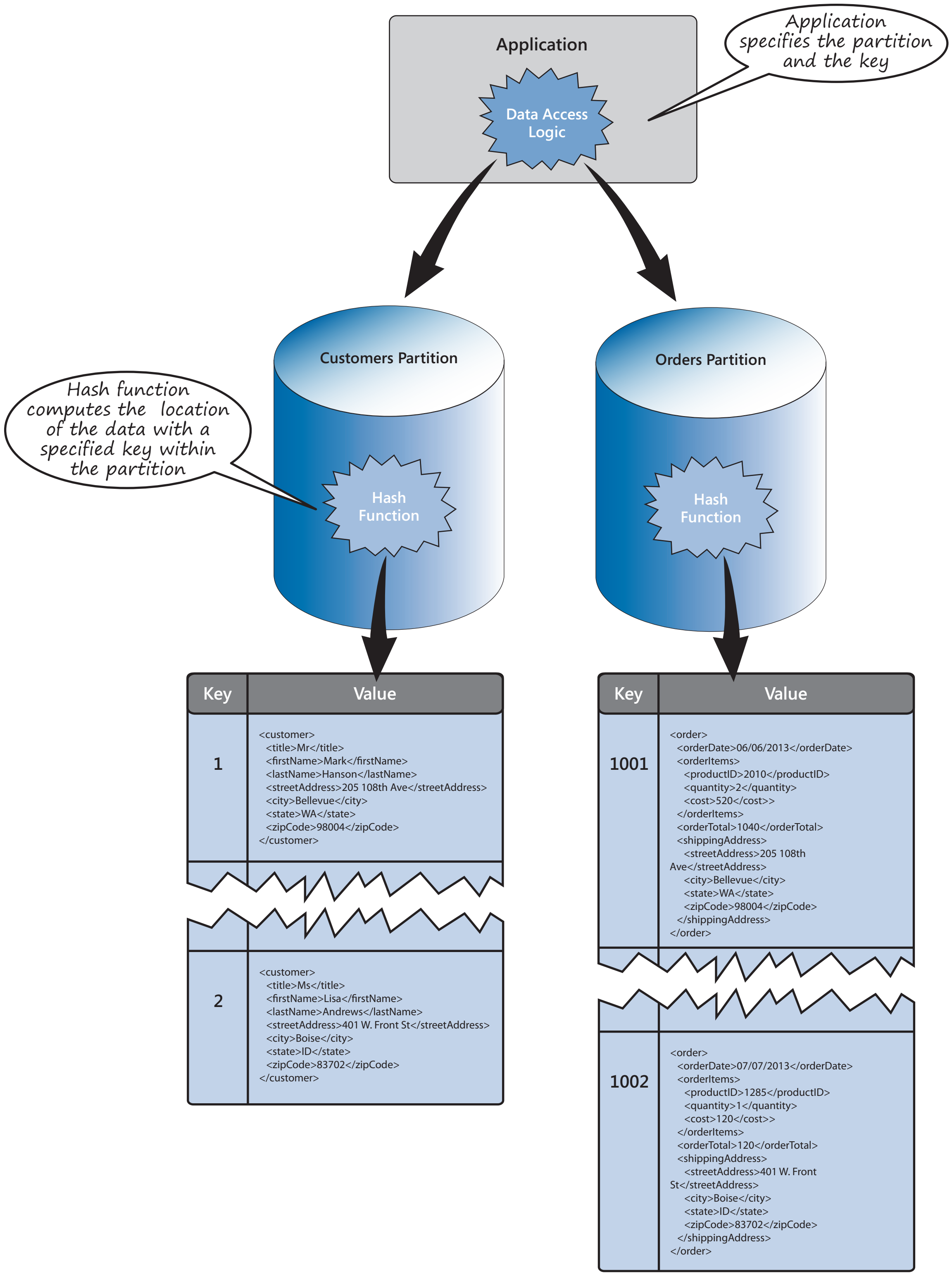
The format and structure that you use for keys can have an impact on the performance of the hashing function and the applications storing and retrieving values. If you select a lengthy key, it can take significant effort to generate the hash that identifies the location of the data. Additionally, each key has to be stored with the data (so that the key/value store can verify that it has found the correct data), so using large keys can affect the overall size of the database, especially if you have a vast number of items. However, the size and range of the keys must be sufficient to support the expected volume of data items. There is little point in using short keys with a limited keyspace if the number of data items is likely to exceed the quantity of keys available.

Repartitioning a key/value store can be a time-consuming and resource intensive process. If your key/value store implements manual partitioning, create as many partitions as you are likely to need when you first create the store, even if these partitions are very sparsely populated initially.

Because the data values are opaque to the database you can store practically any information in them. However, for the data to be useful to applications, they should include some metadata that describes the structure of the data, or be held in a format that an application can easily parse. Many systems store serialized objects, or even collections of objects, using a self-describing layout such as JSON or XML for maximum interoperability and portability. If you know that the database is only going to be accessed by applications built by using a specific technology, you can employ a less generalized but more efficient mechanism, such as .NET Framework serialization.

For key/value stores that enable you to store binary data, you can serialize objects held in memory and save them to the database with an appropriate key. This is a similar strategy to that proposed by object databases in the 1990s. However, this strategy is only suitable for standalone objects (and possibly simple, isolated collections of objects) that have no external dependencies or relationships with other objects. Such dependencies and relationships would necessitate storing references to objects serialized in other data values in the same or a different database, which may in turn reference objects held in further data values, and so on. If you must store complex networks of related items together in a database, it may be better to use a Graph database, as described in Chapter 7, "Implementing a Graph Database."

Some systems enable you to explicitly direct values to specific partitions and simply use the key to hash to a location in that partition. This strategy enables you to store data in specific partitions based on their data type, where each partition holds data values of a single type. Figure 4 shows this approach, with customers and orders information held in separate partitions (the values are stored as XML data).


Application
Data Access Logic
Application specifies the partition and the key
Customers Partition
Orders Partition
Hash function computes the location of the data with a specified key within the partition
Hash Function
Hash Function

Key	Value
1	see below
2	see below

Key 1:

```
<customer>
  <title>Mr</title>
  <firstName>Mark</firstName>
  <lastName>Hanson</lastName>
  <streetAddress>205 108th Ave</streetAddress>
  <city>Bellevue</city>
  <state>WA</state>
  <zipCode>98004</zipCode>
</customer>
```

Key 2:

```
<customer>
  <title>Ms</title>
  <firstName>Lisa</firstName>
  <lastName>Andrews</lastName>
  <streetAddress>401 W. Front St</streetAddress>
  <city>Boise</city>
  <state>ID</state>
  <zipCode>83702</zipCode>
</customer>
```

Key	Value
1001	see below
1002	see below

Key 1001:

```
<order>
  <orderDate>06/06/2013</orderDate>
  <orderItems>
    <productID>2010</productID>
    <quantity>2</quantity>
    <cost>520</cost>>
  </orderItems>
  <orderTotal>1040</orderTotal>
  <shippingAddress>
    <streetAddress>205 108th
Ave</streetAddress>
    <city>Bellevue</city>
    <state>WA</state>
    <zipCode>98004</zipCode>
  </shippingAddress>
</order>
```

Key 1002:

```
<order>
  <orderDate>07/07/2013</orderDate>
  <orderItems>
    <productID>1285</productID>
    <quantity>1</quantity>
    <cost>120</cost>>
  </orderItems>
  <orderTotal>120</orderTotal>
  <shippingAddress>
    <streetAddress>401 W. Front
St</streetAddress>
    <city>Boise</city>
    <state>ID</state>
    <zipCode>83702</zipCode>
  </shippingAddress>
</order>
```

Figure 4
Storing different types of data in different partitions

If you choose to store different types of data as values in the same partition, it can be useful if the structure of the keys provides some information about the nature of the corresponding values. For example, if you are storing information about customers and products, the keys for customers could be structured to indicate that the data to which they refer are customers, and keys for products should be formatted to indicate that they reference products. A common approach is to prefix the information in the key with a short piece of type information. In this way, an application retrieving data by using a customer key knows to expect the data for a customer and can handle the data that it retrieves appropriately.

The size of the values being stored also has considerable influence on the performance of a key/value store. Many key/value stores enable you to store very large values, possibly many gigabytes in size. However, the bigger the values being saved or retrieved, the longer it takes to save or retrieve them. Additionally, because the data being stored is opaque, most key/value stores do not support update operations that modify only part of the data (if you are storing data values in a binary format, then updates of this type may be meaningless anyway). This means that each modification to a saved value can require that an application retrieves the entire value, makes the appropriate changes, and stores the result back in the database. The section "Patterns for Accessing and Storing Data" later in this chapter describes the issues surrounding this mode of working in more detail.

Distributing Data Across Partitions

A well-balanced key/value store distributes data evenly across the partitions that comprise the database. Each partition can be a shard, located on a different computer. This strategy can help to improve scalability as the number of concurrent requests increases. Some key/value stores provide support for pluggable data placement strategies, enabling you to geolocate shards, and place them close to the applications that most frequently reference the data that they contain. This strategy can help to reduce network latency, as shown in Figure 5. However, for this approach to work effectively, the partitioning and hashing logic should be implemented close to the applications, following the Shared Nothing pattern described in Chapter 3, "Implementing a Relational Database."

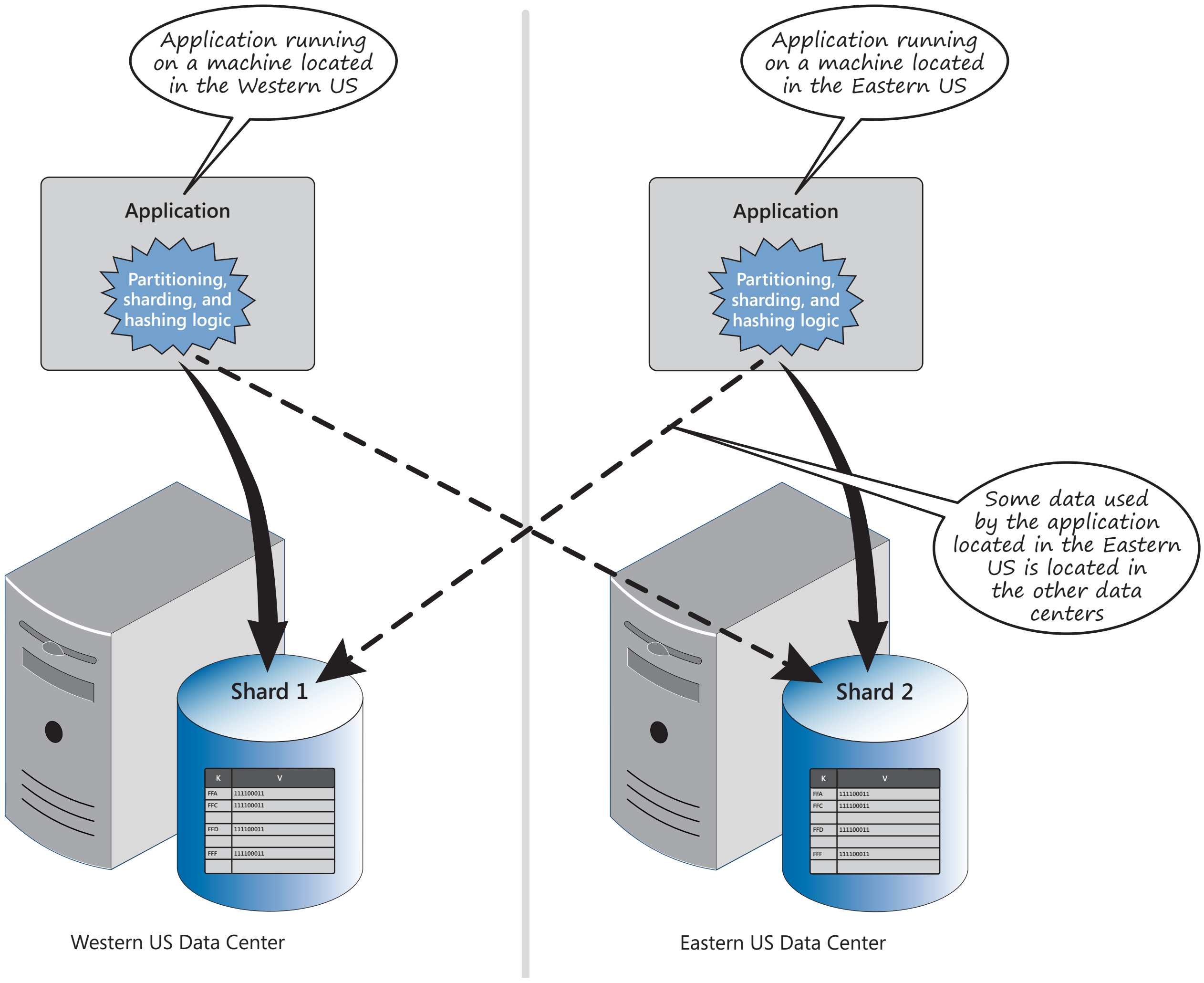

Application running on a machine located in the Western US
Application running on a machine located in the Eastern US
Application
Partitioning, sharding, and hashing logic
Application
Partitioning, sharding, and hashing logic
Some data used by the application located in the Eastern US is located in the other data centers
Shard 1
K V
FFA 111100011
FFC 111100011
FFD 111100011
FFF 111100011
Shard 2
K V
FFA 111100011
FFC 111100011
FFD 111100011
FFF 111100011
Western US Data Center
Eastern US Data Center

Figure 5
Using partitions and shards to place data close to the applications that use it

In this scheme, the hashing function determines which keys and values are mapped to which partition. A hashing function that balances the load across partitions can help to reduce the chance of hotspots (areas that are subjected to disproportionately large amounts of reading and writing) occurring in the database. Hotspots can cause I/O contention and lead to increased response times. This layout is suitable for applications that perform *point* queries (queries that return a single data item, given a key) such as "find the details for customer 99."

Some key/value stores take a different approach and implement a partitioning mechanism that provides efficient support for *range* queries, such as "find the information for products 1 through 50." Examples include the Windows Azure Table service that enables an application to direct data items to a specific partition, so all items in a given key range can be stored together and data is stored in key order within a partition rather than being hashed. A further mechanism that some key/value stores implement is to provide an index containing the keys and the locations of the values enabling the values to be quickly retrieved in sequence without hashing the keys. However, although these indexes can optimize queries, maintaining them can impose a significant overhead in systems that have to support a large number of insert, update, and delete operations. Finally, a small number of key/value stores provide basic search facilities and indexing over fields in the data values, enabling applications to query data based on the values in these fields. However, if you require this functionality extensively in your applications, you should consider using a document database or even a column-family database. See Chapter 5, "Implementing a Document Database," and Chapter 6, "Implementing a Column-Family Database" for more information.

The distinction between a key/value store and a document database or column-family database is somewhat artificial, and a number of NoSQL databases are available that offer a hybrid combination of key/value semantics with the features of a document database or column-family database. However, these databases often sacrifice a little performance to support the additional features that they offer, so if you simply require high-performance key/value functionality then a dedicated key/value store may be the most appropriate technology to use.

If your applications mainly perform range queries, then you should design your partitions to implement a hashing strategy that stores logically adjacent values together.

Patterns for Accessing and Storing Data

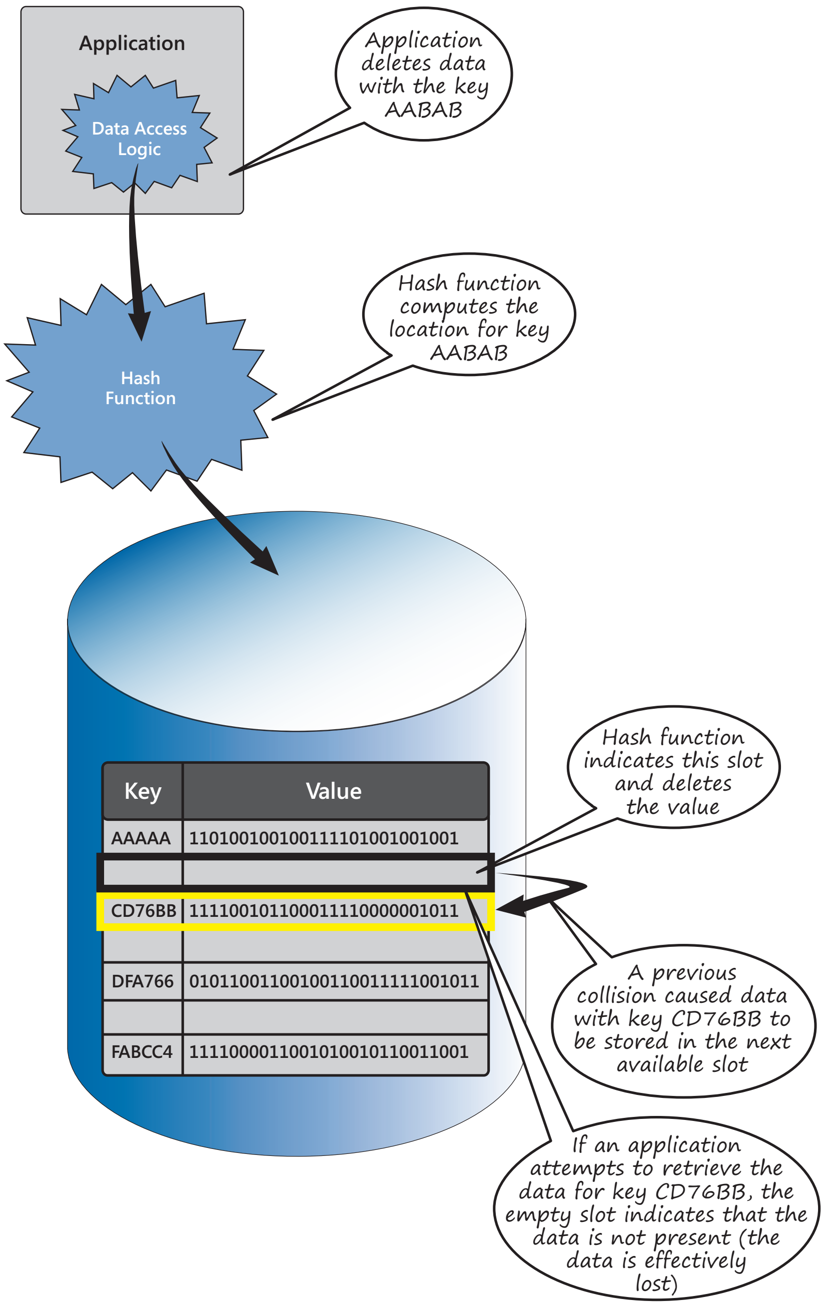
The main feature of most key/value stores is the ability to retrieve and store data very quickly, preferably by performing a single read or write operation. For this reason, many key/value stores are ideally suited for performing point queries although some are also optimized for performing range queries, as described in the section "Distributing Data Across Partitions."

A key/value store is not an optimal repository for applications that require data to be sorted by a particular field other than the key (remember that values and any fields that they contain are opaque to the key/value store). In many cases, the only way to implement queries such as these is to perform an iterative set of point queries, fetching the data for each key in turn and then sorting the data in memory.

Microsoft Research has created a map/reduce framework for Windows Azure, called Daytona. Using Daytona, you can scale-out map/reduce functionality across hundreds of servers. You can find more information about Daytona on the Microsoft Research website.

For the same reason, a key/value store is not a good fit for applications that perform ad hoc queries based on information other than key values, for queries that need to join data from several values based on some application-specific relationship, or for queries that need to perform complex analytical processing. Some vendors support map/reduce frameworks to assist in generating summary data enabling you to store aggregated information for each partition, but these aggregations must be maintained as data is added to or removed from a partition. A typical key/value store has no understanding of the information in the data values being added or removed, so such map/reduce frameworks often require that the developer provides their own custom code to analyze values and update aggregations as insert and deletes occur. This code can slow the system down in an insert/delete intensive system.

Frequent delete operations can impose a storage overhead in a key/value store. In many key/value stores that implement hashing, deleted values are not actually removed from storage, rather they are marked as being no longer present. The rationale behind this approach concerns the way in which a typical key/value store handles collisions. If the store simply removes an item, the key is available for use. However, any existing values that previously clashed with the now-deleted item run the risk of being lost. If an application attempts to find one of these items, the hash function returns the location of the deleted item in the now-empty slot, and the expected data is not retrieved. Figure 6 illustrates the problem.


Application
Data Access Logic
Application deletes data with the key AABAB
Hash Function
Hash function computes the location for key AABAB
Hash function indicates this slot and deletes the value
A previous collision caused data with key CD76BB to be stored in the next available slot
If an application attempts to retrieve the data for key CD76BB, the empty slot indicates that the data is not present (the data is effectively lost)

Key	Value
AAAAA	110100100100111101001001001
CD76BB	111100101100011110000001011
DFA766	01011001100100110011111001011
FABCC4	1111000011001010010110011001

Figure 6
Deleting an item, and losing items that collided with the deleted item

Marking the item as deleted rather than removing it enables the key/value store to follow its regular collision-detection strategy and locate the real information. However, if there is a large amount of data churn in a database, the number of deleted items can become significant, and it may be necessary to compact the database to remove these items and relocate colliding items. This could be a resource-intensive process, and may require that the partition is locked while it is compacted. Key/value stores that support replication often allow read-only access to data in a replica of the affected partition while it is being compacted, while others may support full read-write access to the replica (the changes made to the replica have to be applied to the compacted partition when the compaction operation is complete).

Key/value stores tend to impose few restrictions on the information that can be stored, subject to size limitations imposed by the vendor. In many cases, the size of an individual value can be several gigabytes, but in the unlikely scenarios where the maximum size of a data value is too restrictive you will have to split your data into smaller chunks, each with its own key value, and reconstruct the complete data item in your application code.

Some key/value stores implement data compression for values. This strategy reduces the size of the data that is stored at the expense of expanding the data when it is retrieved.

A greater concern is that many key/value stores do not support atomic update operations. An application must retrieve the data, delete the value from the database, change the data, and then save it back to the database. Note that if you don't delete the old data, the key/value store may either end up containing two versions of the same information with the same key, or reject the inserted data if the key/value store rigidly enforces uniqueness of keys. Applications that are update-intensive can generate a significant volume of I/O and network traffic, and can also result in inconsistencies and lost updates unless the key/value store implements pessimistic locking. To counter these issues, a few key/value stores support *upsert* (hybrid update/insert) operations. An upsert operation performs the following series of steps:

1. An application attempts to store a value by using a specified key.
2. If no current value exists in the database with this key, the data is added to the key/value store.
3. If an existing value has the same key as the new data, the key/value store overwrites the value with the new data.

How Adventure Works Used a Key/Value Store to Hold Shopping Cart Information

Refer to Chapter 2, "The Adventure Works Scenario" for a more detailed description of the functionality surrounding the shopping cart.

In the Shopping application, when a customer browses products and adds the first item to an order, the application creates a shopping cart object. If the customer selects further products, they are added to the same shopping cart object. After the customer has successfully placed the order and paid for the goods, the shopping cart is removed.

Each shopping cart is a self-contained entity, independent of any other shopping carts created for other customers. A shopping cart should also be durable. If the customer logs out or the customer's computer loses connectivity to the web application, the contents of the shopping cart must be saved so that they can be made available the next time the customer logs in.

Across the system, many 1000s of customers might be browsing products and placing orders at any one time, so the amount of shopping cart activity is potentially vast. To prevent the data storage used by this feature becoming a bottleneck in the system, the designers at Adventure Works decided save shopping cart information in a partitioned key/value store that supports fast query, insert, and delete access. The following sections describe the technologies and data structures that they used.

Understanding Windows Azure Table Service Concepts

The Shopping application runs as a web application that uses the MvcWebApi web service, both hosted using Windows Azure. Therefore, the developers at Adventure Works decided to use Windows Azure storage to save shopping cart data. This approach enabled Adventure Works to take advantage of the automatic provisioning implemented by the Windows Azure data centers, and meant that Adventure Works did not have to invest in expensive infrastructure to host the key/value store themselves.

Windows Azure actually provides two forms of storage; the Windows Azure Blob service and the Windows Azure Table service. The Windows Azure Blob service enables you to store large amounts of unstructured data. A single blob can be up to 1TB in size. The Windows Azure Table service lets you store semi-structured schema-less data, with each item having a maximum size of 1MB. In both cases you can save up to 100TB of information. For the Shopping application, the developers at Adventure Works chose the Table service because the data for each shopping cart is very unlikely to exceed 1MB, and it is useful to be able to map the contents of a shopping cart to a structure in storage for programming and debugging purposes.

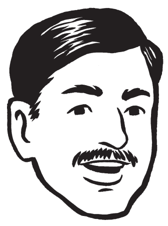
Using Windows Azure storage enables you to place the data that your web applications and services use close to where they are deployed, and can help to reduce network latency.

The Table service closely follows the key/value store model. The Table service is organized as a series of tables (these are not the same as tables in a relational database), and each table contains one or more *entities*. You can divide tables into one or more logical partitions, and you associate each entity with a two-part key that specifies the partition and an entity ID (also called the *row key*). Entities stored in the same partition have the same value for the partition element of this key (called the *partition key*), but each entity must have a unique row key within a partition. The Table service is optimized for performing range queries, and entities in the same partition are stored in row key order. The data for an entity comprises a set of key/value pairs known as *properties*. Like other NoSQL databases, the Table service is schema-less, so entities in the same table can have different sets of properties.

Storing and Retrieving Data in the Windows Azure Table Service

For more information about using the Windows Azure SDK to create and manage data in the Windows Azure Table service, see "How to use the Table Storage Service" on the Windows Azure developer website.

For further information about the structure of data supported by the Windows Azure Table service, see "Understanding the Table Service Data Model" on MSDN. The article, "Open Data Protocol by Example," also available on MSDN, contains a detailed description of OData.



OData is a generalized XML data serialization format, based on the ATOM feed standard.

The Table service uses the OData (Open Data Protocol) format for storing data. Additionally, the Table service exposes a REST interface so you can write applications that access the Table service by using any programming language that can generate HTTP REST requests. The Windows Azure SDK provides a set of APIs that wraps the REST interface. Using the .NET Framework you can define your own classes, and the APIs in the Windows Azure SDK can convert objects created by using these classes into OData format. The developers at Adventure Works decided to use this approach to develop the code that stores and retrieves shopping cart information. Note that OData supports only a limited set of primitive types. These types include integer and floating point numbers, Booleans, strings, dates, GUIDs, and byte arrays. If you need more complex types, it may be necessary to serialize the data as a string or byte array.

Defining the Shopping Cart Data

The developers at Adventure Works created a table named **ShoppingCart** in the Table service, and defined the **ShoppingCartTableEntity** class shown in the following code example to represent shopping cart information that the Mvc-WebApi web service stores in this table:

> *The value for the **PartitionKey** property is generated by the application to distribute shopping cart information evenly across partitions. This is discussed in more detail in the section "Implementing Scalability for Shopping Cart Information" later in this chapter.*

```
public sealed class ShoppingCartTableEntity : TableEntity
{
    public ShoppingCartTableEntity(string userId)
    {
        base.RowKey = userId;
        base.PartitionKey = ...;
    }

    public ShoppingCartTableEntity()
    {
    }

    public string ShoppingCartItemsJSON { get; set; }
    public Guid TrackingId { get; set; }
    ...
}
```

The data for the items in the shopping cart are stored in the **ShoppingCartItems-JSON** property (described shortly). The **TrackingId** property in the **Shopping-CartTableEntity** class is used elsewhere in the application to relate the shopping cart to an order created by using the contents of this shopping cart (Chapter 8, "Building a Polyglot Solution" provides more information about this relationship).

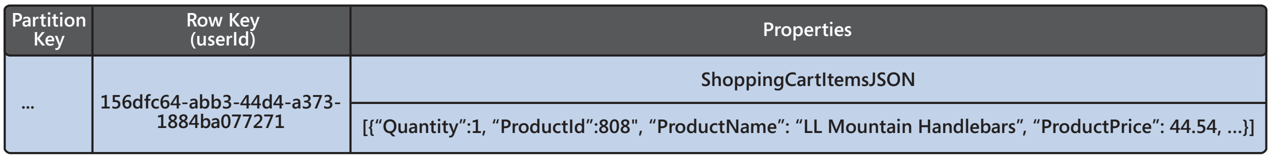
All classes that can be used to store data in the Table service must inherit from the ***TableEntity*** *type. This type defines the* ***PartitionKey*** *and* ***RowKey*** *properties that specify the logical partition in which to store an item, and the unique key that identifies the item in the partition. The remaining properties in the* ***ShoppingCartEntity*** *class (****ShoppingCartItemsJSON*** *and* ***TrackingId****) are serialized as properties of the row in the table when the item is saved.*

The developers also created the **ShoppingCartItemTableEntity** class to represent a single item in a shopping cart, containing the details of the product and the quantity ordered:

```
public class ShoppingCartItemTableEntity
{
    public int Quantity { get; set; }
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public decimal ProductPrice { get; set; }
    public string CheckoutErrorMessage { get; set; }
}
```

The **CheckoutErrorMessage** property is used to record any issues that occur when an order is placed, such as whether the price of the item has changed or it is discontinued.

A **ShoppingCartTableEntity** object contains a collection of **ShoppingCartItemTableEntity** objects serialized as a JSON string in the **ShoppingCartItemsJSON** property, as shown in Figure 7 below.



Partition Key	Row Key (userId)	Properties
...	156dfc64-abb3-44d4-a373-1884ba077271	ShoppingCartItemsJSON
		[{“Quantity”:1, “ProductId”:808", “ProductName”: “LL Mountain Handlebars”, “ProductPrice”: 44.54, ...}]

Figure 7
An example ShoppingCartTableEntity object

This serialization occurs when a shopping cart is saved using the Map method of the internal ShoppingCart-Mapper class. The **ShoppingCartMapper** class maps the data storage specific classes to the database-agnostic domain types used by the controllers in the MvcWebApi web service as described in Appendix A.

The code example below shows the **SaveShoppingCart** method in the **ShoppingCartRepository** class, highlighting how the **Map** method is used:

```
public class ShoppingCartRepository : IShoppingCartRepository
{
    ...
    public ShoppingCart SaveShoppingCart(ShoppingCart shoppingCart)
    {
        new ShoppingCartContext().Save(ShoppingCartMapper.Map(shoppingCart) );
        return shoppingCart;
    }
    ...
}
```

The following code fragments show how the relevant parts of the **Map** method are implemented:

```
internal static class ShoppingCartMapper
{
    ...
    public static ShoppingCartTableEntity Map(ShoppingCart shoppingCart)
    {
        ...
        var shoppingCartItems = new List<ShoppingCartItemTableEntity>();

        foreach (var shoppingCartItem in shoppingCart.ShoppingCartItems)
        {
            shoppingCartItems.Add(new ShoppingCartItemTableEntity()
            {
                ProductId = shoppingCartItem.ProductId,
                ProductName = shoppingCartItem.ProductName,
                ProductPrice = shoppingCartItem.ProductPrice,
                Quantity = shoppingCartItem.Quantity,
                CheckoutErrorMessage = shoppingCartItem.CheckoutErrorMessage
            });
        }

        result.ShoppingCartItemsJSON =
            new JavaScriptSerializer().Serialize(shoppingCartItems);

        return result;
    }
}
```

This method is overloaded, and another version of the **Map** method deserializes the data in the **ShoppingCartItemsJSON** property of a **ShoppingCartItemTableEntity** object into a collection of **ShoppingCartTableEntity** objects when a shopping cart is retrieved from the key/value store by the **GetShoppingCart** method in the **ShoppingCartRepository** class, as shown below:

```
public class ShoppingCartRepository : IShoppingCartRepository
{
    public ShoppingCart GetShoppingCart(string shoppingCartId)
    {
        var storedCart = new ShoppingCartContext().Get(shoppingCartId);
        return storedCart != null ? ShoppingCartMapper.Map(storedCart)
                                  : new ShoppingCart(shoppingCartId);
    }
    ...
}
```

The following code example shows how the **ShopperCartMapper** class implements this version of the **Map** method:

```
internal static class ShoppingCartMapper
{
    public static ShoppingCart Map(ShoppingCartTableEntity shoppingCart)
    {
        ...
        var shoppingCartItems = new JavaScriptSerializer().
            Deserialize<ICollection<ShoppingCartItemTableEntity>>(
                shoppingCart.ShoppingCartItemsJSON);

        foreach (var shoppingCartItem in shoppingCartItems)
        {
            result.AddItem(new ShoppingCartItem()
            {
                ProductId = shoppingCartItem.ProductId,
                ProductName = shoppingCartItem.ProductName,
                ProductPrice = shoppingCartItem.ProductPrice,
                Quantity = shoppingCartItem.Quantity,
                CheckoutErrorMessage = shoppingCartItem.CheckoutErrorMessage
            });
        }

        return result;
    }
    ...
}
```

Storing and Retrieving Shopping Cart Data

The developers at Adventure Works created a version of the **ShoppingCart-Repository** class specifically for interacting with the Table service. This class implements the **IShoppingCartRepository** interface described in Appendix A, and provides the **GetShoppingCart**, **SaveShoppingCart**, and **DeleteShopping-Cart** methods.

*As described in Appendix A, "How the MvcWebApi Web Service Works," the developers at Adventure Works followed the Repository pattern to isolate the specifics of the Windows Azure Table service from the remainder of the MvcWebApi web service. They used the Unity Application Block to configure the application to instantiate the **ShoppingCartRepository** object at runtime.*

The APIs exposed by Windows Azure SDK for storing and retrieving data from tables provide a simple object model, but behind the scenes they generate REST requests and convert data into OData format to interact with the Windows Azure Table service.

The **ShoppingCartRepository** class uses the **ShoppingCartContext** class to connect to the Table service by creating a **CloudTableClient** object (this type is part of the Windows Azure SDK). The following example shows this code in the constructor of the **ShoppingCartContext** class:

```
public sealed class ShoppingCartContext
{
    private const string TableName = "ShoppingCart";
    private CloudTableClient tableClient;

    public ShoppingCartContext()
    {
        string connectionString;
        ...

        try
        {
            this.tableClient = CloudStorageAccount.Parse(connectionString).
                CreateCloudTableClient();
        }
        catch
        {
            this.tableClient = null;
        }
    }
    ...
}
```

The **CloudTableClient** class in the Windows Azure SDK provides an interface to the **Table** service that enables an application to perform simple CRUD operations against a table. For instance, to add a new object to a table, you obtain a reference to that table by using the **GetTableReference** method of a **CloudTableClient** object, create an Insert **TableOperation** object that specifies the data to add, and then apply this operation to the table by calling the **Execute** method. You can also create a **Replace TableOperation** to modify an existing object in a table, and a **Delete TableOperation** to remove an object from a table.

The following code example shows the **Save** method in the **ShoppingCartContext** class. This method stores the details of a shopping cart to the **ShoppingCart** table (this table is created when you configure the application). Note that the **ShoppingCartContext** class uses an **InsertOrReplace TableOperation** to either insert a new shopping cart into the table or replace an existing shopping cart that has the same key (an upsert operation):

```
public sealed class ShoppingCartContext
{
    private const string TableName = "ShoppingCart";
    private CloudTableClient tableClient;

    ...

    public void Save(ShoppingCartTableEntity shoppingCart)
    {
        var table = this.tableClient.GetTableReference(TableName);
        TableOperation InsertOrReplaceOperation =
            TableOperation.InsertOrReplace(shoppingCart);
        table.Execute(InsertOrReplaceOperation);
    }
}
```

The **GetShoppingCart** method in the **ShoppingCartRepository** class fetches the shopping cart for a customer. This method calls the **Get** method of the **ShoppingCartContext** class, passing the user ID of the customer as the rowKey parameter to this method. The **Get** method creates a **Retrieve TableOperation** to fetch the shopping cart from the **ShoppingCart** table. The **Retrieve** operation requires the partition key and the row key that identifies the shopping cart:

> *For scalability purposes, the developers decided to spread the data across a set of partitions. The **CalculatePartitionKey** method in the **ShoppingCartTableEntity** class determines the partition to use to store a given shopping cart. The section "Implementing Scalability for Shopping Cart Information" in this chapter describes the **CalculationPartitionKey** method used to generate the partition key for a shopping cart in more detail.*

```
public sealed class ShoppingCartContext
{
    private const string TableName = "ShoppingCart";
    private CloudTableClient tableClient;

    ...

    public ShoppingCartTableEntity Get(string rowKey)
    {
        var table = this.tableClient.GetTableReference(TableName);
        var partitionKey = ShoppingCartTableEntity.CalculatePartitionKey(rowKey);
        TableOperation retrieveOperation = TableOperation.
            Retrieve<ShoppingCartTableEntity>(partitionKey, rowKey);
        try
        {
            TableResult retrievedResult = table.Execute(retrieveOperation);
            return (ShoppingCartTableEntity)retrievedResult.Result;
        }
        catch (System.Exception)
        {
            return null;
        }
    }

    ...
}
```

When the customer places an order, the new order is created from the contents of the shopping cart, and then the **ShoppingCartRepository** removes the shopping cart by using the **Delete** method of the **ShoppingCartContext** class. This method deletes the shopping cart from the **ShoppingCart** table and calls the Execute method to persist this change in the **Table** service.

```
public sealed class ShoppingCartContext
{
    private const string TableName = "ShoppingCart";
    private CloudTableClient tableClient;

    ...

    public void Delete(ShoppingCartTableEntity shoppingCart)
    {
        var table = this.tableClient.GetTableReference(TableName);
        TableOperation deleteOperation =
            TableOperation.Delete(shoppingCart);
        table.Execute(deleteOperation);
    }
}
```

Implementing a Key/Value Store to Maximize Scalability, Availability, and Consistency

A scalable key/value store must be capable of supporting a large number of users querying and maintaining a big expanse of data. This section summarizes in general terms how key/value stores attempt to maximize concurrency while ensuring scalability and availability.

Maximizing Scalability

To accommodate scalability requirements, most commercial key/value stores provide transparent support for partitioning data horizontally, based on the key (sharding). Typically, the partitioning scheme is hidden from the business logic of the applications that use it, either by being embedded in the database system software that manages the key/value store or through pluggable libraries and modules that a developer can link into the applications. For key/value stores that implement partitioning in the database system software, the partitioning scheme is frequently controlled by using management APIs and configuration files that enable an administrator to add or remove partitions, monitor partitions, take them off line to repair them if necessary, and then bring them back online. In some cases, the management APIs are exposed through command line utilities, while other systems provide specific management applications, often in the form of web applications or services.

Some implementations, such as the Windows Azure Table service, have inherent support for geolocating data close to the applications that use it. You can divide your data up into partitions organized geographically, and create these partitions in the same data centers that host instances of a web application that uses this data. This approach enables you to easily implement a model similar to that shown in Figure 5 earlier in this chapter.

Ensuring Availability

Most commercial key/value stores ensure availability by replicating partitions and shards; the primary/secondary and the peer-to-peer replication models are both commonplace (Chapter 1, "Data Storage for Modern High-Performance Business Applications" describes these two strategies in more detail). The degree of control that you have over the replication topology depends on the NoSQL solution that you employ; in some systems you explicitly create the replicas for each server, while in others (such as the Windows Azure Table service) replication is automatic.

A significant number of key/value stores maximize performance by storing data in RAM, writing data to disk only when necessary (when memory is nearly full, or when a server is being shut down, for example). These databases act like large caches, with the most active data held in memory and other less frequently accessed items flushed to disk. Some systems even support data expiration, automatically removing items from storage after a specified period of time (this feature is useful for managing short-lived data such as session state). To prevent data from being lost in the event of a systems failure, many databases record all insert, update (if supported), and delete operations, writing records to an append-only log file (writing to an append-only file is typically a very fast operation). If the server should crash, the recovery process can restore missing data by rolling forward from the log file. Alternatively, some systems can recover a failed server directly from a replica held by a different server.

Maximizing Consistency

Most key/value stores aim to provide extremely fast access to data, and frequently sacrifice consistency features to achieve this objective. A few key/value stores support transactional operations that span multiple data items, while others do not even guarantee that writing to a single value is an atomic operation (this restriction occurs mainly in key/value stores that support very big multi-gigabyte values where implementing atomicity would be prohibitively expensive).

Key/value stores can use read and write quorums to present a consistent view of data, as described in Chapter 1. Versioning is commonly used to help resolve conflicts, although several key/value stores also provide distributed locking primitives that can help to prevent concurrent updates to the same values.

How Adventure Works Implemented Scalability and Availability, and Maximized Consistency for Shopping Cart Information

The decision that the developers at Adventure Works made to use the Windows Azure Table service for holding shopping cart information was influenced by many of the built-in features provided by the Windows Azure platform, as described in the following sections.

Implementing Scalability for Shopping Cart Information

The Table service can load-balance the data in a table based on the partition key, and requests that store and retrieve data from different partitions in a single table can be directed to different servers. Load balancing is managed by the infrastructure at the datacenter and you do not have to modify your application code to take advantage of it, although the design of your logical partition key can have an impact on performance.

Internally the data for a single partition may itself be spread across multiple servers; this scheme prevents the size of a partition being limited to the resources available on a single server. However, this physical storage scheme is transparent to the client application and all requests for that partition will be handled by a single server. Figure 8 illustrates this scheme.

A single server can handle requests for data in multiple partitions, but requests for data in any given partition will be always controlled by a single server, even if the data for that partition has been spread across multiple servers internally.

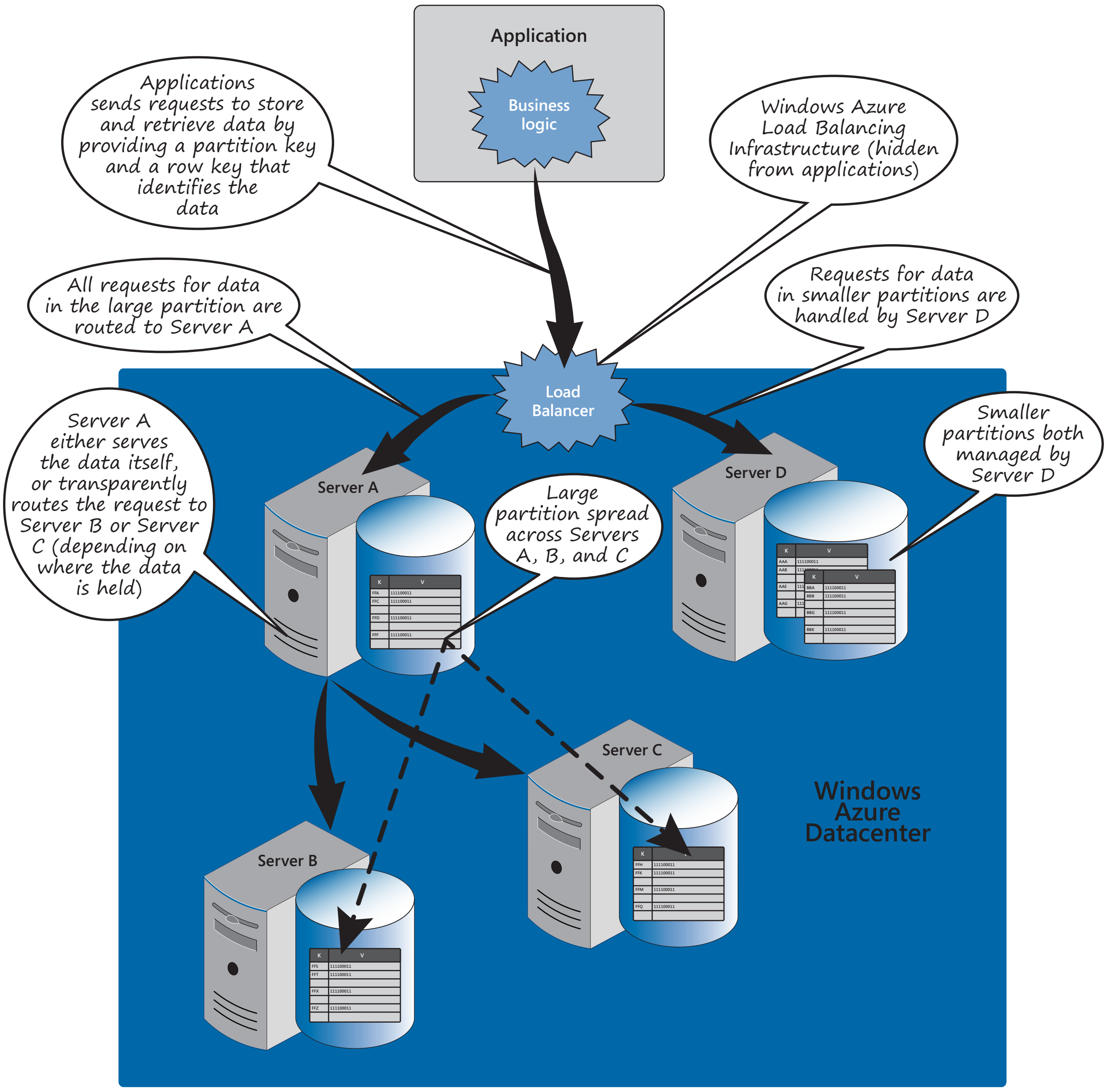
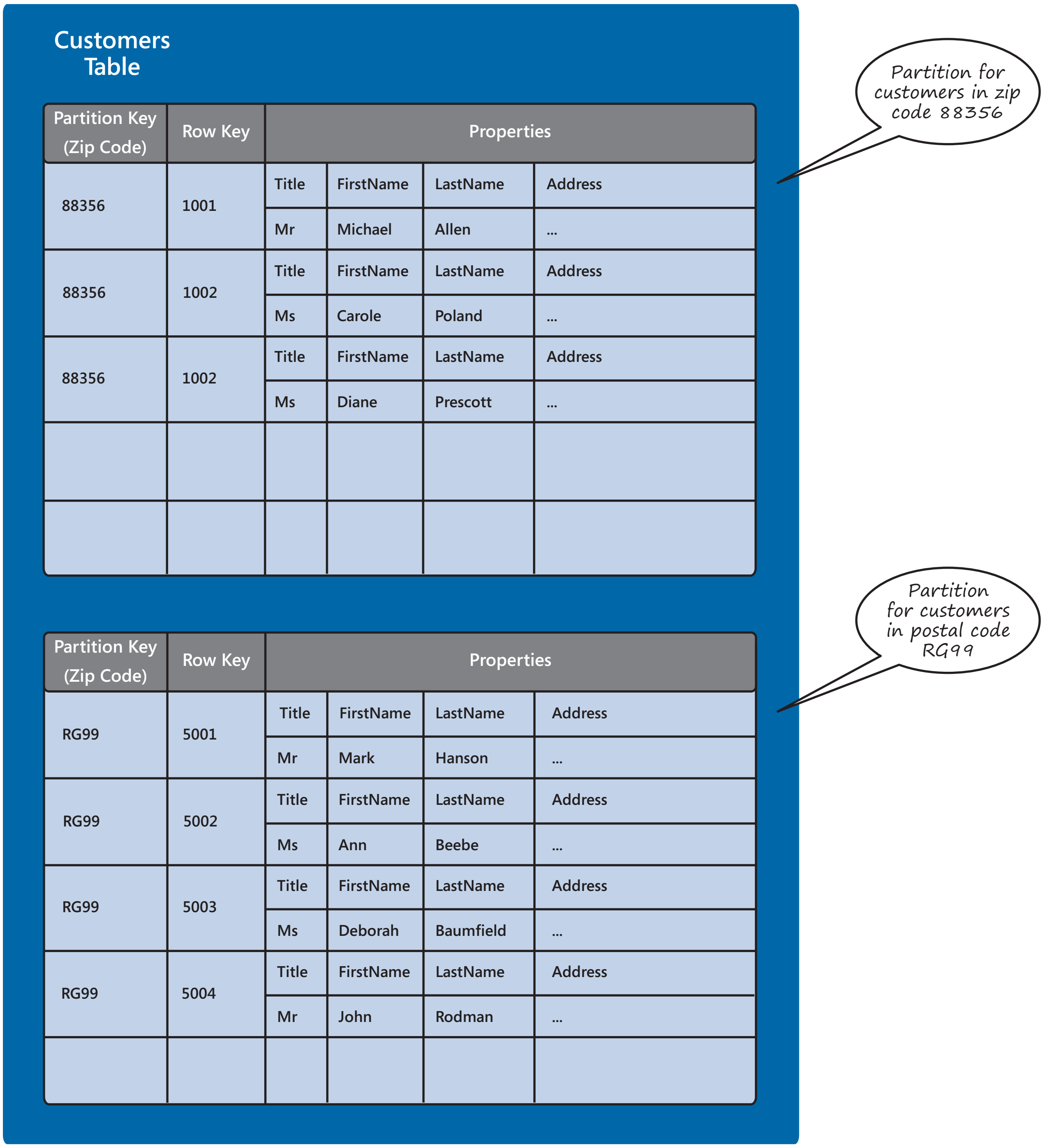

Application
Business logic
Applications sends requests to store and retrieve data by providing a partition key and a row key that identifies the data
Windows Azure Load Balancing Infrastructure (hidden from applications)
All requests for data in the large partition are routed to Server A
Requests for data in smaller partitions are handled by Server D
Load Balancer
Server A either serves the data itself, or transparently routes the request to Server B or Server C (depending on where the data is held)
Server A
Large partition spread across Servers A, B, and C
Server D
Smaller partitions both managed by Server D
Server C
Server B
Windows Azure Datacenter

Figure 8
Load-balancing data in the Windows Azure Table service

You should select the partition key carefully because it can have a significant impact on the scalability of your application. The throughput of each partition is ultimately limited by the number of concurrent requests that are handled by the server managing that partition. If your data is subject to a high volume of point queries, it may be better to implement a schema that creates a larger number of small partitions rather than fewer large partitions. This is because small partitions can be more easily distributed and load-balanced across separate servers in the Windows Azure datacenter, and each partition will be consequently subjected to fewer requests. For example, if you are storing information about customers and their addresses, you could partition the data based on the zip code or postal code of the address, and use the customer ID as the row key within each partition as shown in Figure 9.

Partition keys and row keys for tables in the Table service must be string values.


Customers Table
Partition for customers in zip code 88356
Partition for customers in postal code RG99

Partition Key (Zip Code)	Row Key	Properties			
88356	1001	Title	FirstName	LastName	Address
		Mr	Michael	Allen	...
88356	1002	Title	FirstName	LastName	Address
		Ms	Carole	Poland	...
88356	1002	Title	FirstName	LastName	Address
		Ms	Diane	Prescott	...

Partition Key (Zip Code)	Row Key	Properties			
RG99	5001	Title	FirstName	LastName	Address
		Mr	Mark	Hanson	...
RG99	5002	Title	FirstName	LastName	Address
		Ms	Ann	Beebe	...
RG99	5003	Title	FirstName	LastName	Address
		Ms	Deborah	Baumfield	...
RG99	5004	Title	FirstName	LastName	Address
		Mr	John	Rodman	...

Figure 9
The logical structure of Customer data in the Windows Azure Table service

You should avoid partitioning your data by using a partition key that can lead to hotspots when data is inserted. For example, if you partition customers by their country rather than by zip code or postal code, each new customer for a given country will be appended to the partition for that country (assuming that customer IDs are assigned in a monotonic increasing sequence). Each partition is likely to be quite large, and possibly subjected to a high number of inserts which could adversely affect the performance of other concurrent operations that access the same partition. Using a partition key that divides the data into smaller groups can help to alleviate the volume of writes occurring in each one.

You should also consider the effects that having many small partitions can have on your application if it performs a significant number of range queries. Ideally, the rows that satisfy a range query should be stored together in a single partition; if the data for a given query is spread across multiple partitions then your application may have make multiple requests and visit all of these partitions.

Remember that the Windows Azure Table service stores items in row key order in a partition and does not use hashing. This sequencing helps to optimize range queries and enumeration, which are important features of the Table service.

In the Shopping application, shopping carts are stored and retrieved individually by using point queries. Therefore the developers at Adventure Works wanted to ensure that the shopping carts for all customers were spread evenly throughout the ShoppingCart Table. They decided to distribute this information across 100 partitions, based on a hash of the customer's ID. The ID of a customer is invariant and the **ShoppingCartTableEntity** class makes use of the **GetHashCode** method implemented by all .NET Framework classes that inherit directly or indirectly from **System.Object** to generate a hash value for the customer ID. The **ShoppingCartTableEntity** class then uses modulus arithmetic to determine in which partition the shopping cart should reside. The generated partition key has the form "ShoppingCart_*nnn*" where *nnn* is a value between 1 and 100. The following code example shows the **CalculatePartitionKey** method in the **ShoppingCartTableEntity** class, and illustrates how this method is used to populate the **PartitionKey** property:

```
public sealed class ShoppingCartTableEntity : TableEntity
{
    public ShoppingCartTableEntity(string userId)
    {
        base.RowKey = userId;
        base.PartitionKey =
            ShoppingCartTableEntity.CalculatePartitionKey(userId);
    }
    ...
    private static int NumberOfBuckets = 100;

    public static string CalculatePartitionKey(string userId)
    {
        ...
        if (string.IsNullOrWhiteSpace(userId))
        {
            throw new ArgumentException(
                "userId cannot be null, empty, or only whitespace.");
        }

        return string.Format("ShoppingCart_{0:000}",
           (Math.Abs(userId.GetHashCode()) %
              ShoppingCartTableEntity.NumberOfBuckets) + 1);
    }
}
```

Each Windows Azure table is stored within a Windows Azure storage account hosted at datacenters within a region specified by the developer. Currently the Shopping application uses a single storage account to hold all shopping carts. However, the developers decided to carefully monitor the performance of the shopping cart because it is a critical part of the Shopping application. If necessary, they can create a storage account in each different region supported by Windows Azure, create a **ShoppingCart** table in this storage account, and deploy an instance of the Shopping application to the datacenters at each region. The business logic that handles shopping cart functionality in the Shopping application can then connect to the local instance of the **ShoppingCart** table, so all shopping cart information is stored within the same locality as the instances of the applications that are using it and reduce the amount of contention and network latency that might occur.

*The sample application provided with this guide only creates a single **ShoppingCart** table in a single Windows Azure storage account.*

The Shopping application uses a configurable connection string in the cloud service configuration file to specify the storage account that it should use. An administrator can change this connection string without requiring that you modify the code or rebuild the application.

Ensuring Availability for Shopping Cart Information

When you create a Windows Azure storage account, you specify the geographic region in which the servers holding the storage account should be located. A single geographic region can actually contain several datacenters hundreds of miles apart, and each datacenter contains a large number of servers.

To maximize availability, Windows Azure replicates the data in each storage account three times at the location in which it is held. When an application writes data to a table in the Windows Azure Table service, the operation is not marked as successful until all three servers in the replication cluster indicate that they have saved the data. The servers in a replication cluster are physically isolated with their own redundant networking and power supplies, so a hardware failure in one server should not affect others in the same cluster. If a server should fail, then another can be quickly enrolled and added to the replication cluster.

Replication is managed and controlled by the Windows Azure infrastructure.

The blog post "Introducing Geo-replication for Windows Azure Storage" provides more detailed information on how Windows Azure replication works.

To handle environmental disasters such as fire, flood, and earthquake, each replication cluster in a datacenter is also replicated to another remote datacenter. This replication occurs in the background. If a datacenter is unavailable, the Windows Azure DNS infrastructure automatically routes requests to a working datacenter.

Maximizing Consistency for Shopping Cart Information

The Windows Azure Table service implements a highly consistent model. When an application writes data to a table, the write operation is not completed until the servers in the local replication cluster have all indicated that it was performed successfully. The servers in the replication cluster are all located in the same datacenter and are connected by high bandwidth redundant networks, so latency is minimal. This scheme means that all read requests routed by the Windows Azure infrastructure to any server in the local replication cluster will be consistent.

Replication between sites is performed asynchronously, and in the event of a complete failure of a datacenter it may take a short time to recover data before applications can resume.

Unusually for key/value stores, the Table service supports transactions for operations that manipulate items within the same logical partition, although transactions cannot span multiple tables and partitions.

To maximize consistency and make the best use of transactions, try and avoid cross-partition relationships and queries between entities in a table wherever possible.

Timestamps work well as a versioning mechanism if all servers in a replica set are synchronized and physically close to each other, as they are in a Windows Azure datacenter. However, you should not depend on timestamps for situations where servers are remote from each other. The time required to transmit data between them can become significant increasing the window of opportunity for inaccuracies and conflicts. In these situations, using a vector clock may be a more appropriate strategy.

Windows Azure implements snapshot isolation for read operations for queries that retrieve multiple items from a single partition. Such a query will always have a consistent view of the data that it retrieves, and it will only see the data that was committed prior to the start of the query. The snapshot mechanism does not block concurrent changes made by other applications, they will just not be visible until the changes have been committed and the query is re-run.

> *The developers at Adventure Works chose not to use the* ***TableServiceContext*** *class for retrieving and storing data because the Shopping application only manipulates the data for one shopping cart at a time. The* ***CloudTableClient*** *class provides a more efficient interface for performing these types of operations.*

The Windows Azure Table service supports optimistic concurrency. Each item in a table has a **Timestamp** property. The value of this property is maintained by Windows Azure to record the date and time that the last modification occurred. When you retrieve an item, you also retrieve the **Timestamp** property. When you save data back to storage, you include the **Timestamp** property. The Table service automatically checks to make sure that the timestamp of the data in the table matches the value you have provided, and then saves the data and updates the timestamp. If the timestamp in the table has changed since your application retrieved it, the save operation fails and your application is notified with an exception. In this case you can retrieve the latest version of the data and repeat the update if appropriate.

> *You should treat the* ***Timestamp*** *property of a value retrieved from the Table service as opaque. You should not attempt to modify it in your own code, nor should you insert it yourself when you create a new value.*

Summary

This chapter has described how to use a key/value store as a repository for storing and retrieving data quickly and efficiently. In this chapter, you learned that key/value stores are optimized for applications that store and retrieve data by using a single key. The data values stored in a key/value store are largely opaque and their format can vary between different items. The important point is that the structure of this data is the concern of the application and not the database.

However, this lack of knowledge about the data by a key/value store means an application cannot easily filter data by value. An application can only retrieve data by using the key, and any filtering must be performed by the application in memory. This approach can lead to inefficiencies and poor performance; a key/value store is probably not suitable for applications that require this functionality.

This chapter has also shown how Adventure Works used the Windows Azure Table service as a key/value store for holding shopping cart information, and the features that the Table service provides to help you build a scalable, highly-available, and consistent key/value data store. These are critical properties that you should consider when implementing your own key/value store, regardless of which technology you use.

The blog post "How to get most out of Windows Azure Tables" provides more detailed information about best practices, performance, and scalability with the Windows Azure Table service.

More Information

All links in this book are accessible from the book's online bibliography available at: *http://msdn.microsoft.com/en-us/library/dn320459.aspx*.

- You can find information about Daytona on the Microsoft Research website at *http://research.microsoft.com/projects/daytona/default.aspx*.
- The article "How to use the Table Storage Service" is available on the Windows Azure developer website at *http://www.windowsazure.com/en-us/develop/net/how-to-guides/table-services/#header-4*.
- You can find the page "Understanding the Table Service Data Model" on MSDN at *http://msdn.microsoft.com/library/windowsazure/dd179338.aspx*.
- The article "Open Data Protocol by Example" is available on MSDN at *http://msdn.microsoft.com/library/ff478141.aspx*.
- You can find the article "Introducing Geo-replication for Windows Azure Storage" on MSDN at *http://blogs.msdn.com/b/windowsazurestorage/archive/2011/09/15/introducing-geo-replication-for-windows-azure-storage.aspx* .
- For more detailed information about how Windows Azure storage implements availability and consistency, you can download the white paper "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency" from MSDN at *http://blogs.msdn.com/b/windowsazurestorage/archive/2011/11/20/windows-azure-storage-a-highly-available-cloud-storage-service-with-strong-consistency.aspx*.
- You can find the article "How to get most out of Windows Azure Tables" on MSDN at *http://blogs.msdn.com/b/windowsazurestorage/archive/2010/11/06/how-to-get-most-out-of-windows-azure-tables.aspx*.

5 Implementing a Document Database

A key/value store provides arguably the fastest means of storing data. Additionally, as long as you only need to retrieve data by using the key, it is also the quickest mechanism amongst NoSQL databases. Together, these properties are ideally suited for applications that simply need to save and fetch large amounts of data extremely rapidly. However, for situations where an application needs to query or filter data based on attributes other than the key, then a key/value store becomes less useful. To support scenarios such as this, an application can save information to a document database instead.

At the most abstract level, a document database is similar in concept to a key/value store because it holds values that an application can search for by using a key. Indeed, you can use many document databases in exactly the same manner as a key/value store. However, the distinguishing feature of a document database compared to a key/value store is the transparency of the data that the database holds. You can define fields in documents, and applications can query the documents in a database by using these fields. Additionally, most document databases can create indexes over fields, and these indexes can help to optimize queries that reference these fields. This mechanism makes a document database more suitable than a key/value store for applications that need to retrieve data based on criteria more complex than the value of the document key. A document database can also support applications that require simple ad hoc querying capabilities, although the query features provided by most document databases are not as generalized or powerful as those available to relational databases through SQL.

It is sometimes difficult to determine whether a specific NoSQL database is a key/value store or a document database. For example, some key/value stores enable you to include metadata with values to enable the database to retrieve data by using attributes other than the key. The important point is not to worry unduly about the classification of a NoSQL database, rather you should ask yourself whether it provides the features that you require for your application and if so, how can you design a database that uses these features to their best effect.

This chapter describes the primary features of common document databases, and summarizes how you can design documents that take best advantage of these features to store and retrieve structured information quickly and efficiently. This chapter also discusses how Adventure Works used a document store to implement the product catalog and order history information for the Shopping application.

What is a Document?

A *document* in a document database is simply an entity that contains a collection of named fields. Don't confuse them with documents created by using Microsoft Word, or PDF files.

Different document databases use varying terms to describe the structure of the data that they hold. This chapter uses the term database *to refer to a document store; some vendors call them* domains*, or even* buckets*. Similarly, this chapter uses the term* collection *to refer to a group of documents holding similar data. For example, a database may hold a collection of orders documents and a separate collection of products documents. Finally, this chapter uses the term* field *rather than* attribute *or* key *to refer to elements within a document.*

The ability to define the structure of documents dynamically is a very flexible feature that enables you to adapt quickly to changing business requirements. In a relational database, you define the schemas for your data in advance (often expending considerable time and effort in the process). If the schema needs to change it is not always easy to apply these changes retrospectively to exist data in the database. In a document database, you simply include the fields that you require when you store a document.

If you are familiar with relational databases, you probably understand that before you can store data in a database you must first create the tables that will hold this data. These tables define the schema, and specify the fields (columns) that each data item (entity) contains. A document database is different. As described in Chapter 1, "Data Storage for Modern High-Performance Business Applications," most NoSQL databases are schema-less. What this means in practice is that the database simply acts as a data repository but does not impose any structure on that data because that is the responsibility of applications that store and maintain the data. This approach provides applications with an enormous amount of freedom because they can effectively store whatever they like. It also requires a great deal of discipline and consistency by application developers because these same applications need to be able to parse and process the data that they retrieve from the database, and therefore they need to understand the structure of the data. For this reason, it is common to group the documents into collections that have some semblance of commonality. Documents in a collection do not have to contain the same fields, but it is helpful if they model similar types of entities such as customers, orders, or products.

A document database requires a document to be self-describing, and most document databases store information in a well understood and portable format such as JSON, BSON, or XML. A few document databases restrict the fields in a document to simple string values. However, most support other common types such as numbers, dates, and Booleans, and many allow you to define complex structures in fields, enabling you to create nested documents.

Designing a Document Database

The structure of the data in a document database can have an important bearing on the performance of applications that use the database, as well as the throughput and scalability of the database itself. In particular, when you design document structures, you should consider the following factors:

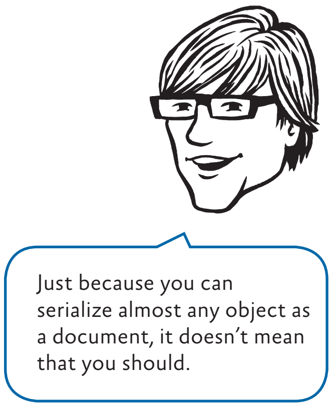
- How should you divide your data into documents and collections, and how far should you normalize or denormalize the data in a document?
- How are applications going query documents, and what predicates and filters are applications likely to use?
- How dynamic is the data; are your applications going to perform a large number of create, delete, and update operations?

The following sections discuss these factors in more detail and describe the choices that you can make.

Modeling Data as Documents

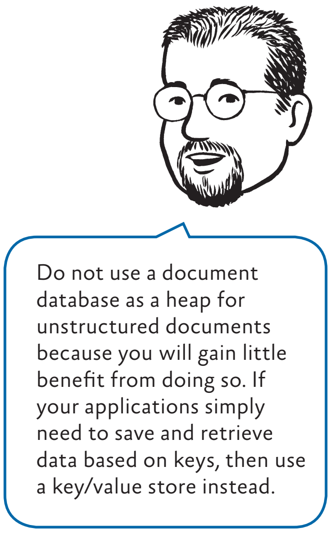
When you save data in a document database, initially it may seem tempting to simply serialize all of the objects that your application uses into the format expected by the database (JSON, BSON, XML, and so on) and then write them to the database. In a few scenarios where your applications process a very limited set of objects that are roughly all of the same type, this strategy might be appropriate. However, in the vast majority of cases this approach is likely to be unsatisfactory as it can lead to a vast amount of data duplication.

To make the best use of a document database, you should plan the structure of documents to optimize the queries that your applications perform and you should not treat the document database as a *document dump*.


Just because you can serialize almost any object as a document, it doesn't mean that you should.

Denormalizing Documents and Embedding Data

You should avoid attempting to design documents by following the relational database approach because although this strategy can eliminate duplicate data it will inevitably lead to lots of small documents. In this case, to reconstruct the data for an entity, your applications may need to retrieve data from multiple documents and combine them together. Relational databases are specifically designed to support and optimize generalized queries that can join data across tables whereas document databases are not. Most document databases do not provide the capability to join documents together in queries, so any processing of this type must be performed by the applications themselves. This may involve the application issuing multiple queries to the database.


Do not use a document database as a heap for unstructured documents because you will gain little benefit from doing so. If your applications simply need to save and retrieve data based on keys, then use a key/value store instead.

The general approach to designing documents is to examine how your application retrieves and processes data; use the business requirements of the application to mold the structure of the data and not vice versa. Ideally, the majority of queries that an application performs should be satisfied by a single visit to the database. This implies that you should denormalize your data, so that a document contains all the information required to display or manage the details of an entity, as described in Chapter 1. For example, if your application stores and retrieves information about customers, including their telephone numbers and addresses, you could use the conceptual structure shown in Figure 1.

Key (Customer ID)	Document		
99	Title:	Mr	
	FirstName:	Mark	
	LastName:	Hanson	
	Address:	StreetAddress:	999 500th Ave
		City:	Bellevue
		State:	WA
		ZipCode:	12345
	Telephone:	111-2223334	
		222-1112223	
		333-2221114	
100	Title:	Ms	
	FirstName:	Lisa	
	LastName:	Andrews	
	Address:	StreetAddress:	888 W. Front St
		City:	Boise
		State:	ID
		ZipCode:	54321
	Telephone:	555-4443332	
		444-5552223	
		333-5554442	

FIGURE 1
The conceptual structure of a document collection containing customer information

In JSON terms, the document structure for these two customers looks like the following code example. Notice that the **Address** field is a nested document, and the **Telephone** field contains a list of telephone numbers:

```
{"CustomerID":99,
 "Title":"Mr",
 "FirstName":"Mark",
 "LastName":"Hanson",
 "Address":{
    "StreetAddress":"999 500th Ave",
    "City”:"Bellevue",
    "State":"WA","ZipCode":"12345"
 },
 "Telephone":["111-2223334","222-1112223","333-2221114"]}

{"CustomerID":100,
 "Title":"Ms",
 "FirstName":"Lisa",
 "LastName":"Andrews",
 "Address":{
    "StreetAddress":"888 W. Front St",
    "City":"Boise",
    "State":"ID",
    "ZipCode":"54321"
},
"Telephone":["555-4443332","444-5552223","333-5554442"]}
```

Several document databases can generate a unique key value automatically when you create a new document and store it in the database. Others enable you to specify your own value for the key.

This structure is highly suitable for retrieving data based on the customer ID, and you can also perform queries that retrieve data based on other criteria such as the customer name. Remember that you can add indexes over fields in a collection to improve the performance of queries. For more information, see the section "Indexing Documents" later in this chapter.

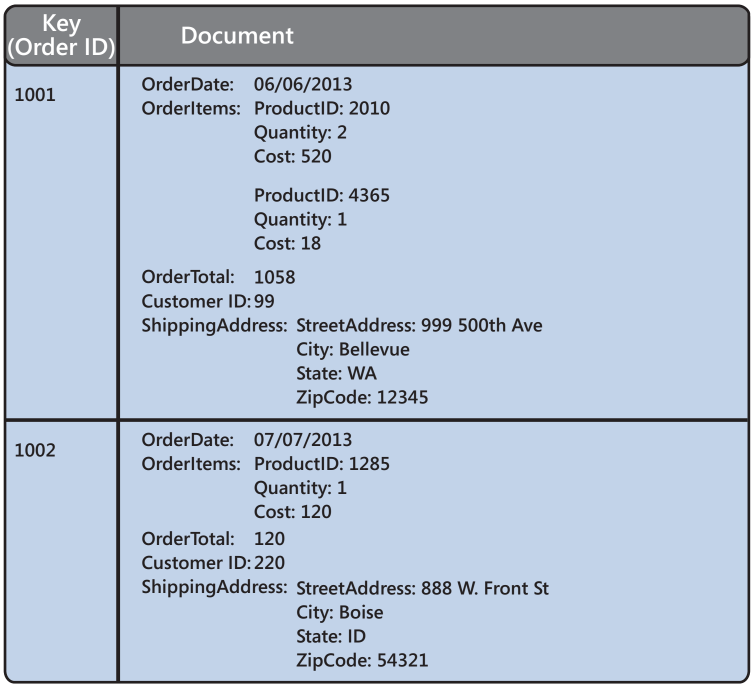
If your applications frequently need to retrieve data based on information stored in a field that contains a nested document, you might find it more beneficial to use a column-family database rather than a document database. For more information, see Chapter 6, "Implementing a Column-Family Database."

Be consistent in your application code. Make sure that you save documents by using the same names and data types for common fields, otherwise you will find it difficult to query the data later. For example, don't save the document for one customer where the customer ID is stored in a field called **ID**, and the document for another customer where the customer ID is stored in a field called **CustomerID**.

You should avoid making your documents too large and cumbersome; many document databases place limits on the sizes of documents, and hefty documents with a multitude of fields can be difficult to read and maintain. In the previous example, storing addresses and telephone numbers with customers is a natural fit, and the quantity of addresses and telephone numbers that a customer has at any one time are likely to be small.

Additionally, many document databases work best when the size of a document does not change (within reasonable parameters), but if repeated updates cause a document to grow, the resulting overhead of managing the space required can cause the performance of the database to decline. In the customers example, a single customer can have a variable number of telephone numbers, but this list is likely to be reasonably stable and not change very often for any given customer. Therefore it is practical to store this data as part of the customers' documents.

In many cases, you can consider a document to be an historical record where the data should not change after the document has been created. The sales order example containing a list of order items and the customer's shipping address described in Chapter 1 is one such scenario. Figure 2 shows the conceptual structure of a pair of sales order documents.



Key (Order ID)	Document
1001	OrderDate: 06/06/2013
	OrderItems: ProductID: 2010
	Quantity: 2
	Cost: 520
	ProductID: 4365
	Quantity: 1
	Cost: 18
	OrderTotal: 1058
	Customer ID: 99
	ShippingAddress: StreetAddress: 999 500th Ave
	City: Bellevue
	State: WA
	ZipCode: 12345
1002	OrderDate: 07/07/2013
	OrderItems: ProductID: 1285
	Quantity: 1
	Cost: 120
	OrderTotal: 120
	Customer ID: 220
	ShippingAddress: StreetAddress: 888 W. Front St
	City: Boise
	State: ID
	ZipCode: 54321

FIGURE 2
The conceptual structure of a document collection containing sales order information

The JSON definition of order 1001 in Figure 2 might look like this:

```
{"OrderID":1001,
 "OrderDate":"\/Date(1370473200000+0100)\/",
 "OrderItems":[{
      "ProductID":2010,
      "Quantity":2,
      "Cost":520
     },
     {
      "ProductID":4365,
      "Quantity":1,
      "Cost":18
     }],
 "OrderTotal":1058,
 "CustomerID":99,
 "ShippingAddress":{
     "StreetAddress":"999 500th Ave",
     "City":"Bellevue",
     "State":"WA",
     "ZipCode":"12345"}}
```

In this example, the shipping address will be duplicated if the same customer places several orders, but the shipping address should indicate the destination of the order regardless of whether the customer subsequently moves to a different location, so this duplication is necessary to ensure the historical accuracy of each sales order document.

Normalizing Documents and Referencing Data

Denormalizing data can help to optimize queries by reducing the number of documents that an application has to query to fetch data. However, it can have some detrimental side effects, and in some circumstances it may be more appropriate to normalize data.

Figure 2 illustrates an example of the normalized approach. Each sales order specifies the ID of the customer that placed the order, but does not include any detailed information about the customer. You could argue that if the applications frequently retrieve information about orders and the details of the customers that placed them, then the customer details should be embedded directly into the sales order documents. However, if the application regularly queries the details of customers and not their orders, the logic for performing these queries can become contorted and it does not feel natural that an application should have to iterate through sales orders to obtain a list of customers. Additionally, it is very likely that a single customer will (hopefully) place many orders, leading to a high degree of duplication and making any updates to customer details difficult and time consuming to perform (unlike shipping addresses, the details of customers may be dynamic). Therefore, in this example, it is better to maintain sales orders and customers as separate collections. The same logic applies to products; each order item contains the product ID rather than an embedded document that describes the product.

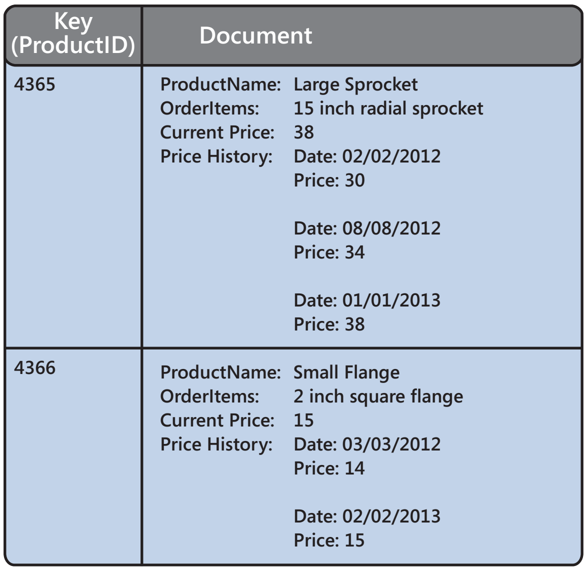
The sales order example references other documents (for customers and products) by using the keys for these items. A few document databases implement reference types that provide a direct link to a related document, enabling them to be retrieved more quickly by applications which do not have to perform a search for the related document.

Other document databases may cache document keys in memory to provide fast access to documents, although this approach may require a large amount of RAM if your database contains tens or hundreds of millions of documents all of which are being accessed frequently (this situation is not uncommon in vary large systems).

Normalizing your documents can have some beneficial effects on performance if you have mutating data. As described in the previous section, you should avoid creating documents that can grow indefinitely, even if the rate of growth is small. This is because many document databases will have to move the data around on disk (or in memory) to make space as the document is updated, reducing the performance of the database. Some document databases enable you to pre-allocate additional space to documents, enabling them to grow without being relocated, but once you exhaust this space the document will have to be moved.

Many document databases support in-place updates as long as the document does not increase in size, but if a modification causes a document to grow the database might need to rewrite it to a different location. If possible, avoid creating documents that exhibit this behavior, and especially avoid creating documents that contain unbounded, growing, mutable arrays.

As an example, your application might need to store information about the products that your organization sells. If the application additionally needs to maintain a price history for each, you might define product documents with the structure shown in Figure 3, where the price history is maintained as a field within the product document.



Key (ProductID)	Document
4365	ProductName: Large Sprocket OrderItems: 15 inch radial sprocket Current Price: 38 Price History: Date: 02/02/2012 Price: 30 Date: 08/08/2012 Price: 34 Date: 01/01/2013 Price: 38
4366	ProductName: Small Flange OrderItems: 2 inch square flange Current Price: 15 Price History: Date: 03/03/2012 Price: 14 Date: 02/02/2013 Price: 15

Figure 3
Storing price history information

As prices change, each product document will increase in size. In many business scenarios, price changes tend to occur across an entire product range at the same time (for example, in the Adventure Works scenario, the prices of all bicycles might increase if the cost of manufacturing bicycles rises, or decrease if the company decides to have a sale.) The effects of a large number of documents being written and relocated rather than updated in place could have a detrimental effect on performance, although in this case these updates are likely to happen relatively infrequently.

In a different scenario that implements a similar structure, if you are writing applications that capture data measured by a series of scientific instruments and logging the measurements for a single instrument with information about that instrument in the same document, the cost of rewriting the document for each data point could become prohibitive, especially if your application captures measurements from a large number of different instruments. Figure 4 shows the information that such a document might contain if the instrument is a thermometer capturing the ambient temperature of the environment at 1 minute intervals.

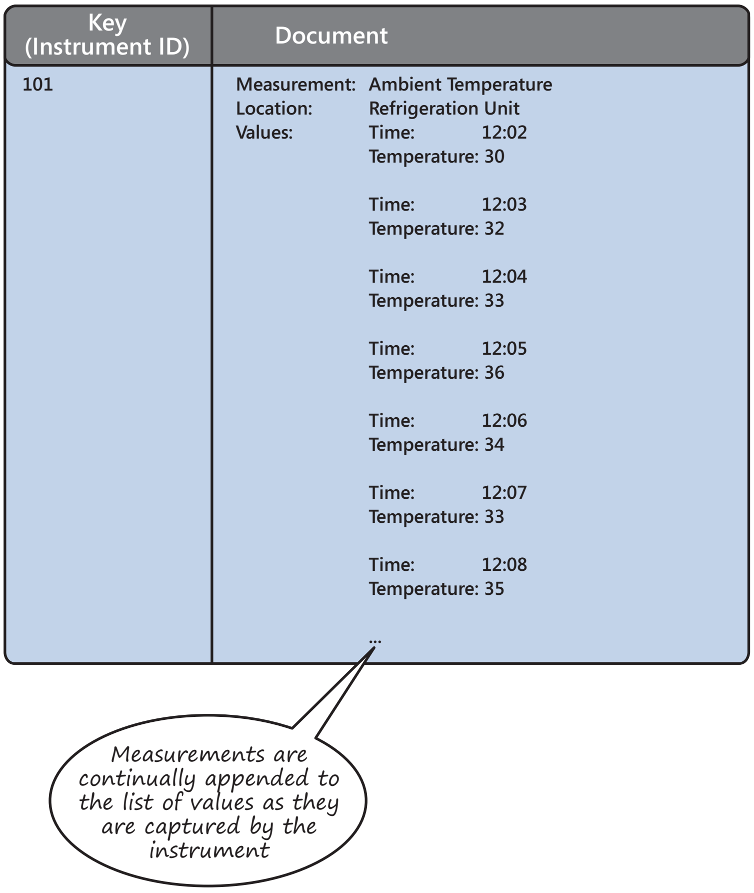
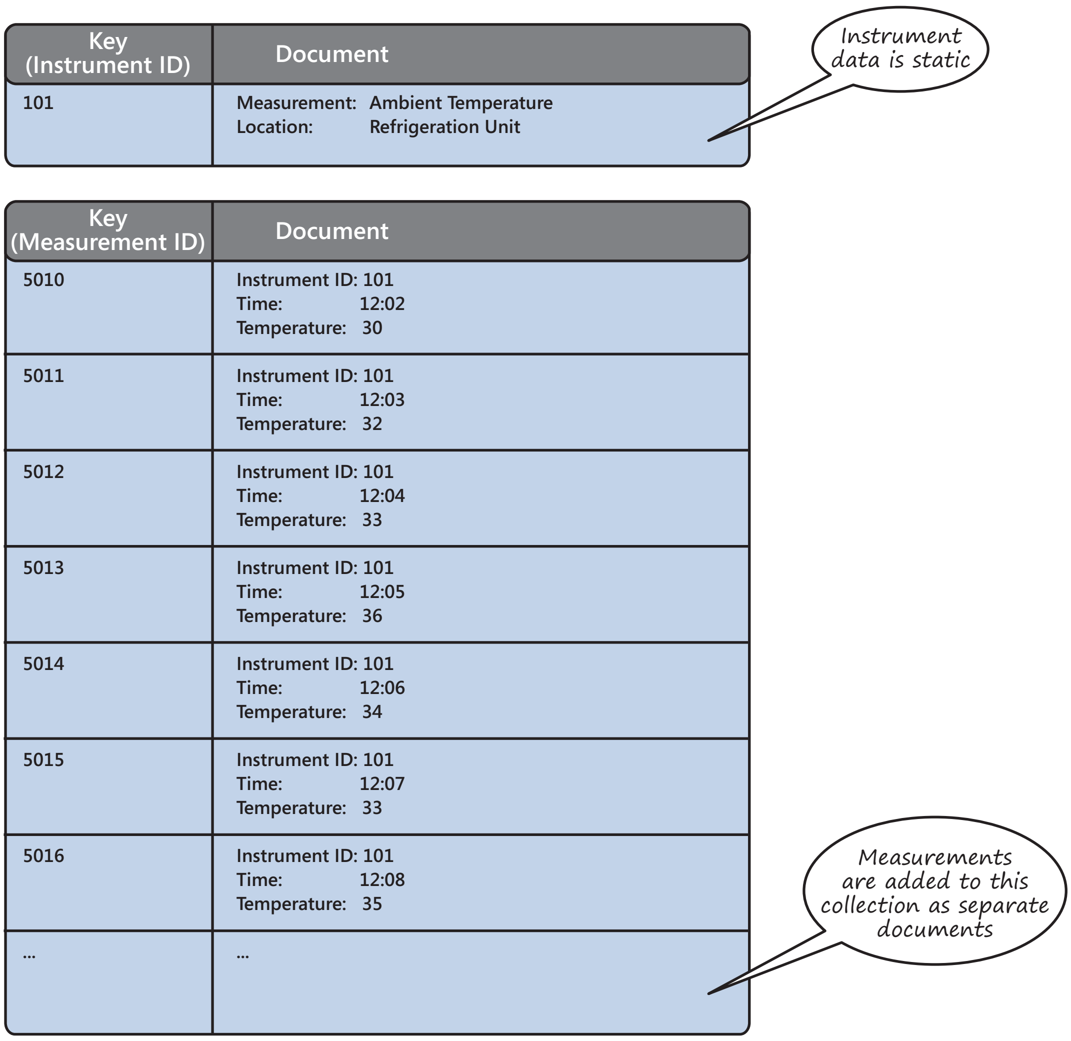

Key (Instrument ID)
Document
101
Measurement: Ambient Temperature
Location: Refrigeration Unit
Values: Time: 12:02 Temperature: 30
Time: 12:03 Temperature: 32
Time: 12:04 Temperature: 33
Time: 12:05 Temperature: 36
Time: 12:06 Temperature: 34
Time: 12:07 Temperature: 33
Time: 12:08 Temperature: 35
...
Measurements are continually appended to the list of values as they are captured by the instrument

Figure 4
Appending measurements to an array in a document

In this case, you should implement scientific measurements as separate documents that reference the instrument document, as shown in Figure 5. Each of these measurements is a fixed size and will not change once they have been saved. Most document databases are optimized to handle precisely this type of behavior.

Alternatively, you could store the information in a column-family database where the time indicates the name of a column and the temperature is the value of that column. Chapter 6, "Implementing a Column-Family Database" provides more information.



Key (Instrument ID)	Document
101	Measurement: Ambient Temperature Location: Refrigeration Unit

Key (Measurement ID)	Document
5010	Instrument ID: 101 Time: 12:02 Temperature: 30
5011	Instrument ID: 101 Time: 12:03 Temperature: 32
5012	Instrument ID: 101 Time: 12:04 Temperature: 33
5013	Instrument ID: 101 Time: 12:05 Temperature: 36
5014	Instrument ID: 101 Time: 12:06 Temperature: 34
5015	Instrument ID: 101 Time: 12:07 Temperature: 33
5016	Instrument ID: 101 Time: 12:08 Temperature: 35
...	...

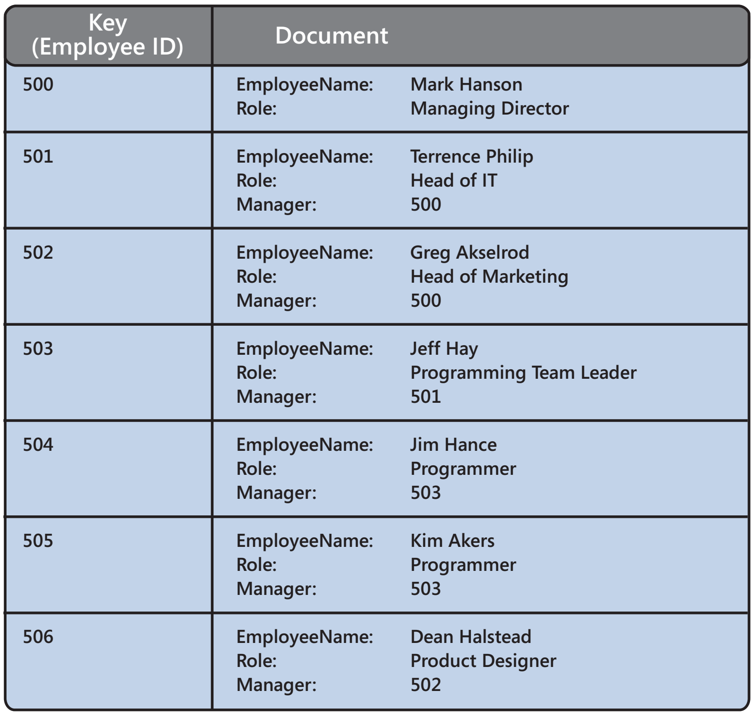
Instrument data is static
Measurements are added to this collection as separate documents

FIGURE 5
Storing measurements as documents in a separate collection

In a normalized relational database, there is often no distinction made between the structure of the various one-to-many (1:n) relationships that can occur, and you nearly always implement the entities that participate in these relationships as separate tables. When you design a document database, you need to understand that there are different types of 1:n relationships. If *n* is small and does not vary much, then you can consider denormalizing and combining both entities into the same document. If *n* is large or can change considerably over time, then implement both entities as separate documents.

Handling Complex Relationships

You can model complex relationships in a document database, such as hierarchical data and tree structures, by using references. Figure 6 shows an example that records which employees in an organization report to which managers.



Key (Employee ID)	Document	
500	EmployeeName: Role:	Mark Hanson Managing Director
501	EmployeeName: Role: Manager:	Terrence Philip Head of IT 500
502	EmployeeName: Role: Manager:	Greg Akselrod Head of Marketing 500
503	EmployeeName: Role: Manager:	Jeff Hay Programming Team Leader 501
504	EmployeeName: Role: Manager:	Jim Hance Programmer 503
505	EmployeeName: Role: Manager:	Kim Akers Programmer 503
506	EmployeeName: Role: Manager:	Dean Halstead Product Designer 502

Figure 6
Modeling hierarchical data in a collection of documents

This collection is not much different from a relational table that implements the same functionality, and searching data to find a list of employees that directly or indirectly report to a specific manager requires an application to perform several queries over the same collection. If a hierarchy is relatively static, you could consider including materialized paths in the data. A materialized path records the path from a document to the top of the hierarchy. Figure 7 shows the employee data from Figure 6 with the **Manager** field replaced by the **ManagerPath** field that contains a materialized path that specifies the manager, the manager's manager, and so on. An application searching for the employees that report to a specific manager can query the **ManagerPath** field to find this information.

> *The order of data in a materialized path is significant, and you may need to store the materialized path as a string rather than a list of employee IDs. This is because some document databases do not preserve the order of data in list fields.*

Key (Employee ID)	Document	
500	EmployeeName: Role:	Mark Hanson Managing Director
501	EmployeeName: Role: ManagerPath:	Terrence Philip Head of IT 500
502	EmployeeName: Role: ManagerPath:	Greg Akselrod Head of Marketing 500
503	EmployeeName: Role: ManagerPath:	Jeff Hay Programming Team Leader 501, 500
504	EmployeeName: Role: ManagerPath:	Jim Hance Programmer 503, 501, 500
505	EmployeeName: Role: ManagerPath:	Kim Akers Programmer 503, 501, 500
506	EmployeeName: Role: ManagerPath:	Dean Halstead Product Designer 502, 500

FIGURE 7
Storing hierarchy information by using materialized paths

You can follow a similar approach for other complex relationships. However, maintaining materialized paths can be resource intensive and time consuming, so only adopt this approach if the relationships do not change frequently.

If you need to store information about dynamic relationships, consider using a graph database. If necessary, you can integrate a graph database and a document database together. Use the graph database to store information about relationships between documents, and add references to documents in the document database to the nodes in the graph database. For more information, see Chapter 8, "Building a Polyglot Solution."

Retrieving Data and Indexing Documents

The main reason for using a document database is to enable applications to quickly find data based on the values held in one or more fields. The fastest form of data retrieval is to use the document key. However, in many cases, simply retrieving data by using the key will not be sufficient (if it is, you should use a key/value store rather than a document database). For this reason, document databases enable you to specify query criteria that identify documents based on the non-key fields in your documents.

The actual query features available depend on the specific document database that you are using. All document databases enable an application to filter data by testing whether the data in a field is equal to a specified value, and some document databases allow an application to specify a variety of comparison operators such as *less than*, *greater than*, and so on. In some cases, you can combine criteria together with *and* and *or* operators. The majority of document databases support projections, enabling an application to specify which fields a query should retrieve, but some document databases can only retrieve the entire document. Additionally, while the various document databases currently available provide their own APIs and support applications written in a variety of different programming languages, most of them also support the REST model of working, enabling your applications to submit queries as REST requests, and allowing your applications to page through data if a query returns a large number of documents.

Summarizing Data by Using Materialized Views

Some document databases support materialized views, enabling an application to quickly retrieve summary information from documents or pose queries that require calculating values across a collection of documents. For example, if your database stores information about sales orders as shown in Figure 2 earlier in this chapter, and your application needs to calculate the average value of each order that includes a specific product, then using a materialized view could be more efficient than searching for all sales orders that contain the product, retrieving these orders across the network, and then performing the calculation by using logic in the application. Figure 8 illustrates the concepts behind a materialized view.

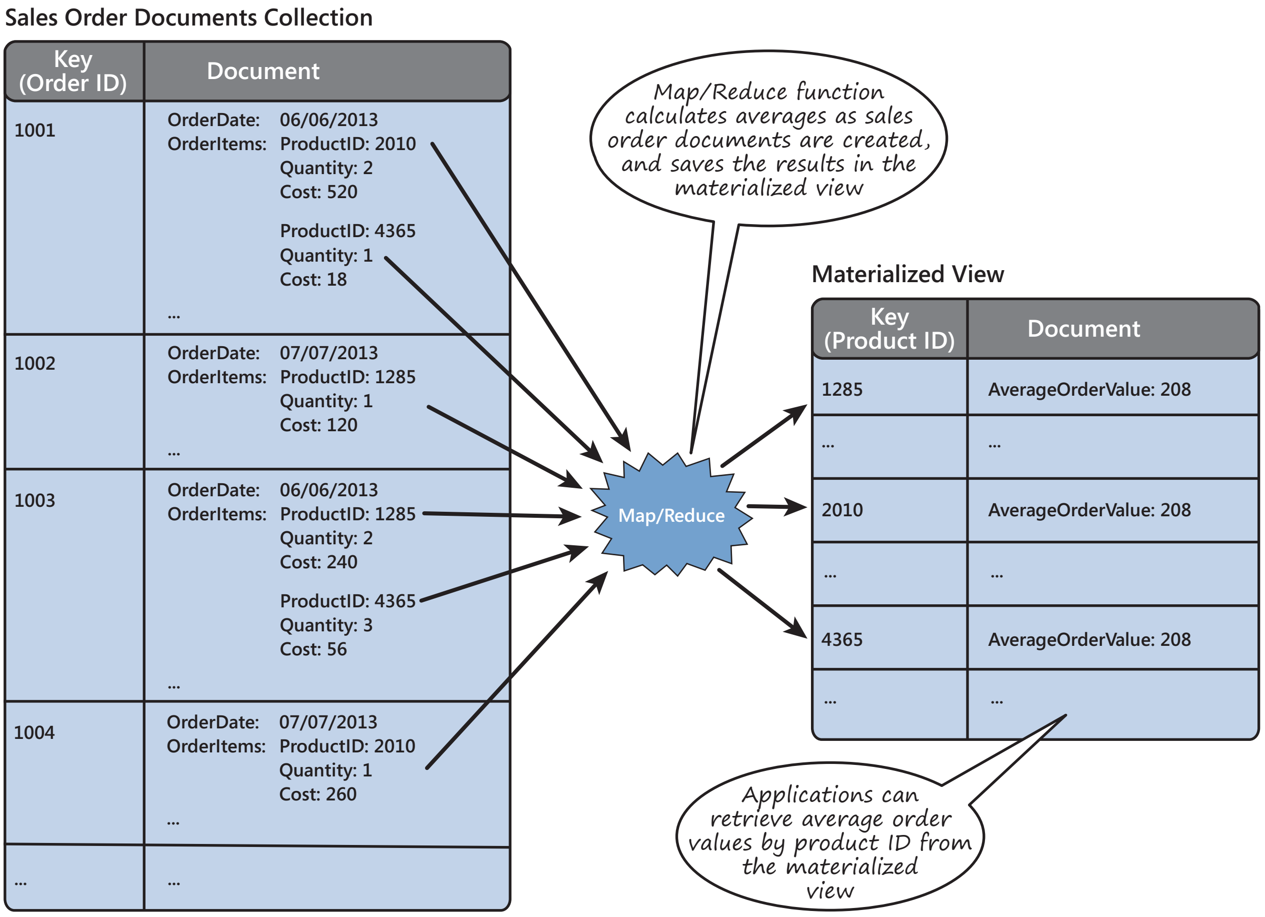


Figure 8
Using a materialized view to calculate and store summary data

Maintaining data in a materialized view will have an impact on the performance of insert, update, and delete operations. You need to balance the frequency of queries that benefit from using a materialized view against the volume of inserts, updates, and deletes that can occur.

Most document databases implement materialized views by using a map/reduce framework. They maintain the information held by the view as documents are added, updated, and deleted. Therefore, you should assess the cost of keeping a materialized view up to date against the performance benefits gained when you query this data.

Indexing Documents

When an application retrieves documents by specifying criteria based on fields other than the key, the document database may need to scan through every document in a collection to find matches. If the collection is large, this process can take some time. Most document databases enable you to define indexes over fields to help speed this process up. An index contains a sorted list of values in a given field (or fields), together with direct references to documents that have these values in the specified fields. When an application queries a collection with criteria that include one or more indexed fields, the database management system can quickly find matching documents by looking them up in the appropriate indexes. Figure 9 shows an example index created over the **City** field in the nested **Address** document in the **Customers** document collection. An application that needs to find customers located in a specific city will benefit from using this index.

Not all document databases enable you to create indexes over fields in nested documents, or fields that contain lists.

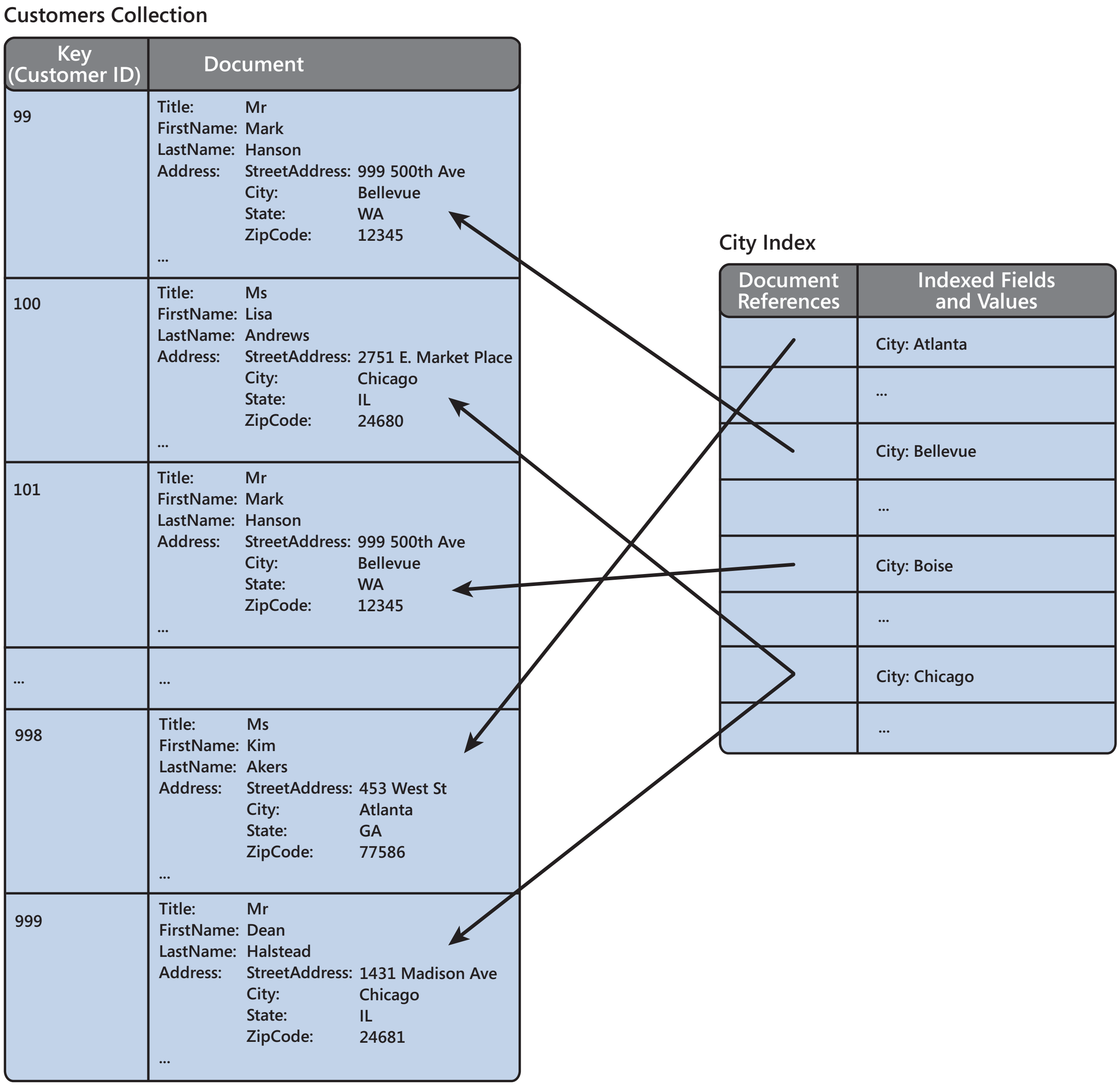


Figure 9
The Customers collection with an index over the City field

> *A few document databases only support queries that either reference the key or indexed fields. In these databases, non-indexed fields cannot be specified as filter criteria.*

The techniques that different document databases use to implement indexes varies. Some databases use balanced B-Tree structures while others implement indexes by creating materialized views with the data ordered by the index key. Some document databases provide support for pluggable third-party indexing libraries, enabling developers to select an indexing implementation that is most closely suited and optimized to the type of data that is being indexed. However they work, all indexes impose an overhead on the database because they have to be maintained as documents are added, modified, and deleted.

If the documents in a collection are subjected to a high proportion of queries compared to inserts, updates, and deletes, then you should consider creating indexes to optimize as many of these queries as possible. If a collection has a high proportion of write operations compared to queries, then you may need to reduce the number of indexes to a minimum. You should also bear in mind that indexes require space on disk (or in memory); the more indexes you create, the more space they will occupy in the database.

> *Some document databases maintain indexes by using a separate low-priority thread running in the background. This approach helps to minimize the overhead associated with an index, although it is possible that the data in an index may be out of date when a query that utilizes that index runs. This may mean that a query may fail to find a document that would be retrieved if the index was not used.*

The primary use of indexes is to support fast searching and filtering of data, but if an application regularly needs to fetch data in a specific order, then indexes can help here as well. The fields in an index are sorted, so an application that requires documents to be ordered by a particular field can benefit if that field is indexed. In the example shown in Figure 9, an application that needs to retrieve and display all customers in order of the city in which they are located can make use of the **City** index.

Several document databases allow indexes to span multiple fields; you specify the relative significance of fields, and the index will be sorted by the most significant field first, then by the next most significant field, and so on. In some cases, if an index contains all the fields required by a query, then there is no need for the database management system to actually retrieve the document itself. This type of index is known as a *covering index*, and they can be very effective at speeding queries in document databases that support projections.

In a static or query-intensive collection, consider creating indexes to optimize as many queries that your applications perform as possible, and define covering indexes where appropriate. In a dynamic collection, only create the indexes that are absolutely essential, to reduce the overhead of maintaining them.

The decision of whether a query submitted by an application to a document databases uses an index or not can depend on many factors. In some cases, the application itself indicates the index to use, while in others the database management system makes the selection. Some document databases implement sophisticated query optimization strategies that rival those found in many relational database management systems. A popular document database even considers the cost of dynamically creating an index over a field if no such index exists and it would prove useful to the query being run. This index is temporary but remains in existence for a short time. If the same (or similar) query is repeated, it will use the temporary index, and if this temporary index is used enough within a given period it will be made permanent.

Creating, Updating, and Deleting Documents

For reasons of speed and scalability, most document databases provide only a very basic programmatic interface for creating, updating, and deleting documents. To create a new document you simply encode the data in the appropriate format (JSON, BSON, XML, or whatever the document database expects) and then add it to a collection in the database. There are no defined schemas in a document database, and as long as the document is well formed and has a valid structure the database will store it.

To delete a document, your application typically identifies the document either by its key value or by specifying other query criteria to match against fields in the document. The application then sends a request to the database to remove this document. Some document databases provide a *search and delete* API, enabling your application to combine these steps into a single atomic operation. Depending on the database, the document might not be physically removed at this point but might instead be marked as *for deletion* (it will not be retrieved by any subsequent queries, however.) At some point in the future, the space used by this document will be reclaimed.

It is important to remember that unlike relational systems, document databases tend not to implement any form of relational integrity. This means that your applications can easily delete a document that is referenced by another document, and it is the responsibility of the application code to ensure that any relationships between documents remain valid and intact.

Many document databases can remove multiple documents in the same request if they all match the specified criteria. There is no *"Are you sure?"* confirmation, and it is the responsibility of your application code to make sure that only the intended documents are deleted.

Document databases frequently provide two types of update operation, as follows.

- In the first type, the application identifies the document (or documents) to modify by following a similar process to that for deleting documents. The application also provides a collection of field and value pairs, and the data in these fields in the document are updated with the new values.
- In the second type, the application submits an entire document, including the key, to the database. If a document with this key already exists it is replaced with the new document. If the document does not already exist it is added to the database. This is an *upsert* operation.

  Remember that if a document increases in size, the database may need to relocate it. This can take a little time and will slow down the update operation.

Some document databases enable you to batch multiple updates together and send them to the database as a single request. However, most document databases do not support transactions in the same way as relational systems. In most cases you can assume that operations that write to a single document are atomic (the database will not insert half a document, for example), but operations that span multiple documents are not. This means that if, for example, your application submits a request to update 10 documents, you cannot guarantee that all 10 documents will be modified at the same time and some inconsistencies can creep in, especially if these documents are heavily used.

In most document databases, only insert, update, and delete operations that affect a single document are atomic. This will have a bearing on the design of your documents and your applications.

To counter possible problems arising from inconsistency, some document databases enable an application to lock documents. The lock will prevent concurrent applications from modifying these documents, although they may be able to read locked data. It is the responsibility of the application to ensure that locks are released in a timely manner when the updates are complete.

Some document databases perform insert, update, and delete operations asynchronously. This means that you cannot guarantee when these operation will actually be completed, and this approach can lead to inconsistences occurring if developers are not aware that this is happening.

Pessimistic locking can severely impact the performance of the system, possibly resulting in extended response times by concurrent applications sharing the same database. It is better to design your applications assuming that documents may change while your applications are using them than to lock documents for exclusive access.

If this type of pessimistic locking is inappropriate to your solution, you can consider implementing an optimistic strategy instead. Some document databases provide *check and set* operations to help with this approach. In this case, when the data is updated, the database first verifies that it has not changed since the application retrieved it, passing back an error indication and if it has. The application should check for this error, and retrieve the latest version of the document before attempting to update it again.

How Adventure Works Used a Document Database to Store the Product Catalog and Order History Information

As described in Chapter 2, "The Adventure Works Scenario," when a customer starts the Shopping application it shows a list of product categories. The customer can click a category and the Shopping application then displays a list of subcategories within that category. If the customer clicks a subcategory, the Shopping application presents the products within that subcategory. The customer can then click a product to view its details. You can see from this description that the product catalog constitutes a three-level hierarchy, with categories at the top, subcategories in the middle, and products at the bottom. The application has to be able to quickly find and display a list of items at each level in this hierarchy. However, the category, subcategory, and product information rarely changes (with the possible exception of product pricing), and as far as the Shopping application is concerned this data is all read-only. The category, subcategory, and product information are good candidates for restructuring as product catalog in a document database.

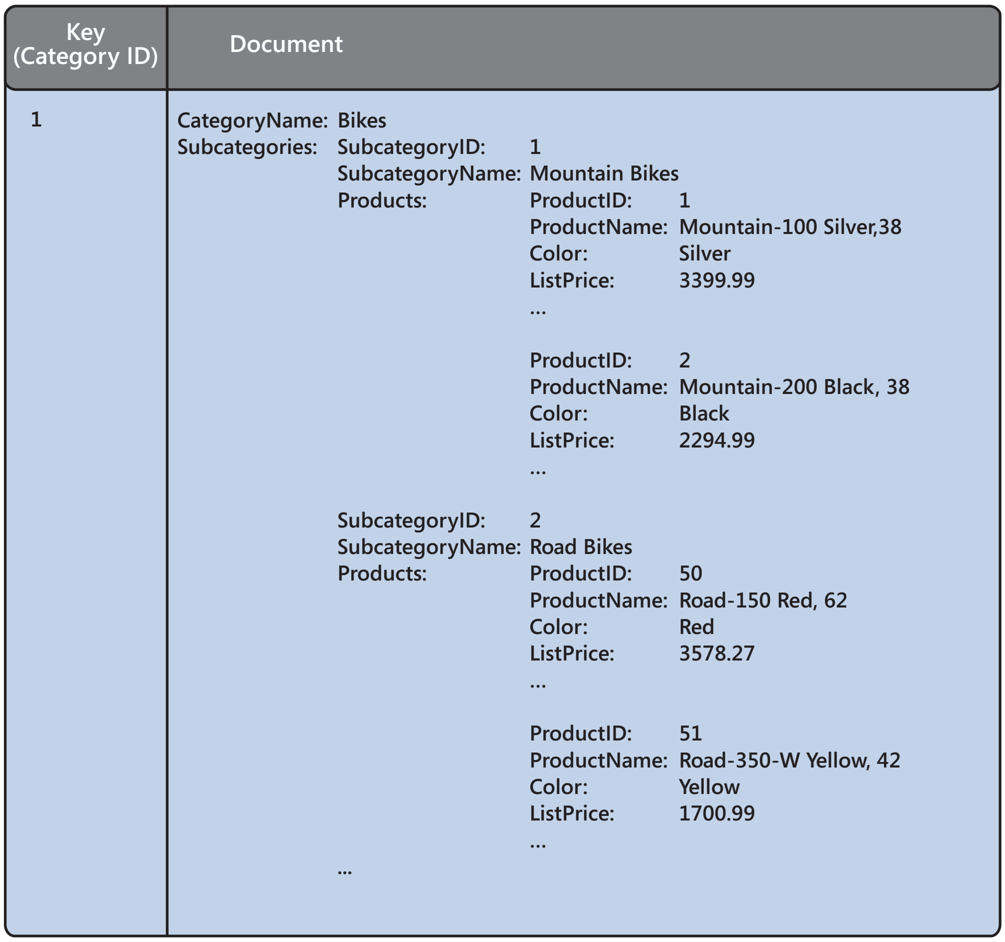
Refer to Chapter 2, "The Adventure Works Scenario" for a more detailed description of the features of the Shopping application.

When a customer places an order, the Shopping application stores the details of the order in a SQL Server database (Chapter 3, "Implementing a Relational Database" describes this part of the process in more detail). The warehousing and shipping systems inside Adventure Works retrieve and modify the status of an order as items are picked and dispatched (these systems are OLTP-based services and are outside the scope of the Shopping application). The Shopping application enables a customer to view the details of the orders that they have placed, and it would be possible to retrieve this information from the SQL Server database. However, once an order has been fulfilled it should be considered a historical record. If any details of an order change, the system should maintain a full audit trail to enable these changes to be tracked. The volume of historical order information in the database will increase as customers place orders, and it is anticipated that the database could easily be expected to hold hundreds of thousands of orders by the end of the first year of operation, increasing to millions of orders in the years after that. The database must be scalable to store this amount of data, and also be quick to find the historical details of the orders for any customer. Therefore, the developers chose to make a copy of each order as it is created and each time it is updated, and store these copies in a separate database, optimized for fast query access. The developers considered that a document database might provide better scalability and performance for holding this order history information than SQL Server.

Designing the Documents for the Product Catalog

The developers at Adventure Works considered several options for structuring the documents for the product catalog. First, they thought about adopting a completely denormalized approach, where the category information includes a list of subcategories, which in turn contain a list of products. Figure 10 shows an example of a **Category** document. Note that subcategories are not shared across categories, and each product can only be in a single subcategory, so there is no duplication of data. Additionally, it is possible to create indexes over the **SubcategoryID** and **ProductID** fields, enabling an application to quickly locate the details of a document containing a specified subcategory or product. However, each document is potentially quite large (possibly exceeding the document size limitations for many document databases). Additionally, the unit of I/O for most document databases is the document, meaning that queries that needed to find the details of a single product may end up reading the entire document containing that product into memory. This is potentially a very expensive overhead.

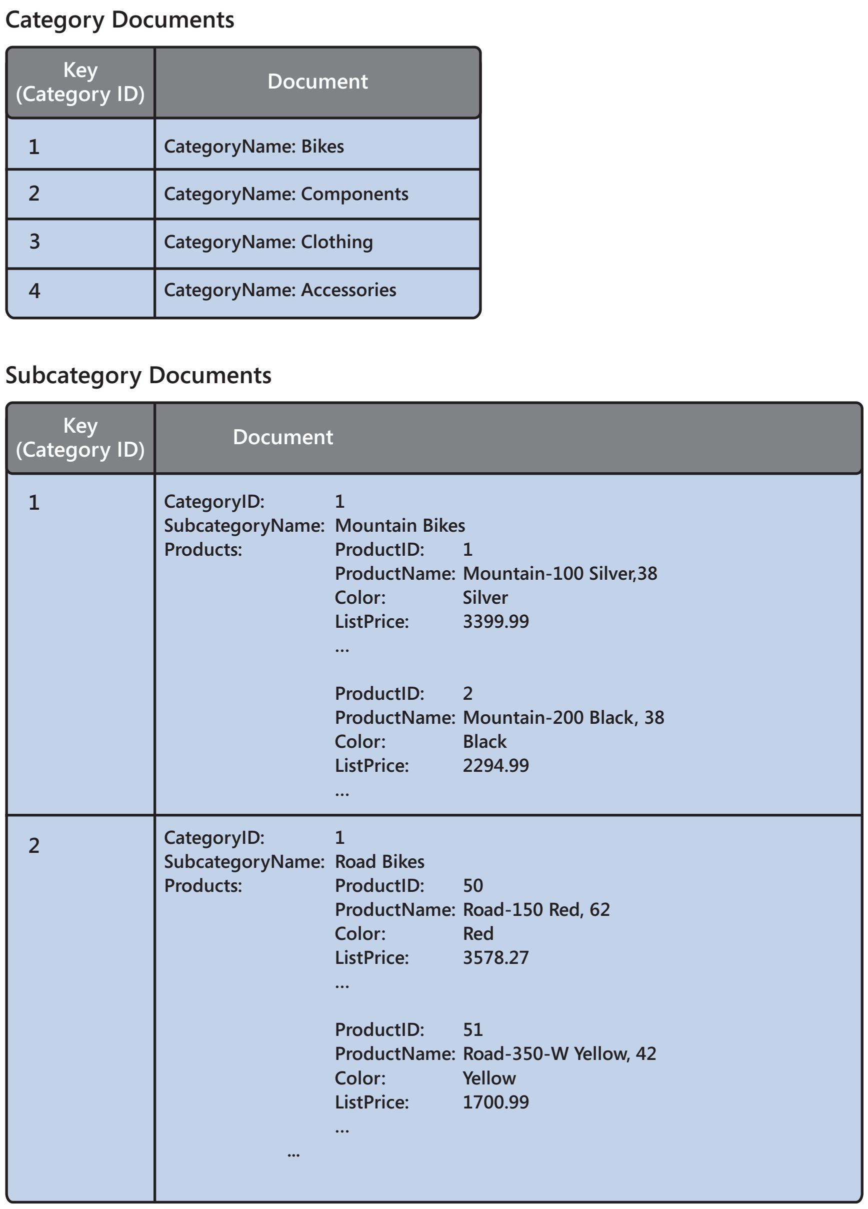
Some document databases support projections, enabling a query to return only the specified fields from a document. However, the database server may still need to read the entire document from the database to extract this information.



```
Key
(Category ID)  Document

1              CategoryName:  Bikes
               Subcategories: SubcategoryID:   1
                              SubcategoryName: Mountain Bikes
                              Products:        ProductID:   1
                                               ProductName: Mountain-100 Silver,38
                                               Color:       Silver
                                               ListPrice:   3399.99
                                               ...

                                               ProductID:   2
                                               ProductName: Mountain-200 Black, 38
                                               Color:       Black
                                               ListPrice:   2294.99
                                               ...

                              SubcategoryID:   2
                              SubcategoryName: Road Bikes
                              Products:        ProductID:   50
                                               ProductName: Road-150 Red, 62
                                               Color:       Red
                                               ListPrice:   3578.27
                                               ...

                                               ProductID:   51
                                               ProductName: Road-350-W Yellow, 42
                                               Color:       Yellow
                                               ListPrice:   1700.99
                                               ...
                              ...
```

Figure 10
The structure of a denormalized document containing category, subcategory, and product information

A second option was to partially normalize the data; split category and subcategory information into separate document collections, and store product details with subcategories as shown in Figure 11. Again, although this approach reduces the size of the **Category** documents, the **Subcategory** documents could still exceed the document size limitations of many document databases and impose an unnecessary overhead on queries that retrieve the details of individual products so it was also discarded.



Category Documents

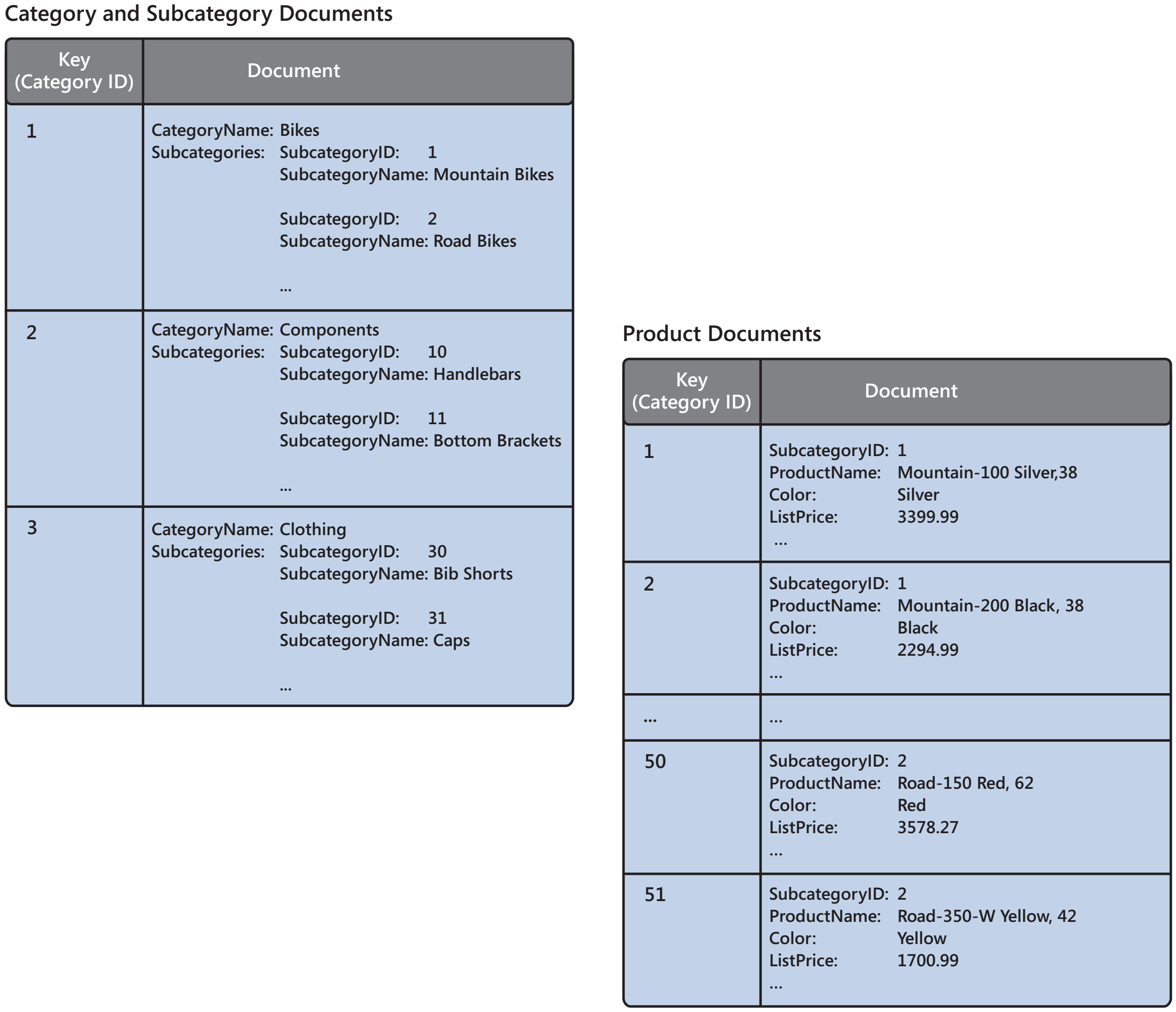
Key (Category ID)	Document
1	CategoryName: Bikes
2	CategoryName: Components
3	CategoryName: Clothing
4	CategoryName: Accessories

Subcategory Documents

Key (Category ID)	Document
1	CategoryID: 1 SubcategoryName: Mountain Bikes Products: ProductID: 1 ProductName: Mountain-100 Silver,38 Color: Silver ListPrice: 3399.99 ... ProductID: 2 ProductName: Mountain-200 Black, 38 Color: Black ListPrice: 2294.99 ...
2	CategoryID: 1 SubcategoryName: Road Bikes Products: ProductID: 50 ProductName: Road-150 Red, 62 Color: Red ListPrice: 3578.27 ... ProductID: 51 ProductName: Road-350-W Yellow, 42 Color: Yellow ListPrice: 1700.99 ... ...

FIGURE 11
Normalized category documents with denormalized documents containing subcategory and product information

A further option was to completely normalize the data and create separate collections for **Category** documents, **Subcategory** documents, and **Product** documents. However, to optimize the retrieval of categories and subcategories by the Shopping application, the developers chose to combine these **Category** and **Subcategory** documents together, resulting in the document structure illustrated by Figure 12. The relatively small number of subcategories in a category makes it unlikely that the size of each document will exceed the capabilities of the document database (the **Components** category is the largest, with 14 subcategories). Additionally, creating an index over the **SubcategoryID** field in the **Product** documents can speed retrieval of all products in a given subcategory.


Category and Subcategory Documents

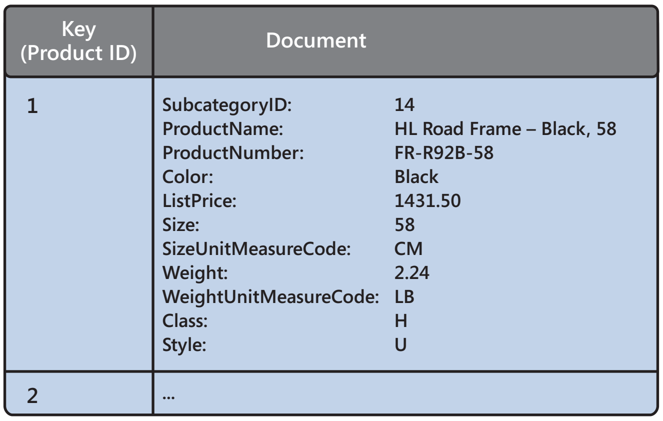
Key (Category ID)	Document
1	CategoryName: Bikes Subcategories: SubcategoryID: 1 SubcategoryName: Mountain Bikes SubcategoryID: 2 SubcategoryName: Road Bikes ...
2	CategoryName: Components Subcategories: SubcategoryID: 10 SubcategoryName: Handlebars SubcategoryID: 11 SubcategoryName: Bottom Brackets ...
3	CategoryName: Clothing Subcategories: SubcategoryID: 30 SubcategoryName: Bib Shorts SubcategoryID: 31 SubcategoryName: Caps ...

Product Documents

Key (Category ID)	Document
1	SubcategoryID: 1 ProductName: Mountain-100 Silver,38 Color: Silver ListPrice: 3399.99 ...
2	SubcategoryID: 1 ProductName: Mountain-200 Black, 38 Color: Black ListPrice: 2294.99 ...
...	...
50	SubcategoryID: 2 ProductName: Road-150 Red, 62 Color: Red ListPrice: 3578.27 ...
51	SubcategoryID: 2 ProductName: Road-350-W Yellow, 42 Color: Yellow ListPrice: 1700.99 ...

Figure 12
Denormalized documents containing category and subcategory information, and normalized product documents

Having decided on the high-level structure of the documents, the developers looked more closely at the information held for each product and the possible queries that the application has to support. The complete data for a typical product is shown in Figure 13:



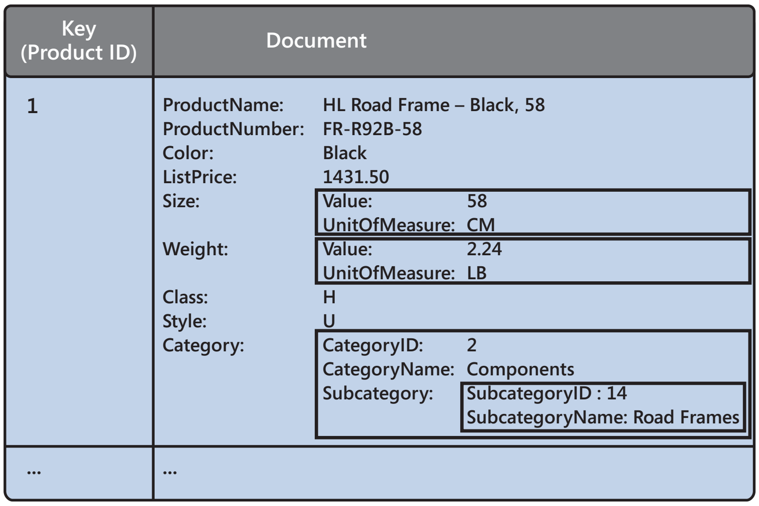
Key (Product ID)	Document	
1	SubcategoryID:	14
	ProductName:	HL Road Frame – Black, 58
	ProductNumber:	FR-R92B-58
	Color:	Black
	ListPrice:	1431.50
	Size:	58
	SizeUnitMeasureCode:	CM
	Weight:	2.24
	WeightUnitMeasureCode:	LB
	Class:	H
	Style:	U
2	...	

FIGURE 13
The complete details of a product

After due consideration, the developers uncovered two issues:

- Although it is not a feature used by the initial version of the Shopping application, the developers felt that it was likely that in a future release, queries that retrieved product information would also need to fetch information about the category that contained the product. The product document structure defined in Figure 12 only lists the subcategory ID, so to retrieve category information an application has to find the category document that contains this subcategory. Therefore, the developers decided to extend the design of the product document structure to include category information as a nested document. The subcategory details are nested inside this document.
- Products have size and weight properties, but these properties are actually pairs; the **Size** property is simply a number and the **SizeUnitMeasureCode** property specifies the units (different products have different units of size–centimeters for wheels, for example). Similarly, the value in the **Weight** property is only meaningful in conjunction with the **WeightUnitMeasureCode** property. The developers decided to implement these properties as small subdocuments that combine the information, to indicate that they are only meaningful when read together.

Figure 14 shows the refined structure of a product document (the subdocuments are highlighted):



Key (Product ID)	Document
1	ProductName: HL Road Frame – Black, 58 ProductNumber: FR-R92B-58 Color: Black ListPrice: 1431.50 Size: [Value: 58 / UnitOfMeasure: CM] Weight: [Value: 2.24 / UnitOfMeasure: LB] Class: H Style: U Category: [CategoryID: 2 / CategoryName: Components / Subcategory: [SubcategoryID : 14 / SubcategoryName: Road Frames]]
...	...

Figure 14
The structure of a product document

The developers decided to use MongoDB to provide the document database. They created two MongoDB collections in the database; **categories** which holds the denomormalized category and subcategory documents, and **products** which contains the product documents.

Adventure Works could have chosen one of many different document databases. They just happened to select MongoDB because that is the document database with which they are most familiar. Another document database could work equally as well.

Retrieving Documents from the Product Catalog

The developers at Adventure Works created the classes shown in the following code example to represent category and subcategory information. These classes mirror the structure of the documents shown in Figure 12. A **Category** object contains a collection of **Subcategory** objects. The **BSonId** attribute identifies the field that MongoDB should use as the document key when it stores and retrieves documents:

```
public class Category
{
    [BSonId]
    public int Id { get; set; }
    public string Name { get; set; }
    public IEnumerable<Subcategory> Subcategories { get; set; }
}

public class Subcategory
{
    [BSonId]
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Product documents are implemented by using the **Product** class shown below. The **SizeUnitofMeasure**, **WeightUnitOfMeasure**, and **ProductCategory** types represent the nested documents in a product document.

```
public class Product
{
    [BsonId]
    public int Id { get; set; }
    public string Name { get; set; }
    public string ProductNumber { get; set; }
    public string Color { get; set; }
    public decimal ListPrice { get; set; }
    public SizeUnitOfMeasure Size { get; set; }
    public WeightUnitOfMeasure Weight { get; set; }
    public string Class { get; set; }
    public string Style { get; set; }
    public ProductCategory Category { get; set; }
}

public class SizeUnitOfMeasure
{
    public string Value { get; set; }
    public string UnitOfMeasure { get; set; }
}

public class WeightUnitOfMeasure
{
    public decimal Value { get; set; }
    public string UnitOfMeasure { get; set; }
}

public class ProductCategory
{
    public int CategoryId { get; set; }
    public string Name { get; set; }
    public ProductSubcategory Subcategory { get; set; }
}

public class ProductSubcategory
{
    public int SubcategoryId { get; set; }
    public string Name { get; set; }
}
```

The developers used MongoDB to provide the document database, and followed the Repository pattern to isolate the document database-specific code from the rest of the Shopping application.

The **CategoriesController** class in the MvcWebApi web service retrieves category information from the document database by using the **CategoryRepository** class. The code that actually establishes the connection is located in the **BaseRepository** class from which the **CategoryRepository** class inherits. The address of the MongoDB server and the name of the database are passed as parameters to a constructor in this class that saves them locally. The **GetDatabase** method in the **BaseRepository** class creates a connection to the MongoDB database using this information (this method also checks to make sure that the database actually exists). The following code example shows the main elements of the **BaseRepository** class:

```
public class BaseRepository
{
    private readonly string databaseName;
    private readonly string hostNames;
    private readonly bool setWriteConcernToJournal;
    private readonly bool setWriteConcernToWMajority;
    ...

    public BaseRepository(string hostNames,
                          string databaseName,
                          bool setWriteConcernToJournal,
                          bool setWriteConcernToWMajorit)
    {
        if (string.IsNullOrWhiteSpace(hostNames))
        {
            throw new ArgumentNullException(hostNames);
        }

        if (string.IsNullOrWhiteSpace(databaseName))
        {
            throw new ArgumentNullException("databaseName");
        }

        this.hostNames = hostNames;
        this.databaseName = databaseName;
        this.setWriteConcernToJournal = setWriteConcernToJournal;
        this.setWriteConcernToWMajority = setWriteConcernToWMajority;
    }

    protected MongoDatabase GetDatabase()
    {
        string connectionString = "mongodb://" + hostNames;
        MongoClientSettings settings =
            MongoClientSettings.FromUrl(new MongoUrl(connectionString));
        if(setWriteConcernToJournal)
        {
            settings.WriteConcern.Journal = true;
        }
        if(setWriteConcernToWMajority)
        {
            settings.WriteConcern = WriteConcern.WMajority;
        }
        var mongoClient = new MongoClient(settings);
        var mongoServer = mongoClient.GetServer();
       if (!mongoServer.DatabaseExists(this.databaseName))
        {
            throw new MongoException(string.Format(
                                     CultureInfo.CurrentCulture,
                                     Strings.DatabaseDoesNotExist,
                                     this.databaseName))
        }

        var mongoDb = mongoServer.GetDatabase(databaseName);
        return mongoDb;
    }
}
```

Appendix A provides more information on how the Shopping application implements the Repository pattern.

You can find more information about the MongoDB APIs for C# and the .NET Framework online at [the MongoDB website](#).

*The **GetDatabase** method also ensures that all writes to the database are recorded in the MongoDB journal (the equivalent of the database transaction log used by SQL Server). This feature guarantees that the database can survive an unexpected shutdown and recover any information that has not been flushed from memory to the MongoDB database.*

The MongoDB-specific data access code is located in the **GetAllCategories** and **GetSubcategories** methods in the **CategoryRepository** class. The code for these methods use the C# APIs exposed by a .NET Framework assembly provided by MongoDB to communicate with the database.

The following code example shows how the **CategoryRepository** class retrieves the documents that describe the product categories. As described earlier, in the product catalog database, a category document also contains a list of subcategories. Therefore, to save network bandwidth when transmitting data back to the client, this method only returns the category ID and name from each document:

*In this code example, the **DE** namespace is an alias for the **DataAccess.Domain.Catalog** namespace that contains the domain entity classes used by the controllers.*

```
public class CategoryRepository : BaseRepository, ICategoryRepository
{
    private const string mongoCollection = "categories";

    ...

    public ICollection<DE.Category> GetAllCategories()
    {
    ...
        // This method is ONLY used for the front page of the reference
        // implementation.  As an optimization, instead of pulling
        // back the whole category document, we are only returning the _id
        // and name of the category with no associated subcategories.
        var categories = GetDatabase().GetCollection<Category>(mongoCollection)
            .FindAll()
            .SetFields(Fields.Include("_id", "name"))
            .ToList();

        if (categories == null)
        {
            return null;
        }

        var result = new List<DE.Category>();

        Mapper.Map(categories, result);
        return result;
    }

    ...
}
```

To optimize the query performed by this method, the developers at Adventure Works decided to create a covering index that spans the customer ID and name fields over the documents in the **categories** collection. These documents are very static, so there is minimal overhead associated with maintaining this index.

In the sample application, the code that creates the indexes for the product catalog is located in the PowerShell scripts that create and populate the MongoDB database.

Note that although the method only returns the category ID and name, the database server still has to read the entire category document, including subcategory information, into memory. However, this action causes the documents to be cached in memory on the database server, and the **GetSubcategories** method that retrieves data from the same documents and which is typically invoked soon afterwards by the Shopping application, should run more quickly as a result. The following code example shows the **GetSubcategories** method. This method makes use of the **GetMongoCategory** method (also shown) that retrieves the details of a single category from the document database:

```
public class CategoryRepository : BaseRepository, ICategoryRepository
{
    private const string mongoCollection = "categories";

    ...

    private Category GetMongoCategory(int categoryId)
    {
        return GetDatabase().GetCollection<Category>(mongoCollection).
            FindOneById(categoryId);
    }

    ...

    public ICollection<DE.Subcategory> GetSubcategories(int categoryId)
    {
        var category = GetMongoCategory(categoryId);

        if (category != null && category.Subcategories != null)
        {
            var subcategories = category.Subcategories.ToList();

            if (subcategories == null)
            {
                return null;
            }
            var result = new List<DE.Subcategory>();

            Mapper.Map(subcategories, result);
            return result.Select(s =>
                {
                    // Map the parent category.
                    Mapper.Map(category, s.Category);
                    return s;
                }).ToList();
        }
        else
        {
            return null;
        }
    }
}
```

For more information on how the repository classes uses AutoMapper, see the section "How the Entity Framework and MongoDB Repository Classes use AutoMapper to Convert Data" in Appendix A.

MongoDB stores data in BSON format in the document database. The MongoDB APIs deserialize the data retrieved from the database into .NET Framework objects (or collections of objects). The **GetAllCategories** and **GetSubcategories** methods convert this data into database-neutral domain entities by using AutoMapper before returning them.

The **ProductRepository** class, which is used by the **ProductsController** to fetch product information from the document database, follows a similar pattern except that it reads data from the **products** document collection. The **GetProducts** method in the **ProductRepository** class returns a list of products in a subcategory and the **GetProduct** method returns the details of a single product. The **GetProducts** method has to locate products by subcategory, so the developers created an index over the **SubcategoryID** field in the products document collection. The following code example shows how these methods are implemented:

```
public class ProductRepository : BaseRepository, IProductRepository
{
    private const string mongoCollection = "products";
    ...
    private IEnumerable<Product> GetMongoProducts(
        IDictionary<string, object> matchingFilter = null)
    {
        return (matchingFilter != null)
             ? GetDatabase().GetCollection<Product>(mongoCollection).
                 Find(new QueryDocument(matchingFilter))
            : GetDatabase().GetCollection<Product>(mongoCollection).FindAll();
    }

    private Product GetMongoProduct(int productId)
    {
        return GetDatabase().GetCollection<Product>(mongoCollection).
            FindOneById(productId);
    }
    ...
    public ICollection<DE.Product> GetProducts(int productSubcategoryId)
    {
    ...
        var products = GetMongoProducts(new Dictionary<string, object> {
                { "category.subcategory.subcategoryId", productSubcategoryId
}
            }).ToList();

        if (products == null || products.Count == 0)
        {
            return null;
        }
        var result = new List<DE.Product>();

        Mapper.Map(products, result);
        return result;
    ...
    }

    public DE.Product GetProduct(int productId)
    {
    ...
        var product = GetMongoProduct(productId);
        if (product == null)
        {
            return null;
        }

        var result = new DE.Product();

        Mapper.Map(product, result);
        return result;
    ...
    }
}
```

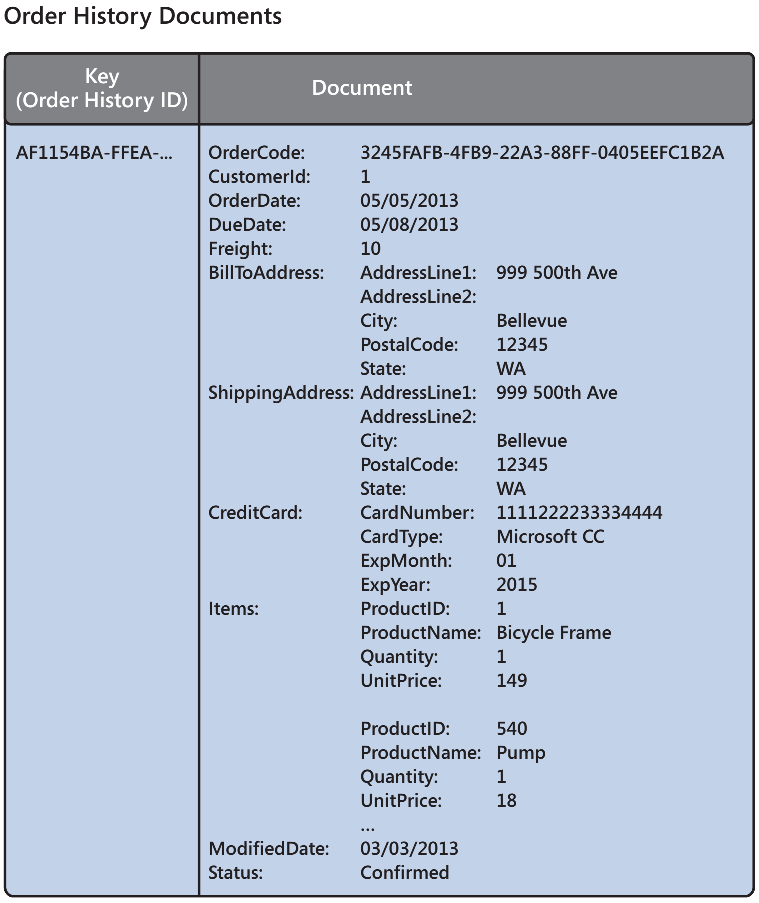
The **ProductExists** method in the **ProductRepository** class returns a Boolean value indicating whether the specified product is available in the product catalog. The **CartController** uses this method to check that the customer has added a valid product to the shopping cart. The MongoDB query that this method runs reads the document key only (the product ID field), and this information is available in the index over this field. Therefore this query can run very quickly and does not actually need to retrieve any product documents from the database:

```
public class ProductRepository : BaseRepository, IProductRepository
{
    private const string mongoCollection = "products";

    ...

    public bool ProductExists(int productId)
    {
        var mongoProductId = GetDatabase().GetCollection(mongoCollection)
            .Find(Query.EQ("_id", productId))
            .SetLimit(1)
            .SetFields(Fields.Include("_id"))
            .FirstOrDefault();
        return mongoProductId != null;
    }

    ...
}
```

Designing Documents to Hold Order History Information

When a customer places an order, the Shopping application creates a normalized set of records that it stores in the **SalesOrderHeader** and **SalesOrderDetails** tables in the SQL Server database. The Shopping application also has to maintain a full audit trail of each order, including a complete history of any changes made to each order. The **SalesOrderHeader** and **SalesOrderDetails** tables in SQL Server are designed to support the OLTP requirements of the warehousing and shipping systems in Adventure Works, but they are not suitable for storing the change history of orders. Equally, they are not optimized for performing the queries required by customers viewing order history information, especially as the database could contain tens of millions of orders. Therefore, when a customer places a new order, the developers at Adventure Works decided to copy the important details of the order as an order history document to a document database. Each time an order is modified the system stores another copy of the order history that contains the changes in the document database. Each copy is date stamped.

To enable order history information to be retrieved quickly, the developers adopted a fully denormalized approach where each order history document includes the complete details of the billing address, shipping address, credit card, and the list of items in the order. Figure 15 shows the structure of a typical order history record as stored in the document database.



Order History Documents

Key (Order History ID)	Document		
AF1154BA-FFEA-...	OrderCode:	3245FAFB-4FB9-22A3-88FF-0405EEFC1B2A	
	CustomerId:	1	
	OrderDate:	05/05/2013	
	DueDate:	05/08/2013	
	Freight:	10	
	BillToAddress:	AddressLine1:	999 500th Ave
		AddressLine2:	
		City:	Bellevue
		PostalCode:	12345
		State:	WA
	ShippingAddress:	AddressLine1:	999 500th Ave
		AddressLine2:	
		City:	Bellevue
		PostalCode:	12345
		State:	WA
	CreditCard:	CardNumber:	1111222233334444
		CardType:	Microsoft CC
		ExpMonth:	01
		ExpYear:	2015
	Items:	ProductID:	1
		ProductName:	Bicycle Frame
		Quantity:	1
		UnitPrice:	149
		ProductID:	540
		ProductName:	Pump
		Quantity:	1
		UnitPrice:	18
		...	
	ModifiedDate:	03/03/2013	
	Status:	Confirmed	

FIGURE 15
An order history document showing the denormalized structure of the data

In this document, the key (order history ID) is a unique GUID, while the **OrderCode** field actually identifies the order; the same order can have more than one order history record. The **ModifiedDate** field contains the date and time that the order history record was created (this will be the time when the order itself was either placed or modified), and the **Status** field tracks the status of the order. The remaining fields contain the details of the order.

*When an order is first placed the status will be marked as **Pending**, and when the order has been processed its status is changed to **Completed**. Therefore, there will usually be at least two order history documents for each order, comprising an audit trail of these changes in status.*

In the real world, the warehousing and dispatch systems inside Adventure Works may cause the order to pass through additional states, and they will add further records that indicate these changes for each order to the document database. However, the warehousing and dispatch systems are outside the scope of the Shopping application, and this functionality is not implemented by the sample solution provided with this guide.

The order history documents are stored in the **ordershistory** collection in the database.

Storing and Retrieving Order History Information

Following the same pattern as the product catalog, the developers at Adventure Works created a series of classes to hold order history information. The following code example shows these classes:

```
public class OrderHistory
{
    ...
    [BSonId]
    public Guid Id { get; set; }
    public Address BillToAddress { get; set; }
    public CreditCard CreditCard { get; set; }
    public int CustomerId { get; set; }
    public DateTime DueDate { get; set; }
    public decimal Freight { get; set; }
    public IEnumerable<OrderItem> Items { get; set; }
    public DateTime ModifiedDate { get; set; }
    public Guid OrderCode { get; set; }
    public DateTime OrderDate { get; set; }
    public Address ShippingAddress { get; set; }
    public OrderStatus Status { get; set; }
}

public class OrderItem
{
    public short Quantity { get; set; }
    public decimal UnitPrice { get; set; }
    public int ProductId { get; set; }
    public string ProductName { get; set; }
}
```

```
public class Address
{
    public int Id { get; set; }
    public string AddressLine1 { get; set; }
    public string AddressLine2 { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
    public int StateProvinceId { get; set; }
}

public class CreditCard
{
    public string CardNumber { get; set; }
    public string CardType { get; set; }
    public byte ExpMonth { get; set; }
    public short ExpYear { get; set; }
}
```

When a customer places an order, the **Post** method of the **OrdersController** class creates an initial order history document containing the details of the order, and stores it in the **ordershistory** collection in the database with the status of **Pending**. When the order has been processed, another order history document is created for the same order and added to the **ordershistory** collection with the status set to **Completed** (the **ModifiedDate** field is also populated with the date and time that the status changed).

To enable a customer to display their order history, the developers at Adventure Works created the **OrderHistoryRepository** class that provides the following public methods:

- **GetOrderHistoryByTrackingId.** This method retrieves the details of a specific order from the order history document database. The order is specified by the tracking ID passed as the parameter to this method. Note that there will be more than one order history document for a given order, and this method simply returns the first order history document that it finds that has the appropriate order code.
- **GetOrdersHistories.** This method retrieves a list of **OrderHistory** documents for a specified customer. The customer ID is provided as the parameter to this method.
- **GetOrderHistoryByHistoryId.** This method fetches the details of a specific **OrderHistory** document identified by order history ID passed as the parameter to this method.
- **IsOrderCompleted.** This method determines whether the specified order has been successfully completed or not by examining the **Status** property of the order. If this field contains the value **OrderStatus.Completed**, the method returns true, otherwise it returns false. The method uses the unique tracking ID to identify the order.
- **GetPendingOrderByTrackingId.** This method retrieves the **Order** document for the order that matches the tracking ID specified as the parameter. The method examines the Status field of the order, and only returns order information if this status is **OrderStatus.Pending**. If the status is different, or there is no order that matches the tracking ID, this method returns a null value.

Internally, the code in the **OrderHistoryRepository** class operates along similar lines to the other MongoDB repository classes, and uses methods defined by the MongoDB API to retrieve and store documents in the database.

The order history documents in the database are automatically indexed by the order history ID (this is the **Id** field in the **OrderHistory** class). However, the **OrderHistoryRepository** class also retrieves documents by using the **OrderCode** and **CustomerId** fields. Therefore the developers defined indexes over these fields in the **ordershistory** collection.

The indexes are created by the PowerShell script that sets up the MongoDB database.

Implementing a Document Database to Maximize Scalability, Availability, and Consistency

Most document databases are intended to handle large volumes of data being accessed concurrently by millions of users. It is therefore vital that a document database is able to locate, retrieve, insert, update, and delete documents quickly and efficiently.

Maximizing Scalability

As with many other types of NoSQL databases, most scalable document databases enable you to spread a large database across a collection of database servers by using sharding, described in Chapter 1. This is very effective for applications that perform a large proportion of write operations as it can help to reduce disk contention.

In a document database, sharding is implemented at the document collection level, meaning that different collections have their own sets of partitions and shards. A collection of customer documents does not usually occupy the same shard as a collection of orders documents, for example.

Sharding strategies vary from simple hashing of the document key (in a manner similar to key/value stores), to schemes based on the values of one or more fields in a collection of documents. Some document database support range partitioning, enabling you to group documents with adjacent key values together in the same shard. In a few cases, you can customize the sharding process, specifying how the database management system should determine in which shard a given document should be placed, and where to store the document in that shard. The rationale behind customizing the sharding process is that a generic sharding mechanism may not always be suitable for the likely access patterns used required your applications. For example, most document databases do not support referential integrity and they treat each document as an independent entity. By default, if the database management system needs to relocate a document to a different shard it will not worry about keeping related documents together. If you know that certain groups of documents are always likely to be accessed as together, customizing the sharding process can enable you to retrieve these documents more efficiently.

Most document databases that provide sharding automatically rebalance data across the shards as documents are created, modified, and deleted. However, the shard key that you select can have a significant influence on the performance of sharding. The following list highlights some recommendations:

- If possible, select a shard key that matches the most common queries performed by your applications.
- Do not use a shard key that has very few possible values. This can result in a small number of very large shards, reducing the effectiveness of distributing data across servers.
- Unless you are partitioning data into ranges, do not select a shard key that has low randomness. Highly random keys will enable the database management system to spread the data more evenly across shards. Less random keys may result in hotspots occurring in the database.
- If you are inserting a large number of documents and you are not hashing the shard key, do not base the shard key on the date and time that each document is added to the database. This can result in poor write scalability as multiple documents may be written to the same shard in quick succession.

Refer back to the section "Implementing High Availability" in Chapter 1 for more information on replication strategies.

Ensuring Availability

Commercial document databases usually implement availability by using primary/secondary replication with automatic failover and recovery. A few document databases utilize peer-to-peer replication.

In a typical document database, each shard in a cluster will be replicated, and read requests will be directed to the most available replica. You may also be able to implement cross-cluster replication to improve the locality of data. For example, if you replicate a cluster to a data center located in San Francisco, a user based in the western United States can access the data more quickly than if the cluster was based in Tokyo.

Maximizing Consistency

As described in the preceding sections in this chapter, many document databases attempt to optimize operations by reducing the overhead associated with these operations, therefore it is uncommon for a document database to guarantee any form of cross-document transactional consistency.

> *To ensure good performance, document databases typically only provide eventual consistency when you create, modify, or delete documents. It is often the responsibility of applications to handle any data inconsistencies that may occur.*

Document databases that implement replication often use read quorums (described in Chapter 1) to reduce the possibility of inconsistencies occurring when applications retrieve documents. However, read quorums usually only operate within a cluster. Cross-cluster replication will increase the latency of inserts, updates, and deletes. A change made to data in Tokyo has to propagate to the cluster in San Francisco before it is visible to a user in the western United States, possibly resulting in inconsistencies between clusters. In many cases, documents that are modified in Japan will most likely be queried by Japanese users, and documents that are modified in the western United States will most likely be queried by American users, so such inconsistencies might not occur very often. However, your applications should be prepared to handle this eventuality.

How Adventure Works Implemented Scalability and Availability for the Product Catalog and Order History

The product catalog for the Shopping application contains category, subcategory, and product details. As far as the Shopping application is concerned, this data is read-only (the data can change, but such changes are very infrequent and are outside the scope of the Shopping application). Additionally, the set of categories, subcategories, and products is very static. On the other hand, the set of orders generated by the Shopping application constitutes a rapidly increasing collection of documents, as do the documents in the order history collection. Because the product catalog and orders exhibit decidedly different behaviors, the developers at Adventure Works decided to consider their scalability and availability requirements separately.

Implementing Scalability and Availability for the Product Catalog

The section "How Adventure Works Used a Document Database to Store the Product Catalog and Orders" earlier in this chapter described how the developers at Adventure Works divided the product catalog into two document collections; the first collection holds category and subcategory information, and the second collection contains the details of products.

The number of documents in the category and subcategory information collection is very small (there are only four categories), so the developers at Adventure Works decided not to shard this collection. Similarly, although the products collection is considerably larger, it is still of a reasonably finite size and is static, so the developers chose not to shard this collection either.

Sharding a small collection adds an overhead that can reduce the responsiveness of queries that retrieve data from this collection. If you are using MongoDB, you should only consider sharding if the dataset is bigger than the storage capacity of a single node in the system. For example, if the Adventure Works product line consisted of millions of items, the developers could have decided to shard the products collection by using a hash of the Product ID as the shard key. The reason for this choice is that most of the queries performed by the Shopping application that retrieve product information do so by specifying the product ID. Product IDs are assigned in a monotonic increasing sequence, so the most recent products have the highest IDs. It is also possible that these are the most popular products that will be queried most often, so hashing the product ID ensures that the documents for the latest products are distributed evenly across the shards.

To ensure good read-scalability, the developers implemented replication by using MongoDB replica sets with 12 members (the maximum number of nodes that a MongoDB replica set can currently contain). The default configuration of a MongoDB cluster is to perform all read and write operations through the primary server in a cluster and the secondary nodes act as failover servers in case the primary server should crash. However, because the Shopping application does not modify information in the product catalog, the Shopping application was configured to enable read operations to be satisfied by all secondary servers as well.

To reduce network latency, the developers deployed the Shopping application to multiple Windows Azure data centers located in various regions around the world. They also deployed a copy of the product catalog cluster to a collection of virtual machines running at each data center. Figure 16 shows the architecture of the Shopping application and the product catalog at a single data center. Notice that because of the read-only nature of the product catalog, the developers did not configure any form of cross-cluster replication between data centers; they simply deployed a copy of the same cluster at each data center.

The section "How Adventure Works Implemented Scalability and Availability, and Maximized Consistency for Shopping Cart Information" in Chapter 4, "Implementing a Key/Value Store," describes how the developers at Adventure Works deployed the Shopping application to the various Windows Azure data centers around the world.

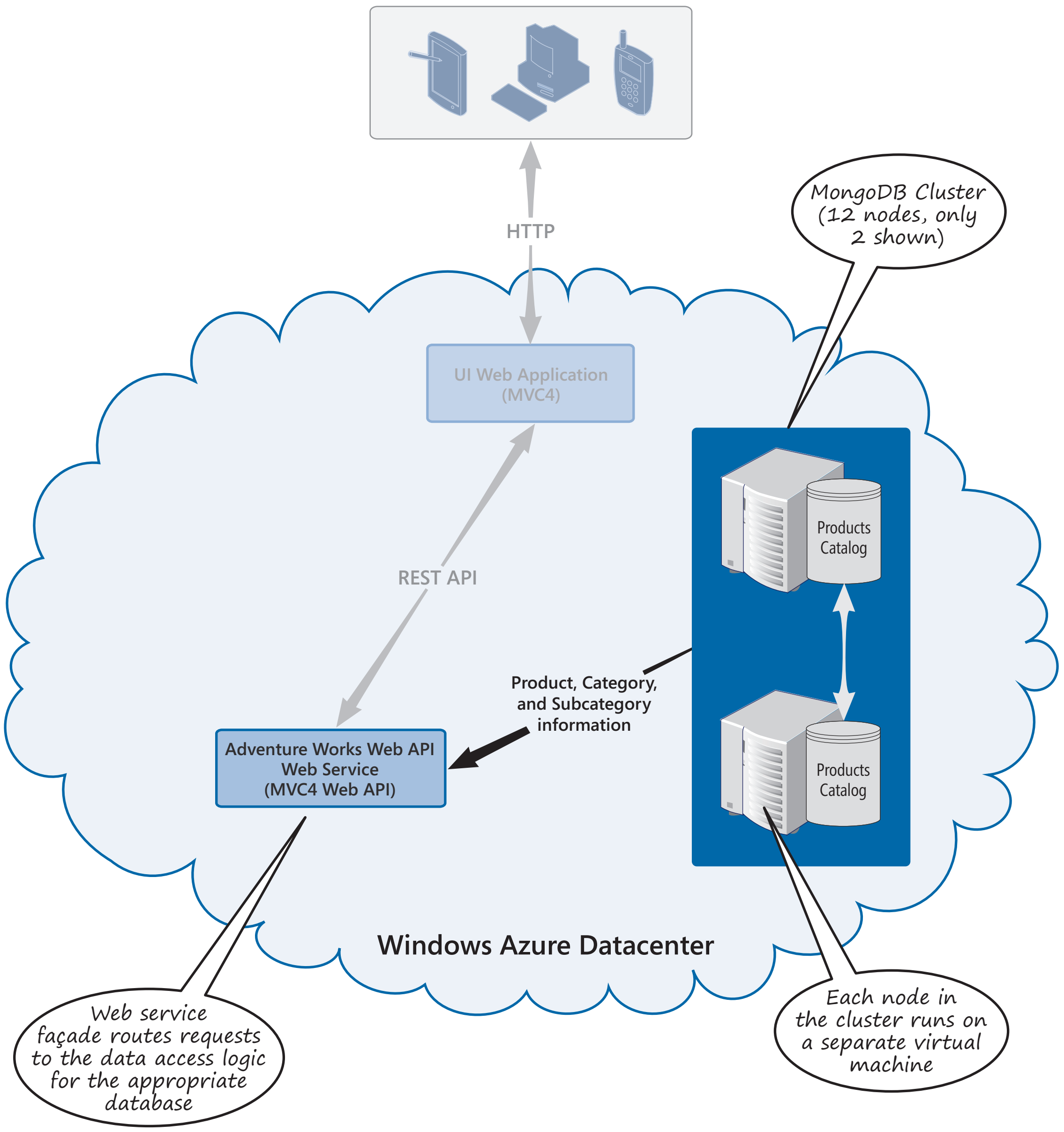

HTTP
UI Web Application (MVC4)
MongoDB Cluster (12 nodes, only 2 shown)
REST API
Products Catalog
Products Catalog
Product, Category, and Subcategory information
Adventure Works Web API Web Service (MVC4 Web API)
Windows Azure Datacenter
Web service façade routes requests to the data access logic for the appropriate database
Each node in the cluster runs on a separate virtual machine

FIGURE 16
The product catalog implemented as a MongoDB cluster by using virtual machines in a Windows Azure data center

Implementing Scalability and Availability for Order History Information

The Shopping application needs to be able to store and retrieve order history documents from a large and rapidly increasing dataset. The indexes over the **OrderCode** and **CustomerId** fields in documents in the **ordershistory** collection enable the application to locate **OrderHistory** documents quickly. However, the system may also need to support many hundreds of write operations per second if a large number of customers are placing orders concurrently. To reduce the possible contention that could occur, the developers decided to shard the **ordershistory** collection and hash the **CustomerId** as the shard key. This approach helps to keep **OrderHistory** documents for a given customer close to each other (ideally in the same shard), but distributes the documents for other customers across the shard cluster. To maximize availability, each shard is actually a replica set. Each replica is implemented as a MongoDB database running on a separate virtual machine hosted using Windows Azure (these virtual machines are distinct from the virtual machines that contain the product catalog).

The Shopping application is hosted in multiple datacenters. As described above, each data center hosts its own set of replicas containing the product catalog. This approach is feasible for data that does not change. The situation with order history documents is somewhat different, and it is impractical to copy order histories to replica sets in different datacenters (at least in real time). The developers instead chose to store order history information for a customer only at the datacenter to which that customer connects. This means that each datacenter holds only its own non-overlapping collection of order history documents. The disadvantage of this approach is that if a customer connects to a different datacenter then their order history documents will not be available, but this is likely to be an infrequent occurrence (customers tend not to move between geographic regions very often).

Summary

This chapter has described strategies for using a document database, and the circumstances under which a document store is a better choice than a key/value store for holding information. A document is an entity that contains a collection of fields, and a document database enables an application to perform queries that retrieve documents based on the values in these fields. In this way, they provide a more flexible mechanism for locating and fetching data than the simple approach provided by a key/value store. You can also create indexes over fields to improve the speed of queries, but indexes can also have a detrimental effect on insert, update, and delete operations, so only define the indexes that your applications really require.

You should pay careful attention to how you design your documents. Do not simply follow the relational approach because this will inevitably lead to poor retrieval performance. Instead, you should examine the requirements of your applications and the data that they use, and then design your documents to optimize the operations that read data; denormalize your data where appropriate.

This chapter also described how Adventure Works used document stores to hold the product catalog and order history information for the Shopping application. It discussed the document structures that the developers decided would best meet the requirements of the application, and has shown how the developers implemented the databases to maximize scalability and availability by using replication and clustering.

More Information

All links in this book are accessible from the book's online bibliography available at:
http://msdn.microsoft.com/en-us/library/dn320459.aspx.

- You can find an overview of document t databases in the article "What the Heck Are Document Databases?" available on MSDN at *http://msdn.microsoft.com/magazine/hh547103.aspx*.
- You can find detailed information about MongoDB on their website at *www.mongodb.org*. Detailed information about the MongoDB driver for C# and the .NET Framework is available at *http://docs.mongodb.org/ecosystem/drivers/csharp/*.

6 Implementing a Column-Family Database

Key/value stores and document databases are very row focused. What this means is that they tend to be optimized to enable an application to retrieve the data for complete entities that match one or more criteria. However, sometimes an application needs to retrieve data from a subset of fields across a collection of documents, in a manner comparable to performing a projection operation in SQL. Column-family databases enable you to store information in a more column-centric approach than is possible with most key/value stores or document databases.

In a column-family database, you can structure the rows as collections of columns. A single row in a column-family database can contain many columns. You group related columns together into column families and you can retrieve columnar data for multiple entities by iterating through a column family. The flexibility of column families gives your applications a wide scope to perform complex queries and data analyses, similar in many ways to the functionality supported by a relational database.

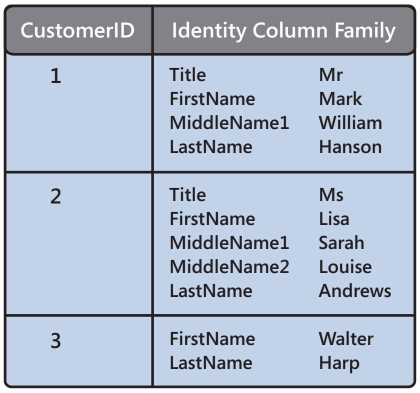
Column-family databases are designed to hold vast amounts of data (hundreds of millions, or billions of rows containing hundreds of columns), while at the same time providing very fast access to this data coupled with an efficient storage mechanism. A well-designed column-family database is inherently faster and more scalable that a relational database that holds an equivalent volume of data. However, this performance comes at a price; a typical column-family database is designed to support a specific set of queries and as a result it tends to be less generalized than a relational database. Therefore, to make best use of a column-family database, you should design the column families to optimize the most common queries that your applications run.

When you design a column-family database, focus on the queries that your application needs to perform and optimize the structures to support these queries.

This chapter provides information on how to design and implement the schema for a column-family database to best meet the needs of your applications.

What is a Column Family?

As the name implies, the column family is the feature that distinguishes a column-family database from other forms of NoSQL databases. A column family is simply a collection of columns that hold the data for a set of entities. Chapter 1, "Data Storage for Modern High-Performance Business Applications" described how, in the simplest of cases, you can think of a column family as being conceptually similar to a table in a relational database because data is organized as a set of rows and columns. However, unlike a table in relational database, the columns in a column family do not have to conform to a rigidly defined schema for every row. In fact, it is preferable to think of a column family as a map of name/value pairs where the contents of this map can vary on a row by row basis. For example, if you need to store the names of customers in a column-family database, you could create the Identity column family shown in Figure 1. This column family holds the title, first name, middle names, and last name of each customer. In this diagram, note that each customer is identified and sorted by using a unique key (CustomerID), but other than that the structure of the data for each customer can be different. If a row does not require a particular column, you can simply omit it from that row. This approach enables a columnfamily to store sparse data very efficiently compared to a table in a relational database.

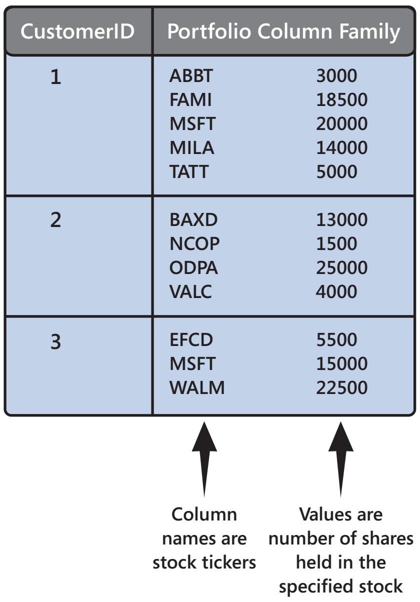


CustomerID	Identity Column Family	
1	Title	Mr
	FirstName	Mark
	MiddleName1	William
	LastName	Hanson
2	Title	Ms
	FirstName	Lisa
	MiddleName1	Sarah
	MiddleName2	Louise
	LastName	Andrews
3	FirstName	Walter
	LastName	Harp

Figure 1
A simple column family with a fixed set of column names holding customer information

Figure 1 uses Title, FirstName, MiddleName1, MiddleName2 (continuing this sequence to as many middle names a particular customer might have), and LastName as a set of domain-specific column names. Column-family databases do not enforce any type of schema on the rows in any given column family, and you can give the columns in each row completely different names. However, depending on the queries that your applications need to perform, it is advisable to establish a convention for the column names otherwise it can be difficult to interrogate the database (if an application does not know the names of the columns, it will need to discover them somehow).

Some scenarios are more suited to using generated names for columns rather than domain-specific identifiers. As an example, Figure 2 shows a column family created as part of a shares portfolio management system for a financial institution that monitors the stocks and shares held by its customers. The portfolio for a customer can contain stocks and shares for any number of publicly listed companies. The Portfolio column family uses stock tickers as the column names and records the number of shares held as the value of each column. If a customer does not hold shares in a particular stock, then it is not listed as part of that customer's portfolio. If a customer invests in a new stock, it is easy to add a column identified by using the appropriate stock ticker symbol to the list of stocks and shares for that customer.



CustomerID	Portfolio Column Family	
1	ABBT	3000
	FAMI	18500
	MSFT	20000
	MILA	14000
	TATT	5000
2	BAXD	13000
	NCOP	1500
	ODPA	25000
	VALC	4000
3	EFCD	5500
	MSFT	15000
	WALM	22500

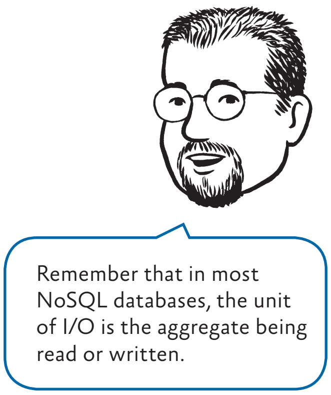
Column names are stock tickers
Values are number of shares held in the specified stock

Figure 2
A column family with a dynamic set of column names based on stock ticker symbols

Most column-family databases only support simple, scalar types as the values of columns. However, a few column-family databases allow an individual column be structured as a collection of child columns, leading to a hierarchical arrangement. Columns that contain collections of child columns are sometimes referred to as super columns.

As described earlier, the data for an entity in a column-family database can span a large number of columns, and you should use column families to group logically related data together. In the examples shown in Figures 1 and 2, both column families could be part of the same database, each column family storing data about a different aspect of a customer (identity information and the portfolio of shares that a customer holds). Both column families use the same row key (CustomerID), and an application can use this key to retrieve the data for a specific customer. The collections of columns in a column family are also ordered by using this row key to enable the database to find data for a specific row quickly.

Many column-family databases also sort the columns within a row by the column name. In Figure 2, the stock ticker symbol columns for each row are stored in alphabetical order. A few column-family databases enable you to customize the way in which columns are sorted.


Remember that in most NoSQL databases, the unit of I/O is the aggregate being read or written.

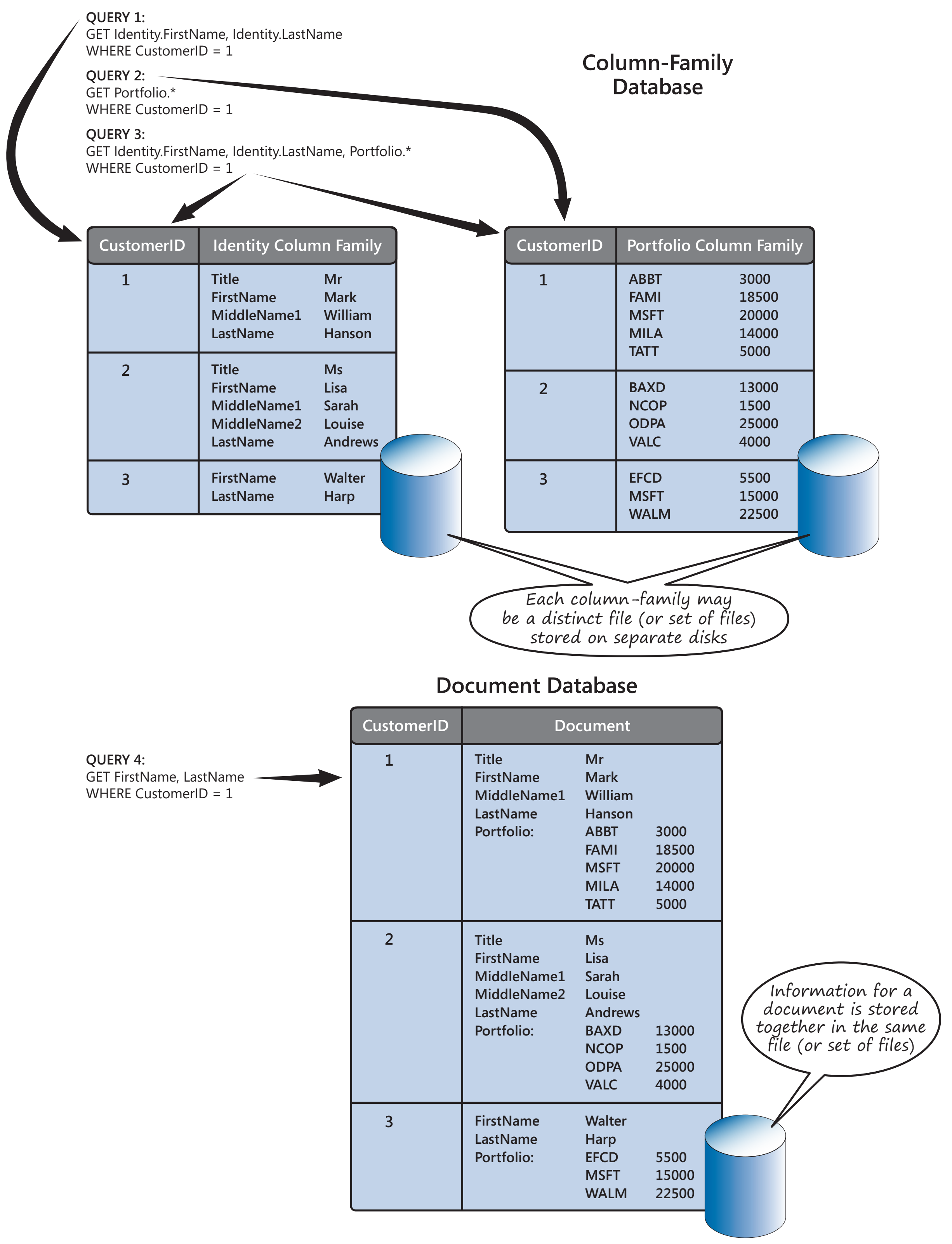
In most column-family databases, a column family also defines the physical storage for the data; different column families are usually stored as separate files, possibly on different disks (the actual implementation depends on the column-family database that you are using). What this means in practical terms is that when an application queries the data for an entity, the database software only needs to read data from the column families that actually contain the data being retrieved. In contrast, if you are using a document database or a key/value store, the database software typically either has to fetch an entire document or value.

Many document databases implement projections which enable an application to specify the fields that a query should return. However, the document database still needs to read the entire document from storage in order to extract this information and return it to the application. Similarly, updating the data in a document may necessitate rewriting the entire document to disk.

A few document databases enable you to create covering indexes *that can remove the requirement to read an entire document for queries that only fetch data in indexed fields. Refer to Chapter 5, "Implementing a Document Database" for more information.*

The physical structure of a column-family database enables you to partition your data vertically. This strategy helps to minimize the amount of data that the database software actually needs to read from disk (or memory) to satisfy a query, or that it needs to save to disk (or memory) to update information.

Similarly, if an application changes the value in a column, the database software only needs to write data back to the affected column family; data for the same row held in other column-families remains unaffected. If you are using a document database or a key/value store, modifying the information in a single field in a row may necessitate writing the entire document or value back to disk. Figure 3 illustrates the process of querying data in a column-family database compared to a document database.


Column-Family Database
Each column-family may be a distinct file (or set of files) stored on separate disks
Document Database
Information for a document is stored together in the same file (or set of files)

QUERY 1:
```
GET Identity.FirstName, Identity.LastName
WHERE CustomerID = 1
```

QUERY 2:
```
GET Portfolio.*
WHERE CustomerID = 1
```

QUERY 3:
```
GET Identity.FirstName, Identity.LastName, Portfolio.*
WHERE CustomerID = 1
```

CustomerID	Identity Column Family	
1	Title FirstName MiddleName1 LastName	Mr Mark William Hanson
2	Title FirstName MiddleName1 MiddleName2 LastName	Ms Lisa Sarah Louise Andrews
3	FirstName LastName	Walter Harp

CustomerID	Portfolio Column Family	
1	ABBT FAMI MSFT MILA TATT	3000 18500 20000 14000 5000
2	BAXD NCOP ODPA VALC	13000 1500 25000 4000
3	EFCD MSFT WALM	5500 15000 22500

QUERY 4:
```
GET FirstName, LastName
WHERE CustomerID = 1
```

CustomerID	Document		
1	Title FirstName MiddleName1 LastName Portfolio:	Mr Mark William Hanson ABBT FAMI MSFT MILA TATT	 3000 18500 20000 14000 5000
2	Title FirstName MiddleName1 MiddleName2 LastName Portfolio:	Ms Lisa Sarah Louise Andrews BAXD NCOP ODPA VALC	 13000 1500 25000 4000
3	FirstName LastName Portfolio:	Walter Harp EFCD MSFT WALM	 5500 15000 22500

Figure 3
Retrieving data from a column-family database compared to a document database

The syntax shown for QUERY1*,* QUERY2*,* QUERY3*, and* QUERY4 *in Figure 3 is for illustrative purposes only. It is not intended to be an example of any particular query language for a column-family database or a document database.*

In Figure 3, *QUERY 1* retrieves customer names from the column-family database and only accesses data in the **Identity** column family. Similarly, *QUERY 2* fetches information about the shares that a customer holds and only accesses data in the **Portfolio** column family (the term **Portfolio.*** means *all columns in the Portfolio column family*). Only *QUERY 3* that combines the data to retrieve the names of customers and the shares that they hold needs to access both column-families. *QUERY 4* that retrieves customer names from a document database holding identity and portfolio information actually has to read the entire document (including the shares data) to fetch just the customer details.

Most column-family databases and document databases cache data in memory, so the amount of physical I/O performed by a database to retrieve data may be less than that indicated by the previous discussion. However, the same principles apply to cached data. If you need to cache an entire document in memory rather than just the relevant parts of the data defined by a column family, you may be wasting precious memory resources which will, in turn, affect the performance of the database.

Designing a Column-Family Database

A column-family database is a suitable repository for capturing large amounts of sparse, volatile information very quickly, and providing efficient query access to this information. However, to make the best use of a column-family database you need to design your column families carefully. You need to understand the data that your applications capture, and the queries that they need to perform over this data. The following sections describe some of the factors that you need to think about.

Designing Column Families for Efficient Data Storage and Retrieval

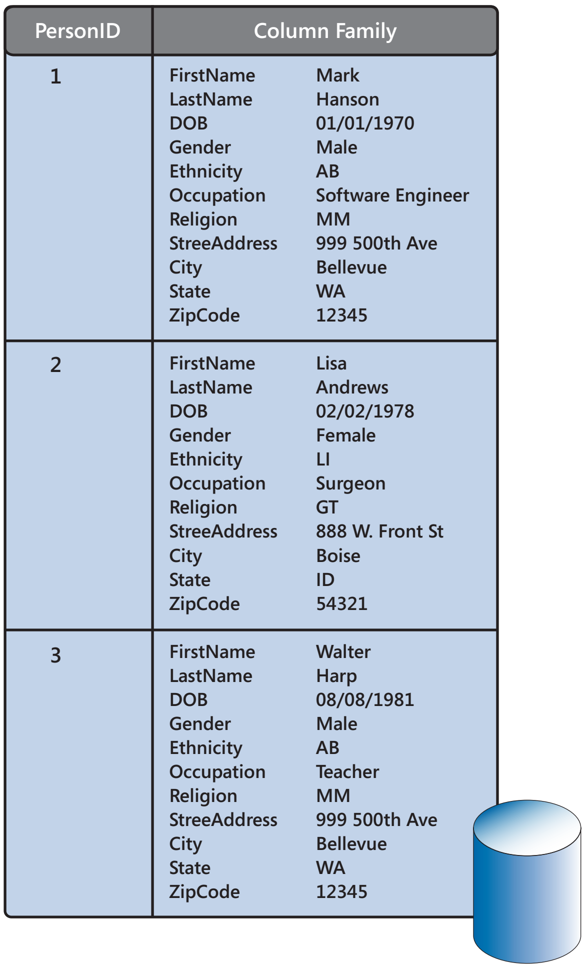
Remember that in a typical column-family database, a column family fulfills two purposes:

- At the logical level, it groups related columns for an entity together.
- At the physical level, it defines the file and storage structure of the database.

Design your column families to optimize the I/O and caching performed by the database.

A well-designed column-family database enables an application to satisfy the majority of its queries by visiting as few column families as possible, while ensuring that the data held by each column family is relevant to the queries that reference them. In other words, you should partition the data that constitutes an entity vertically into sets of columns, where each set fully satisfies one or more queries but does not contain columns that are not required by most queries. Each set of columns is a candidate to become a column family. Ideally each column family should be non-overlapping (you should try and avoid duplicating the same data for the same row across different column families), although if the data is relatively static storing multiple copies of a column in different column families may help to optimize some queries.

As an example, consider a column-family database that stores census information. A typical census database contains information such as the name, address, gender, date of birth, occupation, ethnicity, and religion for every member of the population, and it could contain hundreds of millions of rows. Additionally, a census database has to support a varied set of queries such as "How many people are aged between 40 and 50?," "What is the most common occupation for women in their 20s?," or "What is the ethnic mix of the population?" It would be possible to store all this data in a single column family as shown in Figure 4 below.



PersonID	Column Family	
1	FirstName	Mark
	LastName	Hanson
	DOB	01/01/1970
	Gender	Male
	Ethnicity	AB
	Occupation	Software Engineer
	Religion	MM
	StreeAddress	999 500th Ave
	City	Bellevue
	State	WA
	ZipCode	12345
2	FirstName	Lisa
	LastName	Andrews
	DOB	02/02/1978
	Gender	Female
	Ethnicity	LI
	Occupation	Surgeon
	Religion	GT
	StreeAddress	888 W. Front St
	City	Boise
	State	ID
	ZipCode	54321
3	FirstName	Walter
	LastName	Harp
	DOB	08/08/1981
	Gender	Male
	Ethnicity	AB
	Occupation	Teacher
	Religion	MM
	StreeAddress	999 500th Ave
	City	Bellevue
	State	WA
	ZipCode	12345

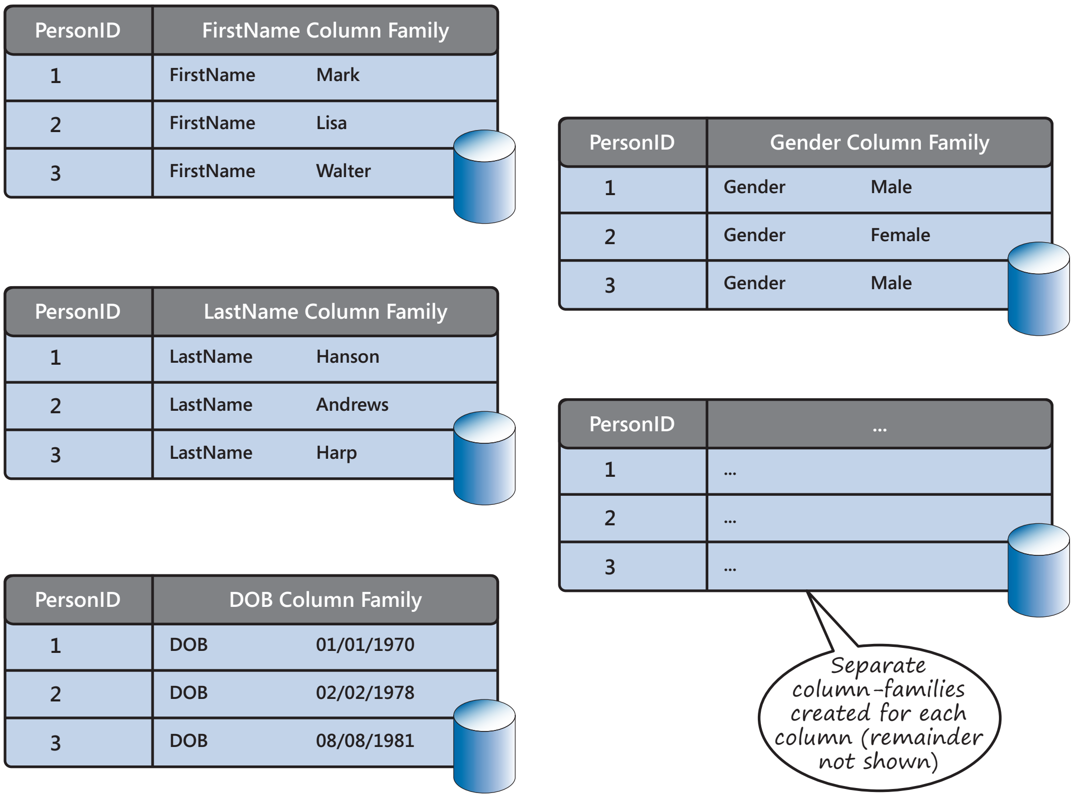
Figure 4
Implementing the census database as a single column family

> *A column-family database often sorts the columns by column name within a row, so the order of the columns in the column family for the census database are likely to be different from that shown in Figure 4.*

If a column-family database caches column families in memory, then caching data for a column family containing a lot of irrelevant or infrequently accessed data is a waste of resources.

This strategy results in a structure that is very similar to that of a document database. A query that only needs to find the age of a person requires the database software to retrieve the entire set of columns for that person, possibly resulting in a significant amount of disk I/O and negating an important advantage of using a column-family database as described in the previous section. The same is true for queries that simply need to find the name or occupation of a person.

An alternative (and probably impractical) approach is to store each column in its own column family, as shown in Figure 5. This time, the database software only needs to read the data that satisfies each query performed by the application, but because the data is spread across multiple column families retrieving this data may require performing a separate I/O operation for each column.



PersonID	FirstName Column Family	
1	FirstName	Mark
2	FirstName	Lisa
3	FirstName	Walter

PersonID	LastName Column Family	
1	LastName	Hanson
2	LastName	Andrews
3	LastName	Harp

PersonID	DOB Column Family	
1	DOB	01/01/1970
2	DOB	02/02/1978
3	DOB	08/08/1981

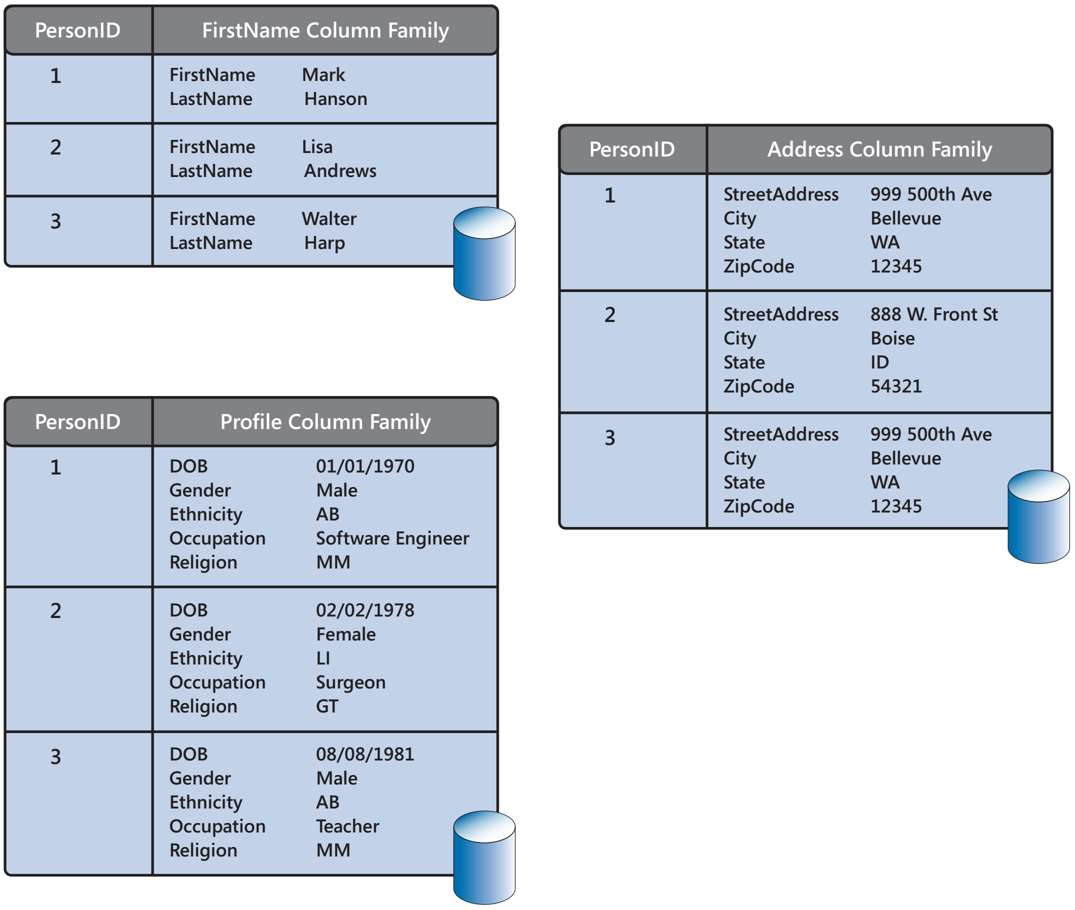
PersonID	Gender Column Family	
1	Gender	Male
2	Gender	Female
3	Gender	Male

PersonID	...
1	...
2	...
3	...

Separate column-families created for each column (remainder not shown)

Figure 5
Implementing the census database as a large collection of column families

In the census database, it is more optimal to divide this data into separate column families based on the query requirements of the application. Figure 6 shows the same data segregated into **Name**, **Profile**, and **Address** column families. Each column family groups data that is regularly queried together into the same vertical partition, but isolates it from other data that is not commonly required by these queries.



PersonID	FirstName Column Family	
1	FirstName	Mark
	LastName	Hanson
2	FirstName	Lisa
	LastName	Andrews
3	FirstName	Walter
	LastName	Harp

PersonID	Address Column Family	
1	StreetAddress	999 500th Ave
	City	Bellevue
	State	WA
	ZipCode	12345
2	StreetAddress	888 W. Front St
	City	Boise
	State	ID
	ZipCode	54321
3	StreetAddress	999 500th Ave
	City	Bellevue
	State	WA
	ZipCode	12345

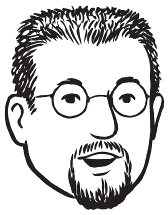
PersonID	Profile Column Family	
1	DOB	01/01/1970
	Gender	Male
	Ethnicity	AB
	Occupation	Software Engineer
	Religion	MM
2	DOB	02/02/1978
	Gender	Female
	Ethnicity	LI
	Occupation	Surgeon
	Religion	GT
3	DOB	08/08/1981
	Gender	Male
	Ethnicity	AB
	Occupation	Teacher
	Religion	MM

Figure 6
Implementing the census database as a Name, Profile, and Address column families

It is important to establish the column families that your applications require early on in the development cycle. Adding a new column family to an existing database may require you to take the database offline while the operation is in progress (information about column families has to be propagated across all database servers if you are implementing sharding and replication). Migrating columns between column families is also a non-trivial issue, especially if you have hundreds of millions of rows.

Using Wide Column Families

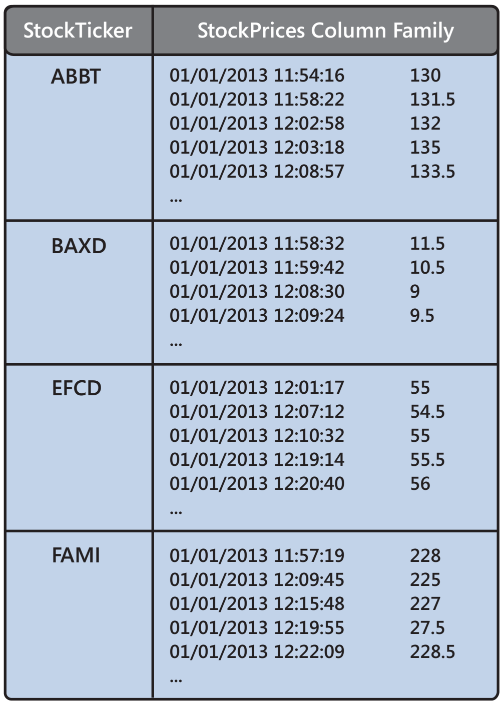
The census database illustrates an example where there may be many hundreds of millions of rows, but each row only consists of column families that contain only a few columns. These types of column families are referred to as *narrow* column families.



Wherever possible, you should design a column-family database to take advantage of wide column families.

The purpose of defining the column families in this way was to avoid the I/O overhead of retrieving *irrelevant* data from disk. However, performing a few large I/O requests that fetch *relevant* data is always going to be more efficient than performing lots of small requests for the same information, and you can exploit this mechanism to good advantage if your applications frequently need to process entities that contain significant amounts of information. The **Portolio** column-family example shown earlier in this chapter described how the columns in a column family can be dynamically generated to list the volume of stock that each customer has purchased. There is effectively no limit to the number of different stocks that a customer can hold, so each row could potentially contain a large number of columns. If the customer is an institution representing a large pension fund rather than an individual person, it is possible that it could hold shares in many hundreds or thousands of different stocks. If an application needs to find the shares held by a given customer, all of these items are located in the same row in the same column family, and they can be retrieved very quickly by performing a small number of contiguous I/O operations. Column families where the rows contain a large number of columns are called *wide* column families.

As a second example, suppose that you are creating a database that needs to hold information about the prices of stocks as they are traded on the stock market. This information can change with great rapidity. You might also need to retain a history of previous prices for applications that analyze trends in stock market data. Figure 7 shows how you can store this data in a wide column family.



StockTicker	StockPrices Column Family	
ABBT	01/01/2013 11:54:16	130
	01/01/2013 11:58:22	131.5
	01/01/2013 12:02:58	132
	01/01/2013 12:03:18	135
	01/01/2013 12:08:57	133.5
	...	
BAXD	01/01/2013 11:58:32	11.5
	01/01/2013 11:59:42	10.5
	01/01/2013 12:08:30	9
	01/01/2013 12:09:24	9.5
	...	
EFCD	01/01/2013 12:01:17	55
	01/01/2013 12:07:12	54.5
	01/01/2013 12:10:32	55
	01/01/2013 12:19:14	55.5
	01/01/2013 12:20:40	56
	...	
FAMI	01/01/2013 11:57:19	228
	01/01/2013 12:09:45	225
	01/01/2013 12:15:48	227
	01/01/2013 12:19:55	27.5
	01/01/2013 12:22:09	228.5
	...	

FIGURE 7
Storing stock price data in a wide column family

In this column family, the stock ticker symbol is the row key, and the columns themselves consist of the date and time that a price change occurred as the column name, and the new price as the value. If the columns in the **StockPrices** column family are sorted, the prices are listed in date/time order (remember that this might happen automatically, depending on the way in which the column-family database stores the data for columns). As new prices are captured for a stock item, they can be quickly added as a column to the row for the stock item. Similarly, because each row holds the entire stock price history for an item, an application can quickly retrieve the historical data for any given stock.

Some column-family databases also enable you to add an expiration time or *time to live* (TTL) to a column. When this time elapses, the column can be removed (the database does this in the background). TTLs are useful if the data in a column has a natural lifetime beyond which it is no longer relevant or useful. For example, in the **StockPrices** column family, the second-by-second movements in the value of a stock become less important as time passes, and a stock market analyst is going to be more interested in the trend in the value of that stock over a weekly or monthly period. An application that captures and stores the prices could specify a TTL for each price of two months. At the end of each month, the application could sweep through the database and archive summary stock price information (maybe just capture the daily or weekly prices to a different data store) for prices that are due to expire in the following month.

Indexing Columns

Each row in column-family database is identified by using a unique key, and all column families that contain data for a row share the same key value. Keys are automatically indexed and rows in a column family are ordered by the key values. It is therefore very quick for an application to retrieve data by using the row key and you should pay special attention to the values that you use for row keys. For example, if your data has a natural unique identifier (such as a social security number in a personnel database), and your applications regularly query data by using this identifier, then use this identifier as the row key.

In many circumstances, it is advisable to avoid using a monotonic increasing value as the row key in a column-family database as this can lead to hotspots and I/O contention resulting in poor scalability. For example, if you are building a database for a retail application that creates and stores information about orders using the order number as the row key, and the order number is generated by using a monotonic increasing sequence (the first order is order 1, the second is order 2, and so on), then the most recent orders are all likely to be stored together in the same physical region of the database. These are the orders that are most likely to be the focus of attention and will be accessed frequently by users running the application. As more orders are created, the focus shifts, but the phenomenon is of users hitting one region in the database very hard before moving on to the next. In these circumstances, it is better to generate a random (and unique) value such as a GUID for the row key and store the order number as a column value.

Some column-family databases provide the option to hash the row keys, which can help to avoid hotspots in your column families but at the cost of losing the ordering of rows. This mechanism is highly suitable for applications that perform point queries (queries that need to retrieve data from a single row quickly), but it can slow the performance of range queries (queries that fetch a contiguous set of rows) as logically adjacent rows are physically distant from each other.

However, an important strength of a column-family database is the ability to query data based on the values in the various columns other than the row key. As an example, in the census database, a query such as "What is the most common occupation for women in their 20s?" requires that an application can quickly find all rows in the **Profile** column family where the value in the **DOB** (Date Of Birth) column falls within a given range and the value in the **Gender** column is **Female**, and then extract the value in the **Occupation** column. To support this type of processing, many column-family databases enable you to create secondary indexes over columns (distinct from the primary index over the row key).

A secondary index spans the data held by one or more columns in a single column family. In the census database, you could create an index over the **DOB** column in the **Profile** column family to support queries that retrieve data based on the date of birth. In many column-family databases, you can create composite indexes that span more than one column. A composite index can further improve the speed of a query if it covers the entire set of columns required by that query. For example, if you create a composite index that spans the **DOB**, **Gender**, and **Occupation** columns (in that order), an application that needs to determine the most common occupation for women in their 20s can find the information directly from the index and may not need to access the underlying column family.

Indexes have to be maintained as rows are added to and removed from a column family. This can be an expensive and time-consuming operation, especially if the column family contains a very large number of highly dynamic rows. To minimize this overhead, many column-family databases maintain secondary indexes by using background threads that do not block any read or write requests performed by applications running in the foreground. Therefore, it is possible that a query that retrieves data by using an index might not find a row that has been recently added.

The factors that determine whether creating a secondary index over a column will improve query performance depend on the number of distinct values that the column can contain and how the column-family database actually implements secondary indexes. For example, some column-family databases implement indexes as balanced B-Trees (or they use a similar hierarchical strategy). In these cases you should consider creating indexes over columns that have a large number of distinct values. The hierarchical nature of a B-Tree index enables the database to quickly home in on rows where the indexed column contains a specific value.

Figure 8 shows an example of a column family containing 100,000,000 rows of information about car drivers and their licenses. The rows are identified by **DriverID**, but in this scenario it is common for queries to retrieve information by driver's **License** number (for reasons outside the scope of this example, the database designers chose not to use the driver's **License** number as the row key). Without an index, these queries would necessitate scanning through all 100,000,000 rows. If the column-family database provides balanced B-Tree indexes, then it is possible to find a row that contains a specified driver's license number by starting at the root node of the index and following the path where the license number falls in the range of values indicated by the **LicenseRange** data in each node. The leaf nodes of the index contain the row keys. The I/O and processing associated with accessing data through the index should be significantly less than reading all 100,000,000 rows of the data and performing a linear search.

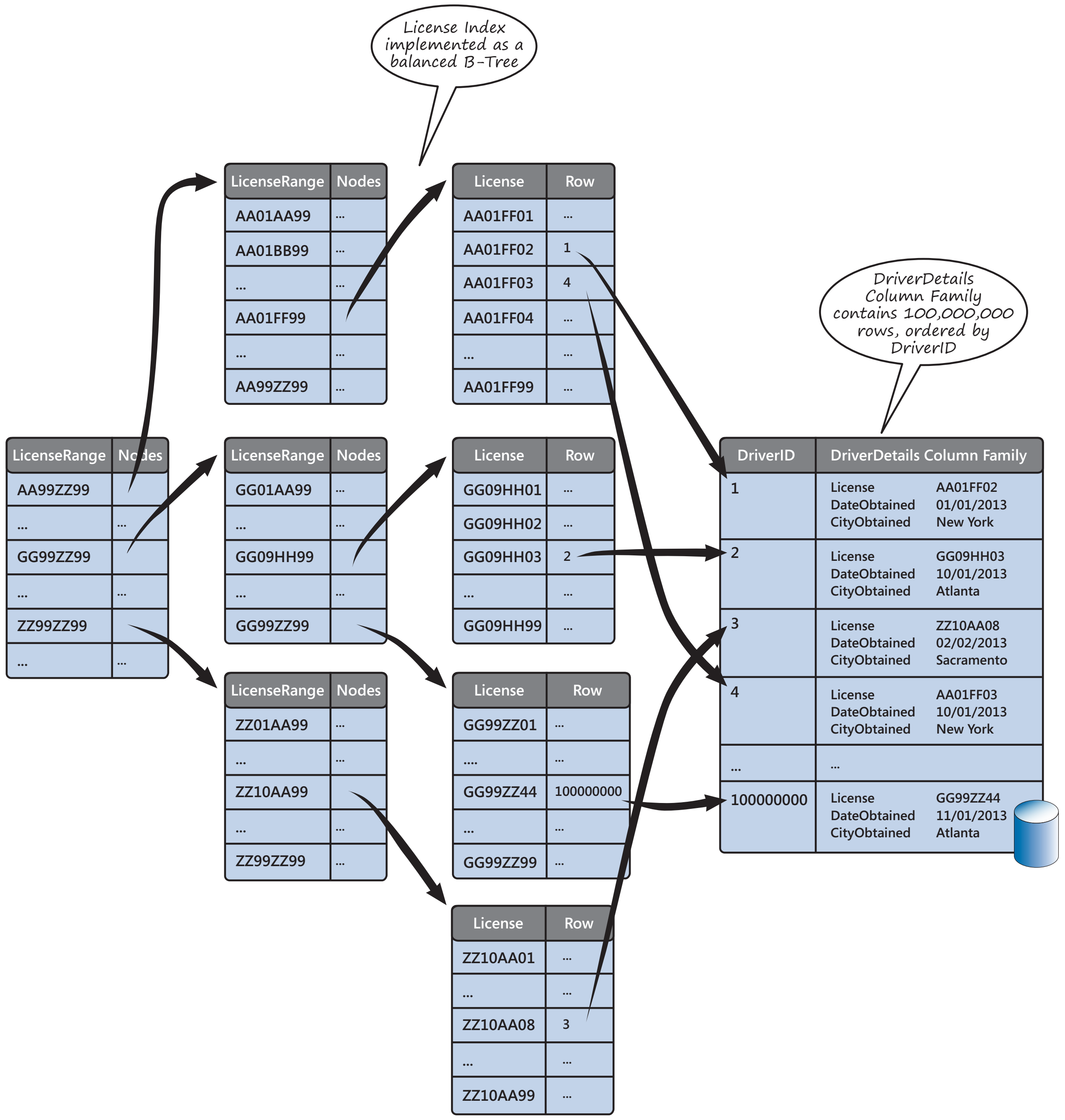
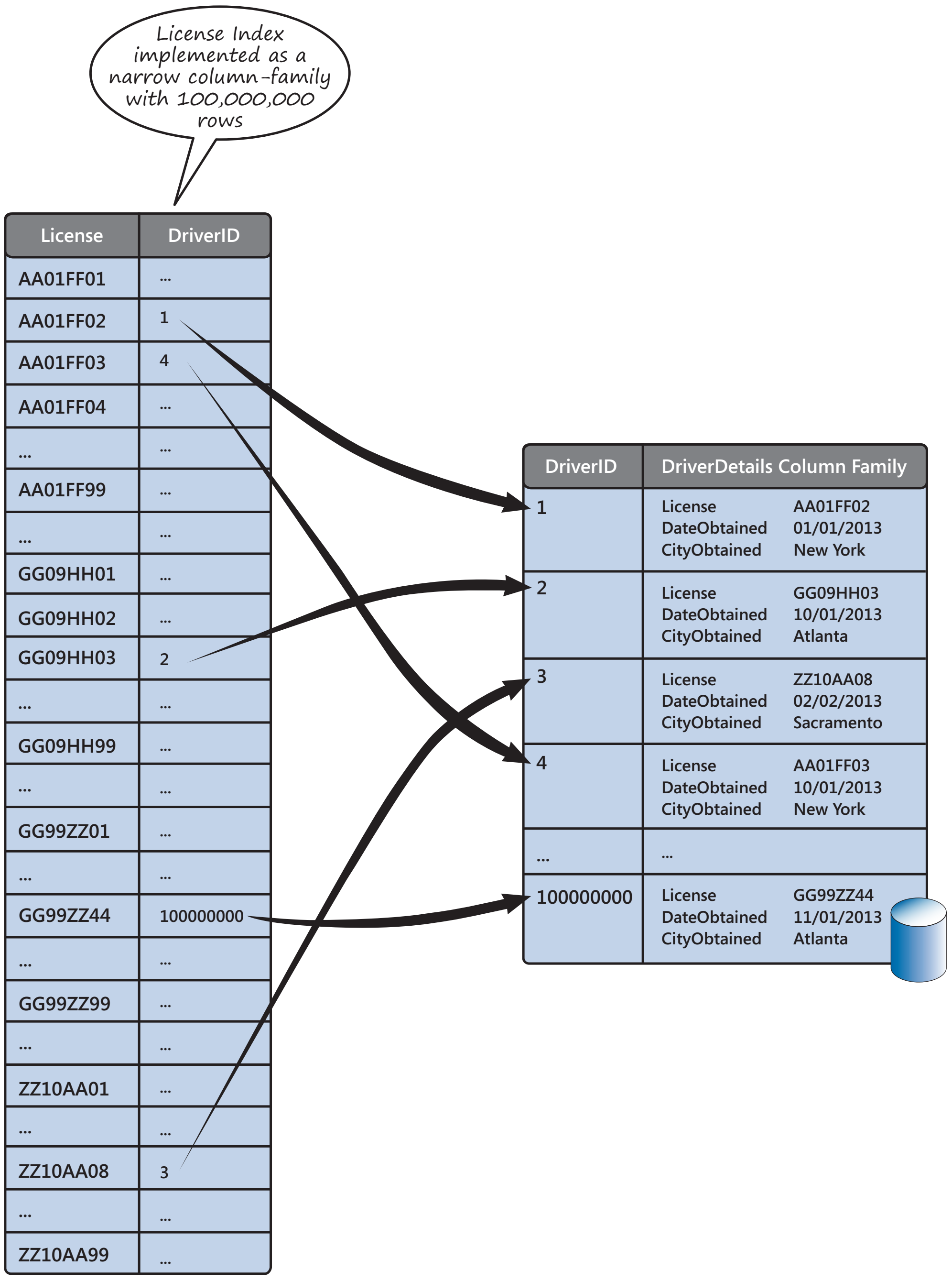

License Index implemented as a balanced B-Tree
DriverDetails Column Family contains 100,000,000 rows, ordered by DriverID
LicenseRange | Nodes
AA99ZZ99 | …
GG99ZZ99 | …
ZZ99ZZ99 | …
LicenseRange | Nodes
AA01AA99 | …
AA01BB99 | …
AA01FF99 |
AA99ZZ99 | …
LicenseRange | Nodes
GG01AA99 | …
GG09HH99 |
GG99ZZ99 |
LicenseRange | Nodes
ZZ01AA99 | …
ZZ10AA99 |
ZZ99ZZ99 | …
License | Row
AA01FF01 | …
AA01FF02 | 1
AA01FF03 | 4
AA01FF04 | …
AA01FF99 | …
License | Row
GG09HH01 | …
GG09HH02 | …
GG09HH03 | 2
GG09HH99 | …
License | Row
GG99ZZ01 | …
GG99ZZ44 | 100000000
GG99ZZ99 | …
License | Row
ZZ10AA01 | …
ZZ10AA08 | 3
ZZ10AA99 | …
DriverID | DriverDetails Column Family
1 | License AA01FF02 DateObtained 01/01/2013 CityObtained New York
2 | License GG09HH03 DateObtained 10/01/2013 CityObtained Atlanta
3 | License ZZ10AA08 DateObtained 02/02/2013 CityObtained Sacramento
4 | License AA01FF03 DateObtained 10/01/2013 CityObtained New York
… | …
100000000 | License GG99ZZ44 DateObtained 11/01/2013 CityObtained Atlanta

Figure 8
A balanced B-Tree index over unique data values

Other column-family databases implement flat indexes, often by creating a separate column family that holds the index data. Remember that column families are especially suitable for retrieving data from wide column families. In this case, indexes over columns that have a relatively small number of different values will work better than indexes over columns that are highly distinct.

Figure 9 shows the **License** index from Figure 8 implemented as a column family. The row keys are the driving license numbers, and the values are the driver IDs. This is a very narrow column family. The index is useful because the data is sorted by driving license number and it is quicker for the database to search through this data than to perform a linear search in the **DriverDetails** column family, but the repeated piecemeal I/O overhead associated with retrieving and searching through a large set of narrow columns might actually be more costly than simply fetching every row from the **DriverDetails** column family.


License Index implemented as a narrow column-family with 100,000,000 rows

License	DriverID
AA01FF01	...
AA01FF02	1
AA01FF03	4
AA01FF04	...
...	...
AA01FF99	...
...	...
GG09HH01	...
GG09HH02	...
GG09HH03	2
...	...
GG09HH99	...
...	...
GG99ZZ01	...
...	...
GG99ZZ44	100000000
...	...
GG99ZZ99	...
...	...
ZZ10AA01	...
...	...
ZZ10AA08	3
...	...
ZZ10AA99	...

DriverID	DriverDetails Column Family
1	License AA01FF02 DateObtained 01/01/2013 CityObtained New York
2	License GG09HH03 DateObtained 10/01/2013 CityObtained Atlanta
3	License ZZ10AA08 DateObtained 02/02/2013 CityObtained Sacramento
4	License AA01FF03 DateObtained 10/01/2013 CityObtained New York
...	...
100000000	License GG99ZZ44 DateObtained 11/01/2013 CityObtained Atlanta

FIGURE 9
An index implemented as a narrow column family over unique values

In contrast, suppose an application needs to query drivers' licenses according to the city that issued them. Figure 10 shows a flat index over the **CityObtained** column in the **DriverDetails** column family (this column contains the city in which the driver obtained their license). In this case, assuming that there are 20,000 city authorities that can issue licenses, each city will account for 5,000 driving licenses on average. Creating an index over this column results in the index utilizing a wide column family. The index is sorted by **CityObtained** (the columns in each index entry may also be sorted by **DriverID**), and an application can quickly use this index to retrieve the list of drivers that obtained their license in a specific city. Furthermore, because the list of drivers for each city is all held in the same row, this data will likely be physically grouped together on disk optimizing the I/O effort required to retrieve the information.

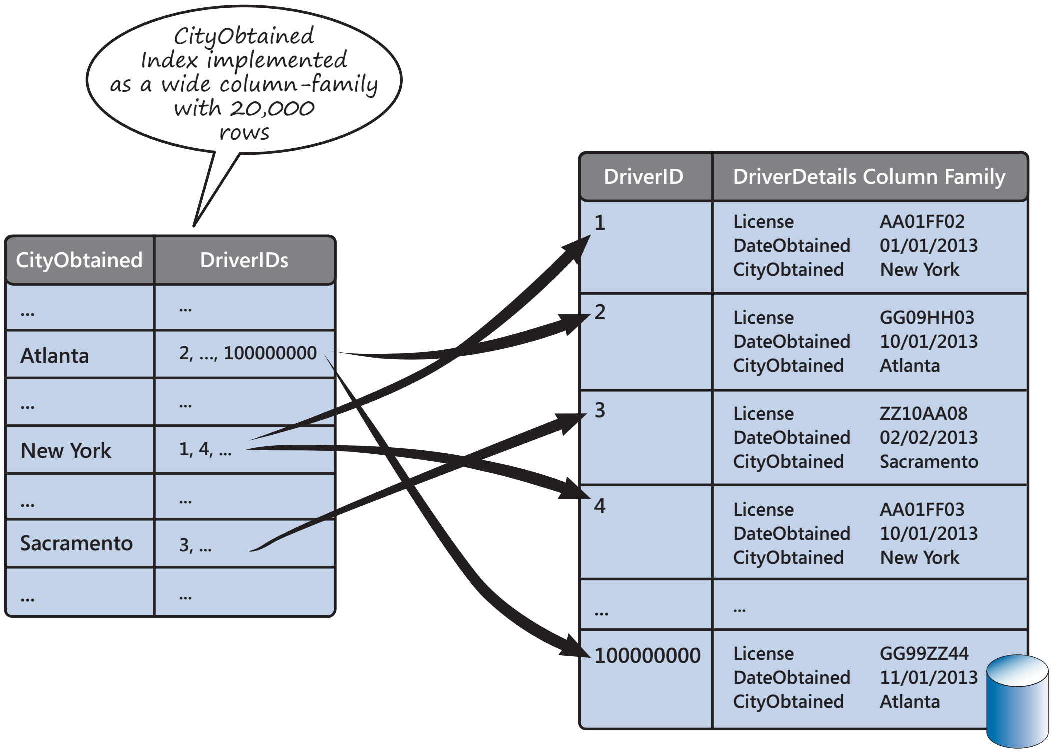


Figure 10
An index implemented as a wide column family over non-unique values

Even with an appropriate index in place, locating and retrieving data can consume some processing power and take a little time. In a large database that has to support many thousands of concurrent users, this effort can add up and become a significant factor that affects the performance of the system. Therefore, many column-family databases also implement Bloom filters to help determine whether a row that matches a specified set of criteria actually exists before attempting to retrieve it.

If your column-family database does not support secondary indexes, you can create your own column families that simulate them. However, it is your responsibility to maintain these indexes as rows are added to and removed from the various column families that they reference.

A Bloom filter is essentially an array of bits and an associated set of hash functions. The bits in the array are all initialized to 0. When the database stores a value in a column family, it hashes a copy of this value by using each associated hash function in turn to calculate a set of locations in the bit array. The database stores a 1 in each specified location in this array. To determine whether the column family contains a specific value, the database runs the same set of hash functions over that value, and uses the results as a collection of indexes into the bit array. If the value at any location is a 0, then there is no matching value in the column family and the search can finish immediately. If the value at every location is a 1, then the column family might contain a corresponding value. However, bear in mind that two values can hash to the same location, so you cannot be sure until you actually retrieve the data, but if you select the hash functions carefully, and have a big enough array of bits, then these cases should be a tiny minority.

Analyzing Data

An index can help an application to locate rows that match a given set of criteria, but many applications also need to analyze the data held in these rows. In the census database query "What is the most common occupation for women in their 20s?," a composite index over the **DOB**, **Gender**, and **Occupation** columns can help to find a list of occupations for females in their 20s, but it does not help the application determine which is the most common occupation. If the column-family database supports a map/reduce framework, you can use this functionality to generate summary information.You write code that extracts the data values from each column that you wish to summarize (the *map* phase), and then perform whatever summary functions you application requires over this data (the *reduce* phase) and store the results in the database. The map/reduce code runs automatically whenever rows are added to or removed from a column family. When your application needs to summarize data, it simply fetches the appropriate summary value from the database.

For more information about using map/reduce frameworks, visit the MapReduce.org website.

Some column-family databases support materialized views that you can use to store more comprehensive summary information. Depending on the way in which the column-family database implements them, a materialized view can either be generated each time it is queried by an application by using code stored in the database, or the data that matches the view can be identified and copied to a separate location when the view is first defined, and as rows are added to or removed from a column family. The first approach is suitable for volatile data, while the second is more efficient if the data is relatively static.

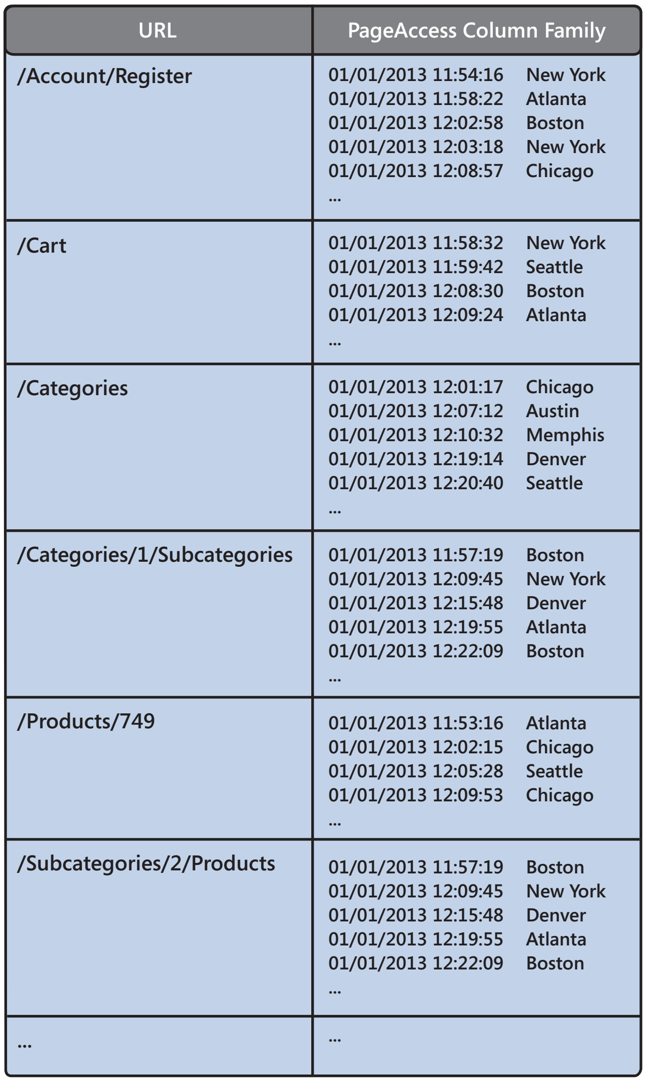
If your column-family database does not support materialized views, you can simulate them by using a separate column family (you can write application code that extracts the data for the view from the database, and then processes and stores the data in the column family.) How up to date you keep the materialized view depends on the nature of the data in the view. Maintaining the information in a materialized view implemented in this way can be an expensive process if the data is dynamic, and it may be sufficient to refresh the view on a daily or even weekly basis rather than every time a change occurs in the underlying data.

Some column-family databases enable you to store and access different versions of data in a column. This can be useful if you need to analyze trends in data, and examine how values have been modified over time.

In these databases, each value can have a version number (sometimes implemented as a date/time field). When you modify the data in a column, the database saves the new value together with a new version number, but leaves the old value in the database as well. In this way, you can think of a column family as being a three-dimensional structure of rows, columns, and versions.

How Adventure Works Plan to Store Information about Website Traffic

To help assess the performance of the Shopping application, and also to gather information about how customers use the application, the developers at Adventure Works wish to analyze the website traffic for their system. This enhancement is not currently implemented, but to perform this analysis, they need to collect information about the location of each user browsing the website and the frequency with which users access each page on the website. They do not need to capture any personal information about users. To support this requirement, the developers designed a column-family database containing the **PageAccess** column family shown in Figure 11.



URL	PageAccess Column Family
/Account/Register	01/01/2013 11:54:16 New York 01/01/2013 11:58:22 Atlanta 01/01/2013 12:02:58 Boston 01/01/2013 12:03:18 New York 01/01/2013 12:08:57 Chicago ...
/Cart	01/01/2013 11:58:32 New York 01/01/2013 11:59:42 Seattle 01/01/2013 12:08:30 Boston 01/01/2013 12:09:24 Atlanta ...
/Categories	01/01/2013 12:01:17 Chicago 01/01/2013 12:07:12 Austin 01/01/2013 12:10:32 Memphis 01/01/2013 12:19:14 Denver 01/01/2013 12:20:40 Seattle ...
/Categories/1/Subcategories	01/01/2013 11:57:19 Boston 01/01/2013 12:09:45 New York 01/01/2013 12:15:48 Denver 01/01/2013 12:19:55 Atlanta 01/01/2013 12:22:09 Boston ...
/Products/749	01/01/2013 11:53:16 Atlanta 01/01/2013 12:02:15 Chicago 01/01/2013 12:05:28 Seattle 01/01/2013 12:09:53 Chicago ...
/Subcategories/2/Products	01/01/2013 11:57:19 Boston 01/01/2013 12:09:45 New York 01/01/2013 12:15:48 Denver 01/01/2013 12:19:55 Atlanta 01/01/2013 12:22:09 Boston ...
...	...

Figure 11
Storing information about website traffic

The **PageAccess** column family records the location from which each page was accessed, using the date and time that the page was read as the column name. The column family stores the data for each page as a separate row, keyed by using the relative URL of the page within the website.

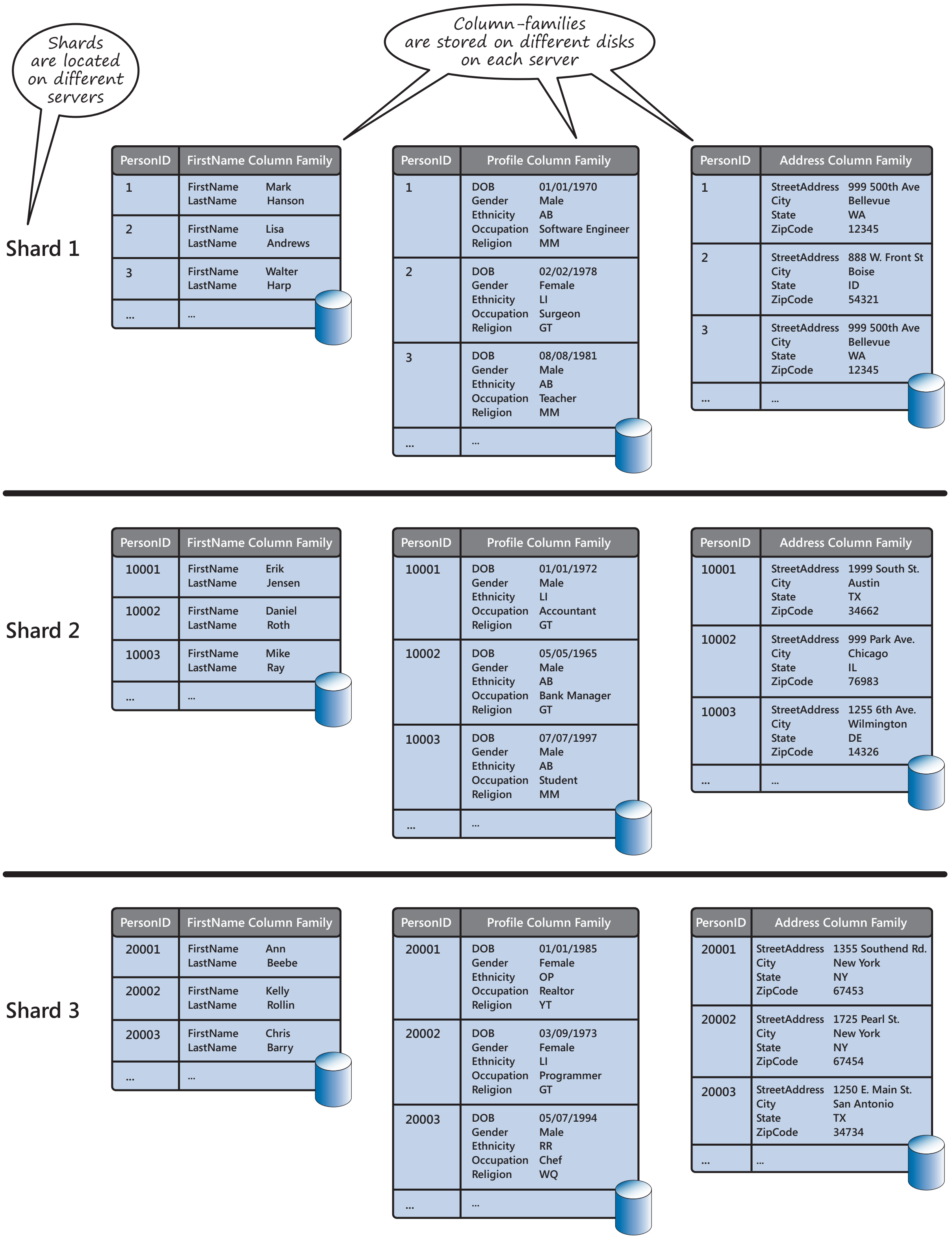
> *The functionality that captures this information is a future development and is not included in the sample application provided with this guide*

Implementing a Column-Family Database to Maximize Scalability, Availability, and Consistency

Historically, column-family databases were explicitly designed to store large amounts of data and enable applications to read and process this data very quickly. As a result, they are inherently scalable and provide high availability, but often at the cost of immediate consistency. This section summarizes the ways in which most column-family databases implement scalability and availability, and describes the tradeoffs commonly made against consistency to help maximize performance.

Maximizing Scalability

In common with most NoSQL databases, the underlying structure of the majority of column-family databases lend themselves to partitioning, both vertically by placing different column-families on different servers, and horizontally by sharding a column family based on the row key. For example, in the census database described earlier in this chapter, it is possible to shard the rows and distribute the data horizontally across servers, based on the **PersonID** key. Additionally, each shard could be partitioned vertically, and the data for the **Name**, **Address**, and **Profile** column families for each shard could be stored on separate disks to reduce I/O contention, as shown in Figure 12. In some cases, it may be possible to distribute the column families for a single shard across multiple servers, spreading the load and increasing scalability even further.


Shards are located on different servers
Column-families are stored on different disks on each server
Shard 1
Shard 2
Shard 3

Shard 1

PersonID	FirstName Column Family	
1	FirstName	Mark
	LastName	Hanson
2	FirstName	Lisa
	LastName	Andrews
3	FirstName	Walter
	LastName	Harp
...	...	

PersonID	Profile Column Family	
1	DOB	01/01/1970
	Gender	Male
	Ethnicity	AB
	Occupation	Software Engineer
	Religion	MM
2	DOB	02/02/1978
	Gender	Female
	Ethnicity	LI
	Occupation	Surgeon
	Religion	GT
3	DOB	08/08/1981
	Gender	Male
	Ethnicity	AB
	Occupation	Teacher
	Religion	MM
...	...	

PersonID	Address Column Family	
1	StreetAddress	999 500th Ave
	City	Bellevue
	State	WA
	ZipCode	12345
2	StreetAddress	888 W. Front St
	City	Boise
	State	ID
	ZipCode	54321
3	StreetAddress	999 500th Ave
	City	Bellevue
	State	WA
	ZipCode	12345
...	...	

Shard 2

PersonID	FirstName Column Family	
10001	FirstName	Erik
	LastName	Jensen
10002	FirstName	Daniel
	LastName	Roth
10003	FirstName	Mike
	LastName	Ray
...	...	

PersonID	Profile Column Family	
10001	DOB	01/01/1972
	Gender	Male
	Ethnicity	LI
	Occupation	Accountant
	Religion	GT
10002	DOB	05/05/1965
	Gender	Male
	Ethnicity	AB
	Occupation	Bank Manager
	Religion	GT
10003	DOB	07/07/1997
	Gender	Male
	Ethnicity	AB
	Occupation	Student
	Religion	MM
...	...	

PersonID	Address Column Family	
10001	StreetAddress	1999 South St.
	City	Austin
	State	TX
	ZipCode	34662
10002	StreetAddress	999 Park Ave.
	City	Chicago
	State	IL
	ZipCode	76983
10003	StreetAddress	1255 6th Ave.
	City	Wilmington
	State	DE
	ZipCode	14326
...	...	

Shard 3

PersonID	FirstName Column Family	
20001	FirstName	Ann
	LastName	Beebe
20002	FirstName	Kelly
	LastName	Rollin
20003	FirstName	Chris
	LastName	Barry
...	...	

PersonID	Profile Column Family	
20001	DOB	01/01/1985
	Gender	Female
	Ethnicity	OP
	Occupation	Realtor
	Religion	YT
20002	DOB	03/09/1973
	Gender	Female
	Ethnicity	LI
	Occupation	Programmer
	Religion	GT
20003	DOB	05/07/1994
	Gender	Male
	Ethnicity	RR
	Occupation	Chef
	Religion	WQ
...	...	

PersonID	Address Column Family	
20001	StreetAddress	1355 Southend Rd.
	City	New York
	State	NY
	ZipCode	67453
20002	StreetAddress	1725 Pearl St.
	City	New York
	State	NY
	ZipCode	67454
20003	StreetAddress	1250 E. Main St.
	City	San Antonio
	State	TX
	ZipCode	34734
...	...	

Figure 12
Sharding and partitioning the data for the census database

If possible, select the partitioning mechanism that your database uses to shard data to match the type of operations that your application performs. If your application uses range queries extensively, partition data by row key. If your application is insert intensive, then partition data by hashing the row key.

Different column-family databases provide varying strategies for determining in which shard the data for a particular row is stored. By default, many column-family databases enforce a strict ordering, so that the data for row 2 is placed after row 1, and the data for row 3 is placed after row 2, and so on. This mechanism populates a single shard, and then overflows onto the next when the first shard has reached its maximum capacity. The rationale behind this strategy is that it preserves the key order of adjacent rows and optimizes applications that perform range queries based on the row key. If you need to analyze and summarize data, using a map/reduce framework can help to optimize calculations by isolating updates to summary values to the shard in which rows are inserted; the summary information in other shards will remain unaffected. However, as described earlier, storing data in row key order can lead to hotspots occurring. In this case, a single shard becomes the focus of all new inserts and this can result in that shard having to handle a disproportionate volume of requests while other shards remain relatively inactive. For this reason, some column-family databases hash the row key, effectively distributing the data randomly across shards. This mechanism spreads the load, but can reduce the performance of applications that perform range queries.

Most column-family databases implement auto sharding, where the sharding strategy and location is transparent to the applications storing and retrieving data. This approach requires configuring each server with information that it can use to locate other servers participating in the sharding cluster. An application can send requests to store to any servers in the cluster, and the servers negotiate with each other to determine where the data should be saved. Similarly, an application can send requests to retrieve data to any server, which communicates with the other servers to determine the location of the data and return it to the application. In this scheme, an administrator can scale the database out to additional servers simply by installing the database system software on these servers and adding the details of these servers to the cluster configuration. If the column-family database software supports automatic detection, new servers may be discovered by existing servers in the cluster.

Ensuring Availability

To prevent data loss and help ensure that the entire database is always available, most column-family databases implement replication. Each server holding one or more partitions should be replicated, and the servers in a single replication cluster should ideally be located on different sites.

Due to the size of the rows (remember that a single row can contain many hundreds of columns), most column-family databases adopt the peer-to-peer replication strategy. In the alternative primary/secondary replication model, the primary server would be responsible for transmitting every insert to all other servers in the replication cluster. Each row could easily be several megabytes or even gigabytes in size, and this approach could result in the primary server becoming a bottleneck. The peer-to-peer model divides the responsibility across all servers in the replication cluster. However, inserts, updates, and deletes may take a little longer to propagate across the entire cluster (the servers may use the Gossip Protocol to share information with each other periodically) and there is consequently more scope for inconsistency.

To optimize insert and update operations many column-family databases attempt to cache changes in memory and use append-only forward logging to record the changes (this form of I/O is much faster than performing the various seek and write operations required to save a row to a specific location in the file for a column family). If a new or modified row has to be flushed from cache, then the data can be persisted to the column family on disk.

To ensure that delete operations are performed quickly, many column-family databases use *tombstoning*. The row is not actually deleted, but simply marked as *no longer present*. A compaction process can remove tombstoned rows as a batch job when the column family needs to be defragmented or cleaned-up.

Maximizing Consistency

In common with other types of NoSQL databases, to maximize performance and scalability most column-family databases only provide a very limited form of transactional consistency. In nearly all column-family databases, only write operations that affect a single row in a single column family are atomic. However, depending on the database software, you might also find that you can group writes that affect multiple rows in a single column family together as an atomic operation (as long as all the affected rows are located within the same shard), and you might even be able to write the data for a single row that spans multiple column-families as a transaction (again, if the column-families are all located on the same server).

Some column-family databases provide row-level locking; an application can lock an entire row to prevent it from being changed by another application. However, row locking also requires that locks are released in a timely manner otherwise concurrency may be severely limited,

If the database implements caching and forward-only logging as described in the previous section, a write operation completes successfully only when the log record for that operation has been saved to disk.

If you require transactional consistency across column-families and rows, you may need to implement this functionality as part of the application code that accesses the database or use a third-party library synchronization service. Chapter 8, "Building a Polyglot Solution," provides more information.

If you are replicating data to ensure availability, the peer-to-peer replication model typically depends on the servers in a cluster being grouped into appropriate read and write quorums to maximize consistency. However, if applications connect to different servers in a cluster to read and write data, write conflicts are highly probable because the more servers that have been added to a replication cluster, the more likely it is that write conflicts will occur. For this reason, many column-family databases also implement vector clocks or other forms of versioning information that can be used to detect and resolve conflicts. Several popular column-family databases automatically add timestamp information to individual columns, and automatically detect and resolve conflicts (the database designer may be able to specify the strategy to use to resolve conflicts). To maximize the performance of write operations, conflict detection and resolution is often deferred and only performed when an application reads data (or data is exchanged between servers during the Gossip Protocol). The rationale behind this approach is that if an application writes conflicting data, but that data is never read subsequently, then there is no need to detect or resolve the conflict.

The section "Improving Consistency" in Chapter 1 provides more information about how read and write quorums work, and the common forms of versioning strategies that many NoSQL databases use.

Summary

This chapter has described how to use a column-family database to store and retrieve structured information. You have seen how the data is organized as a tabular structure with data divided into rows and the columns for each row spanning one or more column families. The data for a row in a column family can comprise hundreds or even thousands of columns, and the structure of different rows in the same column family can vary (different rows do not have to contain the same columns).

Column families define the logical and physical structure of the database; the data for a single column family is typically stored together in the same file, and different column families are contained within different files. You should define the column families for a database to optimize the I/O performed by the database.

Each row is identified by a row key, and all rows for an entity that spans multiple column families share the same row key. The row key is usually indexed, enabling an application to retrieve information by using the row key very quickly. Most column-family databases also enable a developer to create indexes over the other columns in a column family, speeding the performance of queries that reference these columns as query criteria.

In common with most other NoSQL databases, column-family databases support sharding, enabling the rows in a column family to be distributed across servers to spread the load and improve scalability. Column families also provide natural vertical partitioning, and you can place different column families on different physical disks, again to spread the load and improve performance.

Many column-family databases also support replication to help ensure that the data for each column family is always available. Column-family databases tend to provide minimal transactional consistency and in most cases they only support atomic writes for operations that store data in a single column family for a single row. Eventual consistency is typically provided by using read quorums and row versioning with conflict resolution.

More Information

All links in this book are accessible from the book's online bibliography available at: *http://msdn.microsoft.com/en-us/library/dn320459.aspx*.

- You can find more information about using map/reduce techniques to summarize data on the MapReduce.org website at *http://mapreduce.org/*.
- MSDN Magazine published a series of articles on getting started with Cassandra, a popular column-family database. You can find these articles online at:
  - Cassandra NoSQL Database: Getting Started: *http://msdn.microsoft.com/magazine/jj553519.aspx*
  - Cassandra NoSQL Database, Part 2: Programming: *http://msdn.microsoft.com/magazine/jj658980.aspx*
  - Cassandra NoSQL Database, Part 3: Clustering: *http://msdn.microsoft.com/magazine/jj721601.aspx*.

7 Implementing a Graph Database

The driving force behind most NoSQL databases is to enable you to store information in a large scale, highly available, and optimized database, and then retrieve this information again extremely quickly. The focus of these databases is on putting data in and getting it out again in as efficient a manner as possible. Graph databases operate in a different functional space; their main aim is to enable you to perform queries based on the relationships between data items write application that can analyze these relationships.

If you store information in a relational database, you can use SQL to perform complex and intricate data analyses, but the code for these SQL statements can sometimes be difficult to develop and test, cryptic to maintain, and require considerable resources to run. For this reason, many relational database vendors also offer data warehousing solutions that enable a system to take the data held in a relational database and restructure this data into a form optimized for performing data analysis. However, many such solutions can be expensive, and they often require that you run them on powerful hardware that is tuned to support the demands of the data warehouse software.

A graph database enables you to take a different approach. Rather than defining your data as a series of entities or records and implementing costly run-time processes to deduce the relationships between them, you explicitly design the database around the relationships that your application requires. This strategy enables you to optimize the database for your application. Starting at a given point, your application can easily traverse the network of connections between entities without requiring that you write complex code. Additionally, most graph databases do not require lavish hardware, often running on clusters of commodity computers.

This chapter describes how to design a graph database to support the analytical processing performed by an application, and discusses how Adventure Works used a graph database to implement product recommendations for customers buying goods using the Shopping application.

What is a Graph?

In the world of graph databases, a graph is a connected network of nodes. Each node contains information about a specific entity, and each connection specifies a relationship between entities.

A graph can contain many different types of nodes, and a node can contain properties that store information about that node. For example, in a database that models the departmental structure of an organization, you might create nodes for departments and nodes for employees that work in these departments. The properties of a department node could include the department name and location, while the properties of an employee node could include the employee's name, payroll number, and other relevant details.

In theory, you could implement a graph database using almost any technology, including tables in a relational database. In a relational database you could create tables to hold the information for the different types of entities and use SQL to perform queries that join rows retrieved from multiple tables. This approach is very flexible from a query perspective but it can consume considerable resources at runtime (joining tables is a costly operation) and tends not to scale well. Additionally, if business requirements change and you need to restructure the information that you store about entities, or even add new types of entities, the schema-oriented nature of a relational database can make it difficult to accommodate these changes.

When you query a graph database, you specify a node that acts as the starting point and then traverse the relationships from that node. The performance of any given query does not depend on the overall size of the database, only the number of nodes that the query finds. Queries can run in near-linear time even if the volume of data in the database increases exponentially.

NoSQL graph databases are specifically designed to store information about the relationships between nodes, and enable applications to use this information to traverse a graph extremely efficiently. In a graph database, each connection is directional; it has a start point (a node) and an end point (another, or possibly the same node), and the direction of the connection determines how an application can traverse from the start point to the end point. Additionally, connections can have properties that provide additional information about the relationship between the nodes at the start and end points. These are an important feature because they enable your application to quickly filter out paths that are unnecessary or irrelevant for a specific query. Furthermore, like most NoSQL systems, graph databases do not enforce any form of rigid schema, enabling you to easily incorporate additional types of entities into your solutions without requiring that you restructure the database.

Figure 1 shows an example graph database implementing a refinement of the departmental structure for the fictitious organization first presented in Chapter 1, "Data Storage for Modern High-Performance Business Applications." This example separates out the **Works For** and **Manages** connections because these are actually separate explicit relationships in a graph database (in a relational database, these connections would be a single bidirectional relationship that could be inferred by using foreign and primary key fields.)

Additionally, this example includes the **Employs** relationship that indicates the departments in which each employee works. This relationship enables the database to support queries such as "Who works in Marketing?" without having to fetch each employee and examine the **Works In** relationship.

Finally, this example also defines the property **Worked In Since** for the **Works In** relationship, which specifies the starting date of an employee within a department (if you were using a relational database, you would probably need to store this information in a separate table, and any queries would need to join with data from this table). This property belongs to the relationship rather than an **Employee** node, and this structure enables the database to model the situation where a single employee works for more than one department (2 days a week in Sales, and 3 days a week in Marketing, for example).

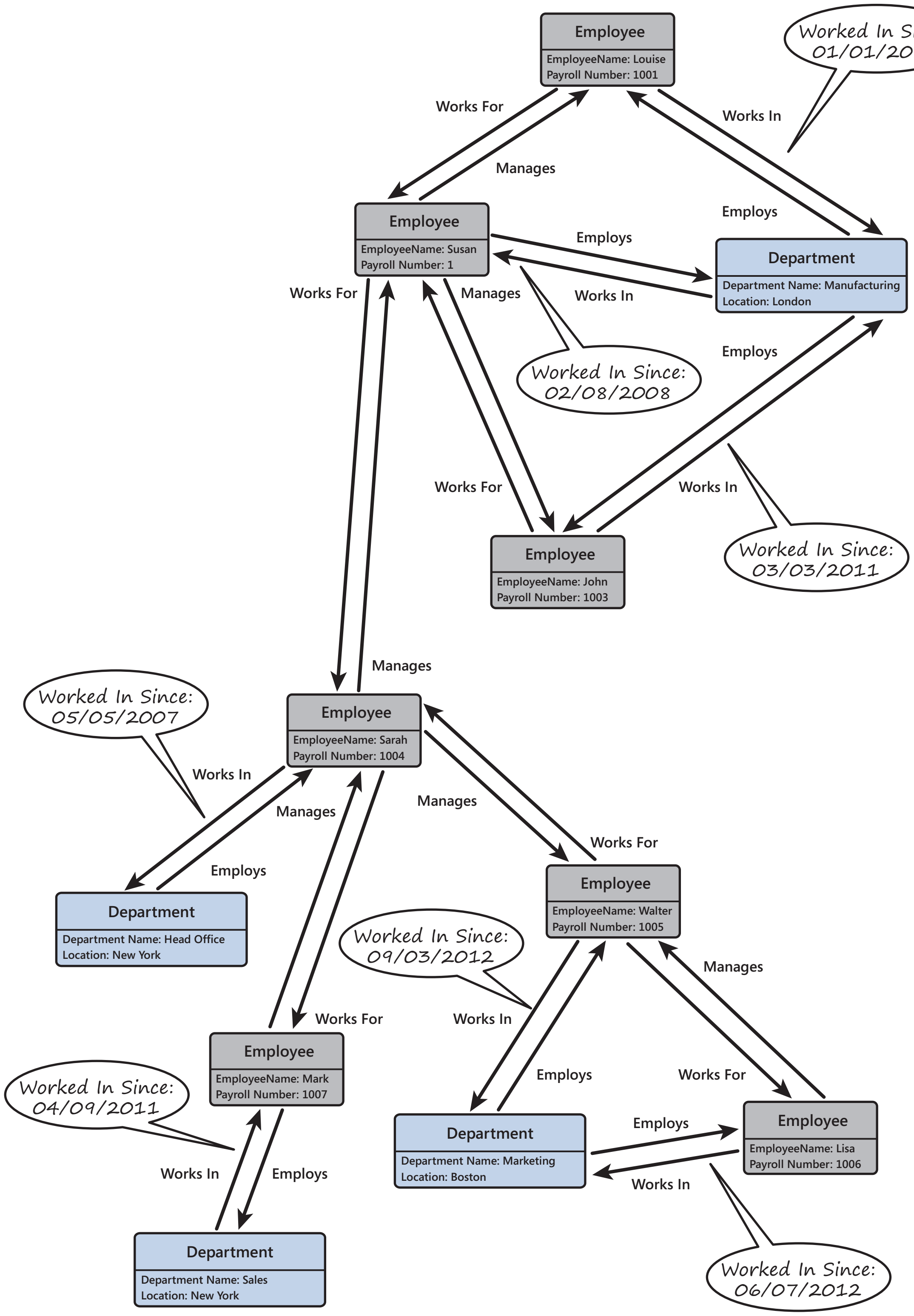


Figure 1
A departmental organization chart, structured as a graph database

Designing a Graph Database

Clearly there are two primary aspects that you need to consider when you design a graph database:

- The nodes that your application needs to retrieve and process, together with the relevant properties for these entities.
- The relationships between these entities and the properties of these relationships.

You also need to give careful thought to the mechanics of how the queries that your application performs will use your graph database. All queries must specify a starting point, or query root, and your graph database must enable an application to quickly find the node that acts as the query root for any given query.

Finally, while there are some scenarios that are eminently suited to graph databases, there are other situations that are best handled by using another form of database. You should not attempt to force the data for an application into a graph structure simply because you have a graph database available.

The following sections discuss these concerns in more detail.

Modeling Data as a Graph

When you design a graph database, try to avoid thinking in terms of tables and rows, and instead concentrate on the queries and operations that your application needs to perform. Examine the business use cases and try to identify the questions that these uses cases ask of your database, such as "*In which department does <employee name> work?*," or "*Who does <manager name> manage?*" This process can help to identify the most common relationships (*"Works In," "Manages"*) that the database needs to support. You can then analyze these relationships to determine the objects on which they should operate (*Employees, Departments, Managers*). You should also be aware that some of these objects might actually be instances of the same type of data (*Employees* and *Managers*, for example), and that it is their role in a relationship that determines how the application uses them. As part of this process you will probably discover other relationships that you should model, such as "*Works For*" and "*Employs*."

> *Some relationships might be bidirectional, while others might only be unidirectional. For example,* Manages *is clearly a unidirectional relational. A manager manages an employee, but the converse is not true. An example of a bidirectional relationship could be "*Colleague Of*" for two employees that work in the same department. If employee A is a colleague of employee B, then employee B will also be a colleague of employee A.*

The queries that your application needs to perform will also help to define the properties that the nodes and relationships should include. In the organization example in Figure 1, the application that uses this data displays employee names and identifies employees by using their unique payroll number, so these attributes are included as properties. You should note that most graph databases do not provide the richness for defining the structure of data in a node that other types of NoSQL databases supply. For example, you might be limited to using simple types such as strings, numbers, and dates, and you might not be able to create complex denormalized structures such as nested documents or handle large binary data values. If you need to store and retrieve highly-structured information you can build a polyglot system; save the basic, essential information about entities as nodes in a graph database, and store the remainder in a more appropriate NoSQL database such as a document database. The information that you store in the graph database should include a key that the application can use to quickly lookup the complete document in the document database. Chapter 8, "Building a Polyglot Solution," describes this scenario in more detail.

Don't try and store highly structured documents in a graph database.

You should pay special attention to the properties that you add to relationships because they can have a significant impact on the performance of the queries performed against the database. When you perform a query, a graph database allows you to qualify the relationships that the query should navigate across by using these properties, enabling the database server to quickly avoid traversing paths that are not relevant to the query and effectively pruning the tree of nodes that it needs to search.

In general, you should design the graph database to optimize the queries performed by the application. Focus on the relationships as these will be the most important items that influence the performance of the database and your application.

Retrieving Data from a Graph Database

All graph databases provide a means to enable an application to walk through a set of connected nodes based on the relationships between these nodes. The programmatic interfaces that graph databases provide vary from vendor to vendor, ranging from the simple imperative approach through to more declarative mechanisms. In the simple imperative approach you select a node as a starting point, examine the relationships that this node has with other nodes, and then traverse each relevant relationship to find related nodes, and then repeat the process for each related node until you have found the data that your require or there is no more data to search. In the more declarative approach you select a starting point, specify criteria that filters the relationships to traverse and nodes to match, and then let the database server implement its own graph-traversal algorithm to return a collection of matching nodes. If possible, adopt the declarative approach because this mechanism can help to avoid tying the structure of your code too closely to the structure of the database.

Neo4j: Cypher Query Examples

As an illustration of the declarative approach, this section contains some examples based on the Cypher query language implemented by Neo4j, a popular NoSQL graph database.

This section is not intended to explain how the Cypher query language works. If you require more information, review the Neo4j documentation available online at The Neo4j Manual.

The examples are based on the organization graph shown in Figure 1. Neo4j assigns a unique identity number to each node and relationship when you create them. You can specify the identity number of a node (or nodes) as the starting point for a Cypher query. The examples that follow assume that the employee nodes have the following identity numbers:

1: Sarah
2: Walter
3: Mark
4: Lisa
5: Louise
6: Susan
7: John

The following Cypher query retrieves the nodes for all the employees that report directly to Sarah. The query starts at node 1 (Sarah), and follows all instances of the **Manages** relationship from that node to find the employees connected directly to node 1:

```
Cypher
START manager=node(1)
MATCH manager-[:Manages]->employee
RETURN employee
```

This query returns a list containing nodes 2 (Walter), 3 (Mark), and 6 (Susan).

The next example shows how to find all employees that report directly or indirectly to Sarah. In this case, the Cypher query uses the "*" modifier to specify that it should traverse all instances of the **Manages** relationship not just for Sarah, but for every node that it finds connected to Sarah, and for each of the nodes connected to these nodes, and so on:

```
Cypher
START manager=node(1)
MATCH manager-[:Manages*]->employee
RETURN employee
```

This query returns a list containing every employee node except for employee 1 (Sarah does not manage herself).

You can specify multiple starting points for a query by providing a comma-separated list of nodes in the START clause. You can also specify the wildcard character "*" as a shorthand for all nodes (this is not recommended if you have a large database). The following example shows how to find the department in which each employee works by using the **Works_In** relationship:

```
Cypher
START employee=node(*)
MATCH employee-[:Works_In]->department
RETURN department
```

You can also filter data by using the properties of nodes and relationships. If you wish to limit the previous query to employees that work in departments located in New York, you can perform a query such as this:

```
Cypher
START employee=node(*)
MATCH employee-[:Works_In]->department
WHERE department.Location=”New York”
RETURN employee
```

This query returns node 1 (Sarah), and node 3 (Mark). These employees work in different departments (Head Office and Sales), but both departments are based in New York.

If you need to find out which employees work in a specific department, such as Manufacturing (assume that this department has the identity number 11 in the graph database shown in Figure 1), you can make use of the Employs relationship:

```
Cypher
START department=node(11)
MATCH department-[:Employs]->employee
RETURN employee
```

This query returns a list containing nodes 5 (Louise), 6 (Susan), and 7 (John).

Indexing Nodes and Relationships

All queries against a graph database require a node (or a collection of nodes) as a starting point. Rather than referencing nodes through their identity numbers, you typically select nodes by specifying a value for one or more properties. For example, in the organization database, if you want to find all the employees that report directly to a particular manager, you would provide a property value (such as the payroll number) that identifies that manager as the starting point.

In most graph databases, you can define indexes over properties to enable the database server to quickly locate a node based on the values of these properties. In particular, you should create indexes over each of the properties that you use to specify the starting criteria for the searches that your application performs. In the organization example, if you wish to retrieve the employees that work for the manager with the **Payroll_Number** property of 1004, you should create an index over this property. If you are using Neo4j and the Cypher query language, you can then use the following code to retrieve the employees (this example assumes that the index is called **Payroll_Number_Index**):

```
Cypher
START manager=node:Payroll_Number_Index(Payroll_Number="1004")
MATCH employee-[:Works_For]->manager
RETURN employee
```

The indexing capabilities of a graph database vary from vendor to vendor. In some cases, the database might maintain indexes automatically, while in others it may be necessary to manually add references to nodes directly to an index as new nodes are inserted into the database. In the latter case, it is important that your application code keeps the indexes up to date otherwise the queries that the application performs might return inaccurate data, or they could omit data that it should have returned.

You may also find indexes beneficial over properties used to filter data in queries (you can think of an index as a sorted key/value store where the value references the node, and an index can quickly help to identify a set of nodes with a property value that falls into a given range). However, you should avoid creating indexes over other properties because they will be unlikely to improve the speed of your queries, and may even be detrimental to the performance of the database (these indexes have to be maintained as nodes are inserted, deleted, or modified, and the more indexes you have the greater the maintenance overhead).

You may also be able to define indexes over the properties of relationships. These indexes can prove extremely useful for queries that filter traversal paths based on these properties, especially if a node has a large number of relationships with other nodes.

Some graph databases support full-text indexing. These indexes are useful if your database stores large textual values and you need to quickly find information based on patterns and regular expressions in these values.

Common Use Cases for a Graph Database

Remember that the purpose of a graph database is to store and maintain information about relationships between nodes, and enable applications to traverse these relationships quickly. Common scenarios that match the model implemented by most graph databases include:

- **Social Networking**. A typical social networking database contains a significant number of contacts together with the connections that these contacts have with each other, either directly or indirectly. The database may also reference other types of information such as shared documents or photographs (these items might be physically stored outside of the graph database, following a polyglot architecture as described in Chapter 8). The result is a typically a complex network of friends (connections between contacts) and likes (connections between contacts and documents/photographs).
- **Calculating Routes**. You can use a graph database to help solve complex routing problems that would require considerable resources to resolve by using an algorithmic approach. For example, you can easily determine the shortest path between two nodes in a highly connected graph.
- **Managing Networks**. A graph database is an ideal repository for modeling complex telecommunications or data networks. Using a graph database can help to spot weak points in the network, and perform failure analysis by examining what might happen if one or more nodes becomes unavailable.
- **Generating Recommendations**. You can use a graph database as a recommendations engine. For example, in a retail system, you can store information about which products are frequently purchased together. You can use the resulting graph to generate "other customers that bought xyz also bought abc" recommendations when the customer views the details for a product.
- **Mining Data**. You can traverse the relationships in a graph database to investigate the direct and indirect interactions between nodes. These interactions can help you to spot trends in your data and enable you to act on these trends.
- **Determining Security and Access Rights**. Many distributed systems need to be able to query and track the access rights that have been granted to a large number of users over an equally large volume of resources. When an application requests access to a resource on behalf of a user, it is important to be able to resolve this request quickly and accurately. A simple and efficient solution is to create graph database that stores information about users and resources, and implements access rights as the relationships between these entities.

Some graph database servers incorporate common graph traversal algorithms such as *"find the shortest path between nodes," "find all paths between nodes,"* and *"find the cheapest path between nodes"* (implementing the Dijkstra algorithm that adds weighting and costs to the paths between neighboring nodes, so the cheapest path from one node to another is not necessarily the shortest). You specify the node to use as the start and end points, and any parameters that the algorithm requires, and the database server returns a list of paths.

In a typical graph database, neither relationships nor nodes are necessarily static. Graph databases excel at queries that traverse connections between nodes, but they are not designed to support high levels of online transaction processing. So, ideally, the volume updates to nodes should be small compared to the number of queries being run. Performing an update that modifies a large number of unrelated nodes can be expensive because the graph database needs to locate each node; nodes are not stored in the well-defined aggregations common to other types of NoSQL databases. In the organization example shown earlier in this chapter, if the graph database stores employee details as nodes and includes the salary as part of this information, performing an update operation that raises the salary for a large proportion of the employees requires that you find each employee individually (this is a different process from finding nodes that have a relationship with other nodes). As described previously, defining indexes can help to speed up the process of finding nodes that match a specific pattern, but the database will also need to maintain these indexes as nodes are added, modified, or deleted.

Modifications to relationships tend to be less expensive than making changes to nodes, usually due to the nature of the business logic that drives these modifications. In the organization example shown earlier, if a manager moves to a new department, it is relatively easy to traverse the **Manages** relationships for that manager to find all the employees that reported to that manager and assign them to a new manager instead.

If you need to combine the ability to frequently modify the data in nodes with queries that involve complex, dynamic relationships, then use two different NoSQL databases. Store the data as entities in a document or column-family database, and reference these entities from the nodes in the graph database. Chapter 8, "Building a Polyglot Solution," provides an example.

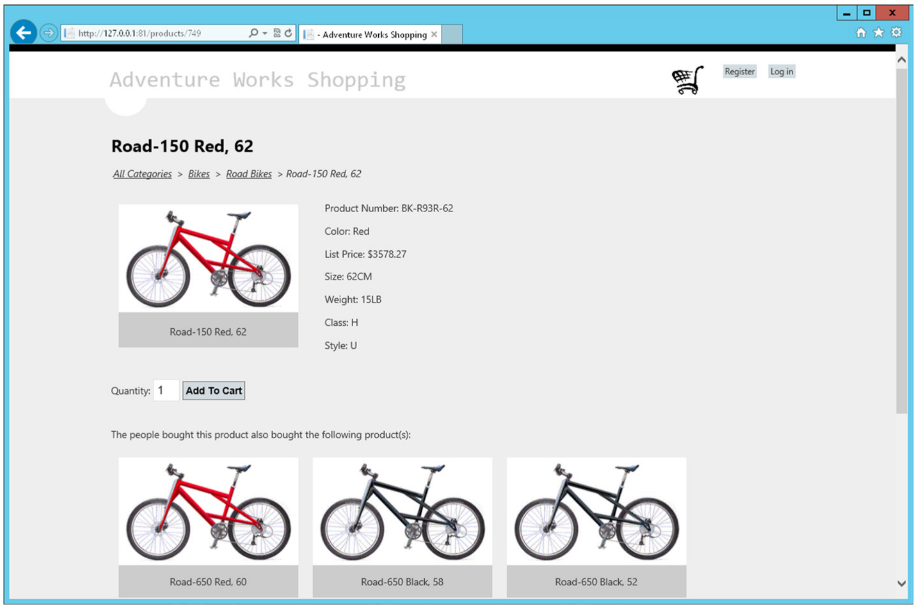
It is also important to realize that a graph database is typically structured to optimize specific queries, and you define the relationships between the nodes in the database that support these queries. This means that a graph database might not be the best type of repository to support truly ad hoc searches; such queries tend to be based on relationships that are *discovered* once the data has been loaded into the database. Similarly, a graph database may not suitable for performing calculations that involve aggregating data unless these aggregations have been predefined by using map/reduce functionality (if the graph database supports it). For example, in the organization database, if you need to calculate the average salary of every employee in the company, or even in a particular department, then you may be able to implement map/reduce functions and store the results directly in the database. However, if you need to calculate results from ad hoc aggregations, such as determining the average salary for every employee who is based in Boston and has started working for the company since 01/01/2013, then storing the information in a relational database or a column-family database might be more efficient.

You can define indexes over properties in nodes and relationships to help speed up direct access to these items. Using these indexes might improve the performance of some ad hoc aggregate calculations by enabling you to iterate through data in a specific sequence rather than traversing a graph, but you should avoid defining too many indexes because the maintenance overhead that they incur may slow your database down.

Some graph databases provide built-in query facilities for calculating simple aggregate values such as Count, Sum, Average, Max, and Min, without writing your own map/reduce functions. However, it can still be expensive to use these features in an application.

How Adventure Works Used a Graph Database to Store Product Recommendations

In the Shopping application, when a customer views the details of a product, the application displays a list of items that other customers who bought this product also purchased. Figure 2 shows an example.



Road-150 Red, 62

All Categories > Bikes > Road Bikes > Road-150 Red, 62

Product Number: BK-R93R-62
Color: Red
List Price: $3578.27
Size: 62CM
Weight: 15LB
Class: H
Style: U

Road-150 Red, 62

Quantity: 1 Add To Cart

The people bought this product also bought the following product(s):

Road-650 Red, 60 | Road-650 Black, 58 | Road-650 Black, 52

Figure 2
The Shopping application showing the details of a product and a list of related recommended products

To keep the list of product recommendations manageable and relevant, the list only contains the five most frequently purchased related products.

It is important to ensure that the application can find recommended products for any given product quickly and easily, so the developers at Adventure Works decided to store product recommendations in a graph database.

The developers selected Neo4j to implement the graph database. However, this choice was simply because this is the graph database with which they are most familiar, and they could have selected one of a number of graph databases from other manufacturers.

Designing the Nodes and Relationships for the Product Recommendations Database

Initially, the developers at Adventure Works implemented the database as a graph containing **Order** and **Product** nodes connected by **Contains** and **IncludedIn** relationships. The database was prepopulated with **Product** nodes containing the details of each product. When a customer placed an order, an **Order** node was added and a **Contains** relationship was established from order to each product in the order.

*An **Order** node is denormalized structure that contains a subset of the order header information together with some details of each item in the order. Remember, that the full details of each order are recorded in the document database; the graph database only contains the information necessary to generate the product recommendations.*

Similarly, an **IncludedIn** relationship was created from the **Product** node back to the order, as shown in Figure 3. When a customer viewed a product, the Shopping application used an index created over the **Product ID** property to quickly find the **Product** node in the graph database, and then followed the **IncludedIn** relationships to find all orders for that product. The Shopping application then used the **Contains** relationship to find any other products that these orders contain.

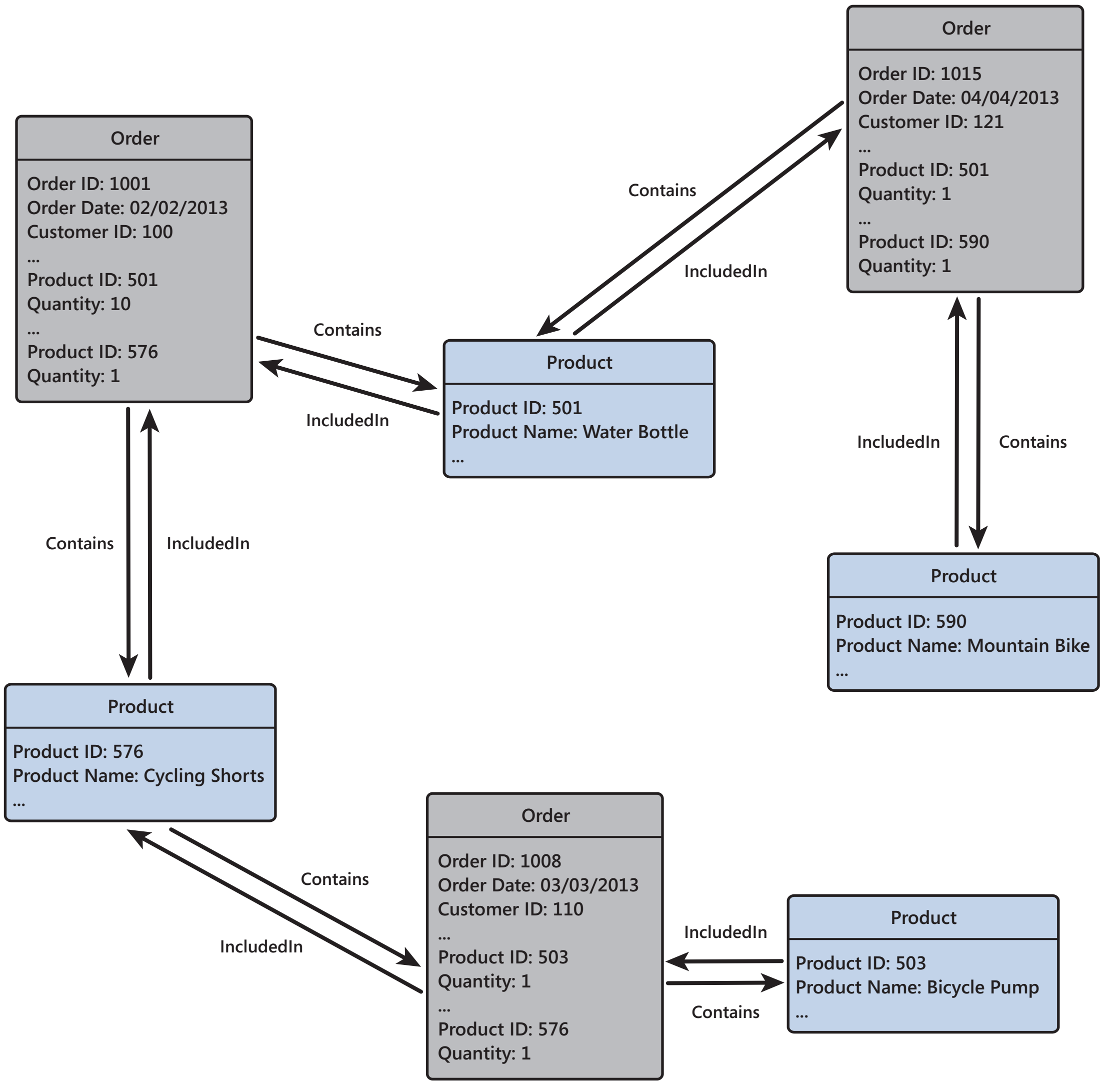


Figure 3
The initial version of the product recommendations database

After evaluating this approach, the developers realized that although it is easy to find products that were purchased together, it was not easy to determine the frequency of these purchases. This made it difficult to ascertain which products the Shopping application should display as its top five recommendations for any given product. To perform this calculation, the Shopping application has to visit each order and count the number of times each product is referenced.

In an attempt to optimize this approach, the developers added a **ReferencedBy** property to each **Product** node. This property is initialized to zero when the database is populated initially, and incremented each time an order is placed for that product. However, this approach still meant that the Shopping application had to visit each product to find the number of times it has been ordered, and the overhead of maintaining the **ReferencedBy** property degraded the performance and caused contention.

Graph databases typically require you to wrap updates in transactions, which lock the data being updated while the transaction is being performed. For more information, see the section "Ensuring Consistency" later in this chapter.

The developers realized that they did not actually need to store information about orders in the database, and could simply hold the product recommendations as a relatively simple network of **RecommendedProduct** nodes containing the product ID and the name of the product. The nodes are connected by a **ProductRecommendation** relationship. For example, if product B is purchased as part of the same order as product A, the **RecommendedProduct** node for product A has a **ProductRecommendation** relationship with product B. The developers also realized that they could store information in the **ProductRecommendation** relationship about the frequency with which product B is purchased as part of an order with product A compared to other products. This frequency is calculated as a percentage. For example, 20% (0.2) of the orders for product B might also occur in orders for product A, whereas only 10% (0.1) of the orders for another product (product C) might occur in orders for product A. The result is a network similar to that shown in Figure 4. Storing this information in the **ProductRecommendation** relationships enables the Shopping application to quickly determine the top five related products for any given product.

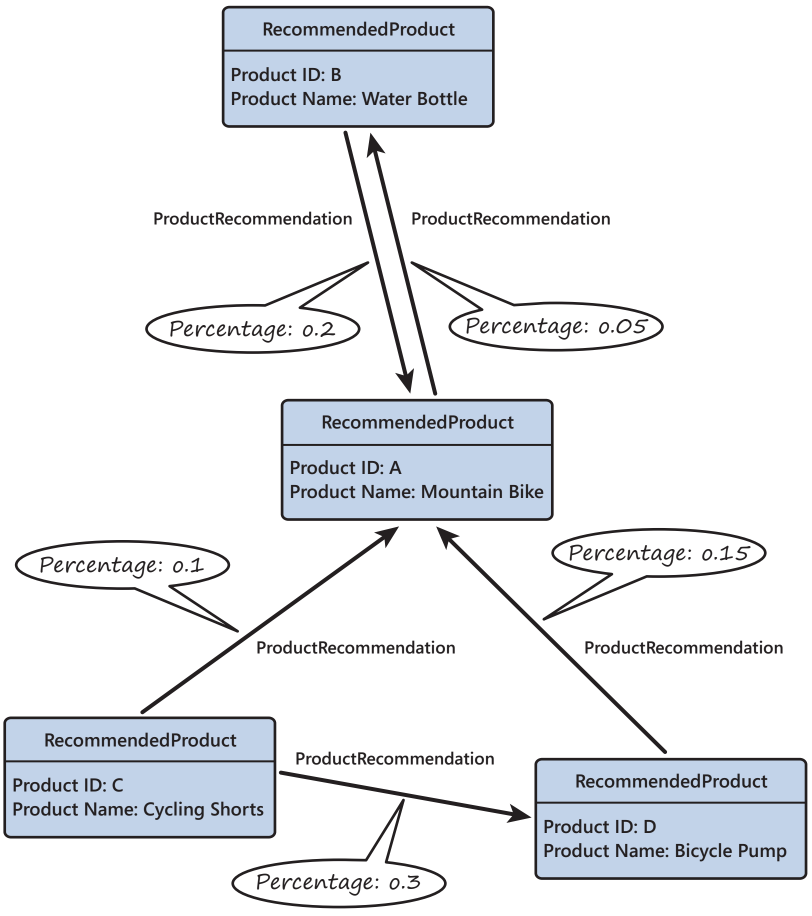


FIGURE 4
A network of recommended products in the graph database

*The **ProductRecommendation** relationship is unidirectional because of the **Percentage** property. 20% of the orders for product B might occur with product A, but the converse is not necessarily true and only 5% of the orders for product A might occur with product B.*

When a customer browses a product, the Shopping application uses the product ID to find the node for this product in the graph database and then it retrieves the list of recommended products by traversing the **ProductRecommendation** relationships from this node. To ensure that the application can locate the node for any given product quickly, the developers at Adventure Works created an index called **product_id_index** over the **ProductId** property in the **RecommendedProduct** nodes in the database.

Due to the volume of orders being continually placed by customers, the Shopping application does not attempt to update the product recommendations database in real time. For more information, see the section "How Adventure Works Implemented Scalability and Availability for the Product Recommendations Database" later in this chapter.

Retrieving Product Recommendations from the Database

As described in Appendix A, "How the MvcWebApi Web Service Works," the developers followed the Repository pattern to connect to the graph databases and isolate the database-specific code from the rest of the application. The **ProductsController** class in the web service creates a **ProductRecommendationRepository** object, and the **GetRecommendations** method in the **ProductsController** class uses this object to retrieve product recommendations. The **ProductRecommendationRepository** class implements the **IProductRecommendationRepository** interface shown in the following code example:

```
public interface IproductRecommendationRepository
{
    ICollection<RecommendedProduct> GetProductRecommendations(int productId);
}
```

The **GetProductRecommendations** method takes the ID of a product and returns a list of **RecommendedProduct** domain entity objects associated with this product. The **RecommendedProduct** class contains the product ID and product name from the corresponding **RecommendedProduct** node in the graph database, but it also contains the value of the **Percentage** property from the **ProductRecommendation** relationship that links the product specified in the **productId** parameter with the product identified by the **RecommendedProduct** object. The **RecommendedProduct** class looks like this:

```
public class RecommendedProduct
{
    public int ProductId { get; set; }
    public string Name { get; set; }
    public decimal Percentage { get; set; }
}
```

The **ProductRecommendationRepository** class uses a series of APIs provided by Neo4j to query the database. Specifically, the code uses a programmatic interface to Cypher provided by the Neo4jClient library for the .NET Framework. The URL of the database to use is specified by using the constructor.

You can find more information about the Neo4jClient library for the .NET Framework online at the Neo4jClient wiki.

The **GetProductRecommendations** method utilizes the p**roduct_id_index** in the database to find the node that matches the value in the **productId** parameter, and then traverses the **ProductRecommendation** relationship from this node to find all product recommendations for this product. Each product recommendation is returned from Neo4j as a **ProductGraphEntity** object. This class is specific to the implementation in the Neo4j database (Neo4j saves the data as JSON formatted objects). The **GetProductRecommendations** method reformats the data and converts it into a collection of **RecommendedProduct** domain objects. The following code sample shows the details of the **GetProductRecommendations** method:

*In the database, the physical name of the logical **ProductRecommendation** relationship is **PRODUCT_RECOMMENDATION**.*

```
public class ProductRecommendationRepository : IProductRecommendationRepository
{
    ...
    private static GraphClient client;
    ...

    public ICollection<RecommendedProduct> GetProductRecommendations(
        int productId)
    {
        ...
        var recommendedProducts = new List<RecommendedProduct>();

        var graphResults = ProductRecommendationRepository.client
            .Cypher
            .Start(new { product =
                Node.ByIndexLookup("product_id_index", "productId", productId) })
            .Match("product-[r:PRODUCT_RECOMMENDATION]->recommended")
            .Return<ProductGraphEntity>("recommended").Results;

        foreach (var graphProduct in graphResults)
        {
            recommendedProducts.Add(new RecommendedProduct()
            {
                Name = graphProduct.Name,
                Percentage = graphProduct.Percentage,
                ProductId = graphProduct.ProductId
            });
        }

        return recommendedProducts;
        ...
    }
}
```

Note that the graph database contains only the top five recommendations for each product. This approach simplifies the query that the **GetProductRecommendations** method has to perform. The section, "How Adventure Works Implemented Scalability and Availability for the Product Recommendations Database" later in this chapter describes how the graph database is populated.

Implementing a Graph Database to Maximize Scalability, Availability, and Consistency

In common with other forms of NoSQL database, an important concern when using a graph database is utilizing the features that it provides to ensure good performance. Graph databases are frequently employed in large-scale environments such as web search engines and social networking applications, and they have to handle queries that can span vast amounts of data quickly and efficiently.

Maximizing Scalability and Availability

The usual approach to maximize scalability in a NoSQL database is to implement sharding. With a graph database, sharding is less useful. In theory, any node in the database can be related to any other node, and splitting the data across servers may slow down the performance of queries rather than speeding them up. Furthermore, many graph databases do not permit nodes connected by a relationship to be distributed across different servers. If you have domain-specific knowledge about nodes, the relationships between them, and the way in which applications query these relationships, you might be able to partition the nodes into isolated groups and store each group on a separate server. For example, in the organization graph described earlier in this chapter, if the organization is a multi-national corporation, you could store the employee and department nodes for departments for each territory in separate databases hosted by servers located in those territories. Strictly speaking, this is not sharding because each server holds a stand-alone database, and the application that queries and maintains the information in the database has to be able to direct operations to the most appropriate database. Figure 5 illustrates this architecture.

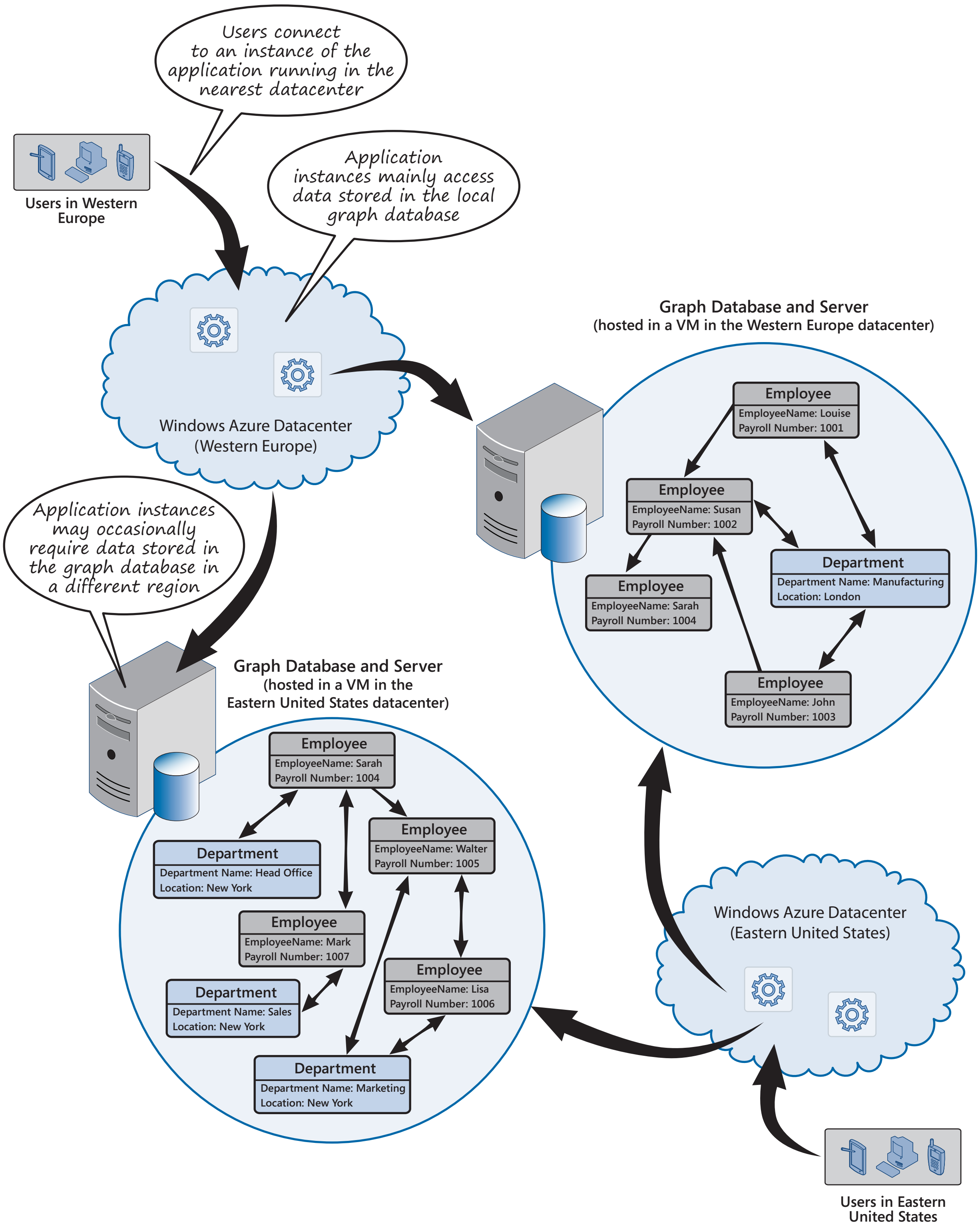

Users connect to an instance of the application running in the nearest datacenter
Users in Western Europe
Application instances mainly access data stored in the local graph database
Windows Azure Datacenter (Western Europe)
Graph Database and Server (hosted in a VM in the Western Europe datacenter)
Employee
EmployeeName: Louise
Payroll Number: 1001
Employee
EmployeeName: Susan
Payroll Number: 1002
Employee
EmployeeName: Sarah
Payroll Number: 1004
Department
Department Name: Manufacturing
Location: London
Employee
EmployeeName: John
Payroll Number: 1003
Application instances may occasionally require data stored in the graph database in a different region
Graph Database and Server (hosted in a VM in the Eastern United States datacenter)
Employee
EmployeeName: Sarah
Payroll Number: 1004
Department
Department Name: Head Office
Location: New York
Employee
EmployeeName: Walter
Payroll Number: 1005
Employee
EmployeeName: Mark
Payroll Number: 1007
Department
Department Name: Sales
Location: New York
Employee
EmployeeName: Lisa
Payroll Number: 1006
Department
Department Name: Marketing
Location: New York
Windows Azure Datacenter (Eastern United States)
Users in Eastern United States

Figure 5
Partitioning organization information across territories

In this example, the Manufacturing department is located in Western Europe (London) while the Marketing and Sales departments and Head Office are based in the eastern United States (Boston and New York). Users view the data through an application that runs in the cloud using Windows Azure. The application has been deployed to Windows Azure datacenters in the Western Europe and Eastern United States regions. The databases and database servers are also deployed to the cloud as Windows Azure Virtual Machines. To reduce network latency, a user connects to an instance of the application that is the same region as the user. Most of the operations performed by the application send requests to the database server in the same datacenter, but if the application requires data that is not held locally it can connect to the database server in the other, remote datacenter.

If a single manager manages employees in different departments across different regions, then the database in each region may need to contain a node for that manager, depending on the queries that the application needs to run. In the example in Figure 5 a node for Sarah has been created in both databases.

You can use replication to improve read scaling. Most commercial graph databases provide support for primary/secondary replication. Write operations are directed to a single primary database, and the replicas provide support for concurrent read operations, although there may be some latency while the data is replicated to the secondary databases. Of course, replication is also a useful technology for ensuring the availability of a graph database. If the computer hosting the primary database fails, one of the secondary servers can be promoted to the primary role, and the original primary server can be enlisted as a secondary server once it has been recovered.

Ensuring Consistency

Most graph databases implement a limited form of internal consistency between nodes and relationships. The reason for this concerns the need to maintain consistency between nodes and relationships. Most graph databases enforce referential integrity checks between nodes and relationships. For example, you cannot create a relationship between nonexistent nodes, and you cannot remove a node that is involved in a relationship. Therefore, unlike other types of NoSQL databases, when you add or modify information in a graph database, you may need to do so in the context of an ACID transaction. If any of the operations in the transaction fail then they are all undone.

In common with many other NoSQL databases, to maximize throughput, a large number of graph databases prefer to cache data in memory and only write to disk when it is absolutely necessary. To prevent data loss, graph databases that follow this approach implement append-only logging, whereby the details of each transaction are added to the end of a transaction log (on disk) when the transaction completes. The transaction is not marked as successful until the transaction log record has been saved. The database file on disk may eventually be updated when records are flushed from memory when the system starts to run short of memory or when the database server is shut down. Appending data to a log file is significantly faster than seeking a location within a database file, making the necessary change, and moving other data around if the change has caused a node to grow. The cost is the increased recovery time in the event of database failure when the database server has to roll the database forward and apply all changes recorded in the log file that had not been written to the database.

To ensure consistency during transactions, many graph databases lock the data being modified, inserted, or deleted. Therefore, it is possible for concurrent transactions to cause deadlock. You should minimize the chances of this happening by designing your applications carefully; always lock resources in a prescribed sequence, and keep transactions as short as possible. Short transactions can also help to alleviate lengthy response times caused by blocking. If two applications attempt to modify the same data, the second application will be blocked by the first, and will only be able to continue once the first application has completed its transaction and released its locks.

If your graph database is subject to a high volume of insert and delete operations, you may be able to improve scalability by configuring a primary/secondary replication topology with writable secondary servers. In this case, write operations can be directed towards the primary server or any writable secondary server, but the transaction surrounding this operation will usually be marked as complete only when the secondary server has successfully synchronized the data with the primary server. Be warned that using writable secondary servers can lead to conflict if two concurrent transactions attempt to modify the same data; the transaction that finishes last may fail if the primary server detects that the data the transaction wishes to modify has already been changed by another transaction using a different secondary server.

How Adventure Works Implemented Scalability and Availability for the Product Recommendations Database

In the Shopping application, product recommendations are displayed when the user browses to a product. This function is performed very frequently, so it is important that product recommendations are retrieved quickly to avoid degrading the customer's experience. The previous chapters have all described how the Shopping application is deployed across multiple Windows Azure datacenters. The system implements read scaling and high availability by using Neo4j Enterprise to replicate the product recommendations database across a collection of virtual machines hosted at each datacenter. Neo4j implements the primary/secondary replication model, and an instance of the Shopping application running in a given datacenter will be directed towards one of the replicas in the same datacenter when it needs to retrieve and display product recommendations.

The developers at Adventure Works decided not to update product recommendations in real time when customers place orders. Instead, they implemented a separate offline batch process to perform this task. This process runs monthly and is outside the scope of the Shopping application. It trawls the list of orders placed that month, and uses this information to generate a list of new product recommendations based on sales recorded in the SQL Server database used by the warehousing system inside Adventure Works. In addition, the batch process determines the top five product recommendations for any given product and only stores these recommendations. This strategy simplifies the queries that the Shopping application performs as it does not have to do any additional filtering other than selecting the recommendations for a product.

The new recommendations replace the existing recommendations in the graph database managed by the primary server in the replication cluster, so the data reflects only the most recent trends in product purchases (as a future extension, the developers might archive the existing recommendations in a separate database before overwriting them, enabling Adventure Works to perform an analysis of historical trends in which products customers purchase). The results are subsequently propagated to the secondary servers. The secondary servers can continue to support read requests (possibly with old data) while the product recommendations are being updated, so there is no down time required.

Summary

This chapter has described how to use a graph database to store information about entities and their relationships.

A graph database is an ideal repository for applications that need to retrieve information about collections of related objects very quickly. You design a graph database to support the specific queries that your applications perform. If you need to perform more generalized, ad hoc queries then an alternative such as a relational database or column-family database may be more appropriate.

You should avoid trying to use a graph database as a store for large, structured entities. If necessary, use a graph database as part of a polyglot solution in conjunction with a document database or key/value store; record information about relationships in the graph database, but save the detailed information in the document database or key/value store.

This chapter also described how Adventure Works used a graph database to hold product recommendations for the Shopping application. It discussed the information that is stored for each product recommendation, and how the product recommendations entities are connected to enable the Shopping application to fetch recommendations for any valid product quickly and easily.

More Information

All links in this book are accessible from the book's online bibliography available at:
http://msdn.microsoft.com/en-us/library/dn320459.aspx.

- You can find information about Neo4j on the website at *http://www.neo4j.org. Full documentation is available online at http://docs.neo4j.org*.
- You can find detailed information about the Neo4jClient library for the .NET Framework at *http://hg.readify.net/neo4jclient/wiki/Home*.

8 Building a Polyglot Solution

Throughout this book, we have emphasized that the purpose of a NoSQL database is to enable an application to store and retrieve data in a scalable, available, and efficient manner. Most NoSQL databases are geared towards specific forms of data access, and you design the structures that you store in a NoSQL database to optimize the queries that your applications perform. Solutions that are based on a relational database tend to store all of their data in a single repository and depend on the relational database management system to perform the optimization for them. However, the advent of NoSQL databases has meant that many application developers, designers, and architects are now looking to apply the most appropriate means of data storage to each specific aspect of their systems, and this may involve implementing multiple types of database and integrating them into a single solution. The result is a polyglot solution.

Designing and implementing a polyglot system is not a straightforward task and there are a number of questions that you will face, including:

- How can you implement a uniform data access strategy that is independent of the different databases? The business logic of an application should not be dependent on the physical structure of the data that it processes as this can introduce dependencies between the business logic and the databases. This issue is especially important of your data is spread across different types of database, where a single business method might retrieve data from a variety of data stores.
- How can you make the best use of different databases to match the business requirements of your applications? This is the driving force behind building polyglot solutions. For example, if your applications need to store and retrieve structured data very quickly you might consider using a document database, but if you need to perform more complex analyses on the same data then a column-family database might be more appropriate. On the other hand, if you need to track and manage complex relationships between business objects, then a graph database could provide the features that you require. In some situations, you might need to support all of these requirements in tandem, in which case you may need to partition the data across different types of database.
- How can you maintain consistency across different databases? For example, an e-commerce application might store customer information in one database and information about orders in another. It is important for the information in these two databases to be consistent; an application should not be able to create and store orders for nonexistent customers.

The developers at Adventure Works implemented the Repository pattern to address the first bullet point in the preceding list. The MvcWebApi web service defines repository classes for each of the different data stores that the controllers use, and these repository classes abstract the details of the underlying database from the controllers. This strategy has been discussed extensively throughout this guide.

The purpose of this chapter is to address the questions raised by the second and third points above, and describe possible solutions that you can implement in your own applications.

Partitioning Data Across Different Types of Database

Up until relatively recently, a traditional in-house data-processing solution would utilize a single database technology, most often a relational database, to provide the repository for the information that it stored and processed. As described in Chapter 1, "*Data Storage for Modern High-Performance Applications*," and Chapter 3, "*Implementing a Relational Database*," relational systems have undoubted strengths and flexibility, but they also have their weaknesses and limitations. These weaknesses have been the driving force behind the rise of NoSQL databases.

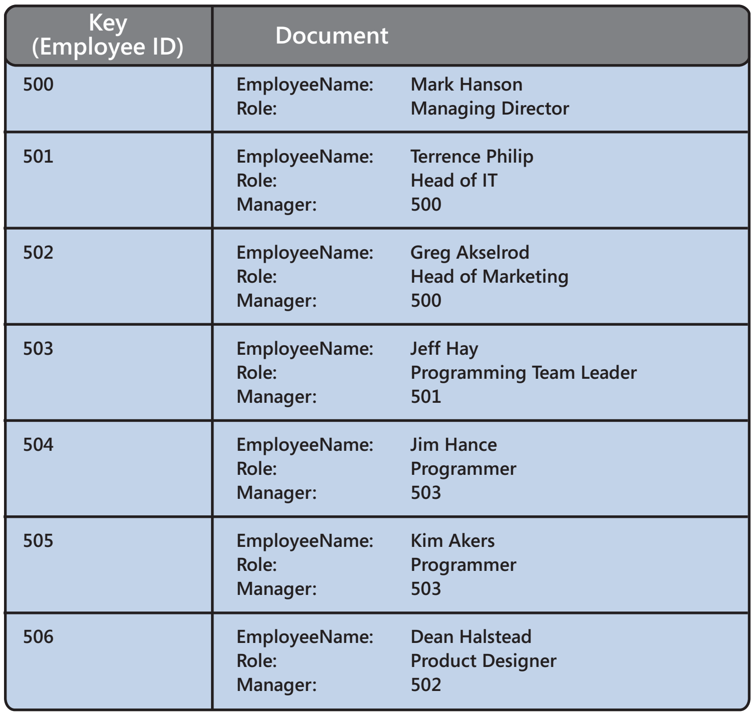
Throughout this guide, it has been pointed out that different NoSQL databases are best suited to different scenarios and use cases, and that the requirements of a typical business application often transcend these use cases. The increasing abundance of highly scalable, highly available, and relatively cheap NoSQL database systems makes building solutions that combine multiple database technologies together a viable and cost effective option. The challenge is in deciding which functionality is most appropriate to a specific type of NoSQL database, and then integrating the resulting conglomeration of technologies into a functional, maintainable application.

There are no hard and fast rules for mapping functionality to specific types of NoSQL databases, but the following list provides some general guidelines that you can consider as a starting point:

- If your application simply stores and retrieves opaque data items and blobs that it identifies by using a key, then use a key/value store.
- If your application is more selective and needs to filter records based on non-key fields, or update individual fields in records, then use a document database.
- If your application needs to store records with hundreds or thousands of fields, but only retrieves a subset of these fields in most of the queries that it performs, then use a column-family database.
- If your application needs to store and process information about complex relationships between entities, then use a graph database.

In many cases, you will encounter situations that cut across this rather simplified set of guidelines. If you need to store and modify highly-structured information, but also need to maintain information about dynamic and possibly complex relationships between records, then should you use a document database or a graph database? Document databases typically don't handle dynamic, complex relationships well, and graph databases tend not to support the rich data-structuring capabilities of a document database. One answer is to use both and take advantage of the synergy that such a solution can provide if it is designed carefully.

Chapter 5, "*Implementing a Document Database*" and Chapter 7, "*Implementing a Graph Database*" presented different aspects of similar examples that could benefit from this approach. In the example in Chapter 5, an application stores information about employees and their managers. To recap, one possible way of structuring this information as a collection of documents is shown in Figure 1.



Key (Employee ID)	Document	
500	EmployeeName:	Mark Hanson
	Role:	Managing Director
501	EmployeeName:	Terrence Philip
	Role:	Head of IT
	Manager:	500
502	EmployeeName:	Greg Akselrod
	Role:	Head of Marketing
	Manager:	500
503	EmployeeName:	Jeff Hay
	Role:	Programming Team Leader
	Manager:	501
504	EmployeeName:	Jim Hance
	Role:	Programmer
	Manager:	503
505	EmployeeName:	Kim Akers
	Role:	Programmer
	Manager:	503
506	EmployeeName:	Dean Halstead
	Role:	Product Designer
	Manager:	502

Figure 1
Storing employee details as a collection of documents

These documents are optimized for simple queries pertaining to the details of a single employee. However, if an application needs to generate an organization chart that lists all the employees that report directly or indirectly to a manager, the application may need to read every document in the entire collection, and perform a significant amount of nontrivial processing to generate the hierarchical structure of the chart. One way of simplifying this processing is to store the information about the managers that an employee reports directly and indirectly to as materialized paths (also described in Chapter 5), but if managers regularly change (as they do in some organizations), then maintaining these materialized paths for a large number of employees could be an expensive and time-consuming process.

The obvious answer to this problem is to store the data in a graph database, as described in Chapter 7. Figure 2 shows the data from Figure 1, but held as a graph of employee nodes.

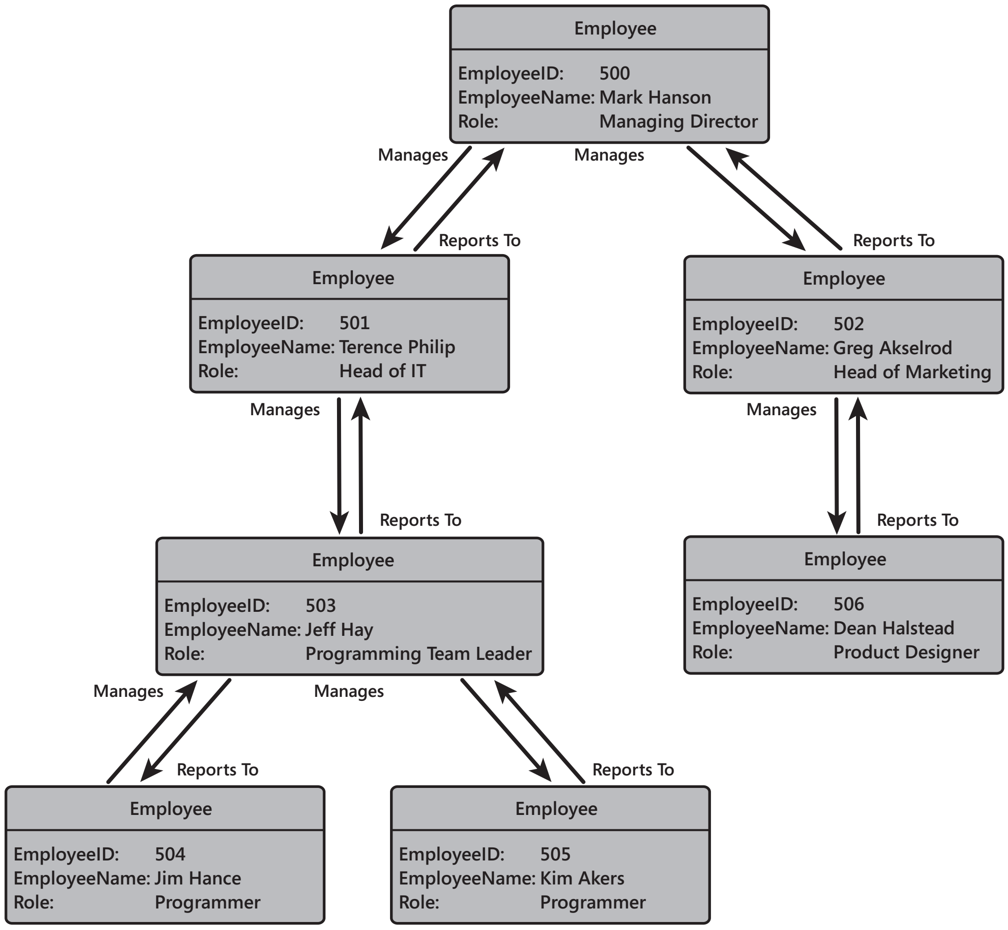
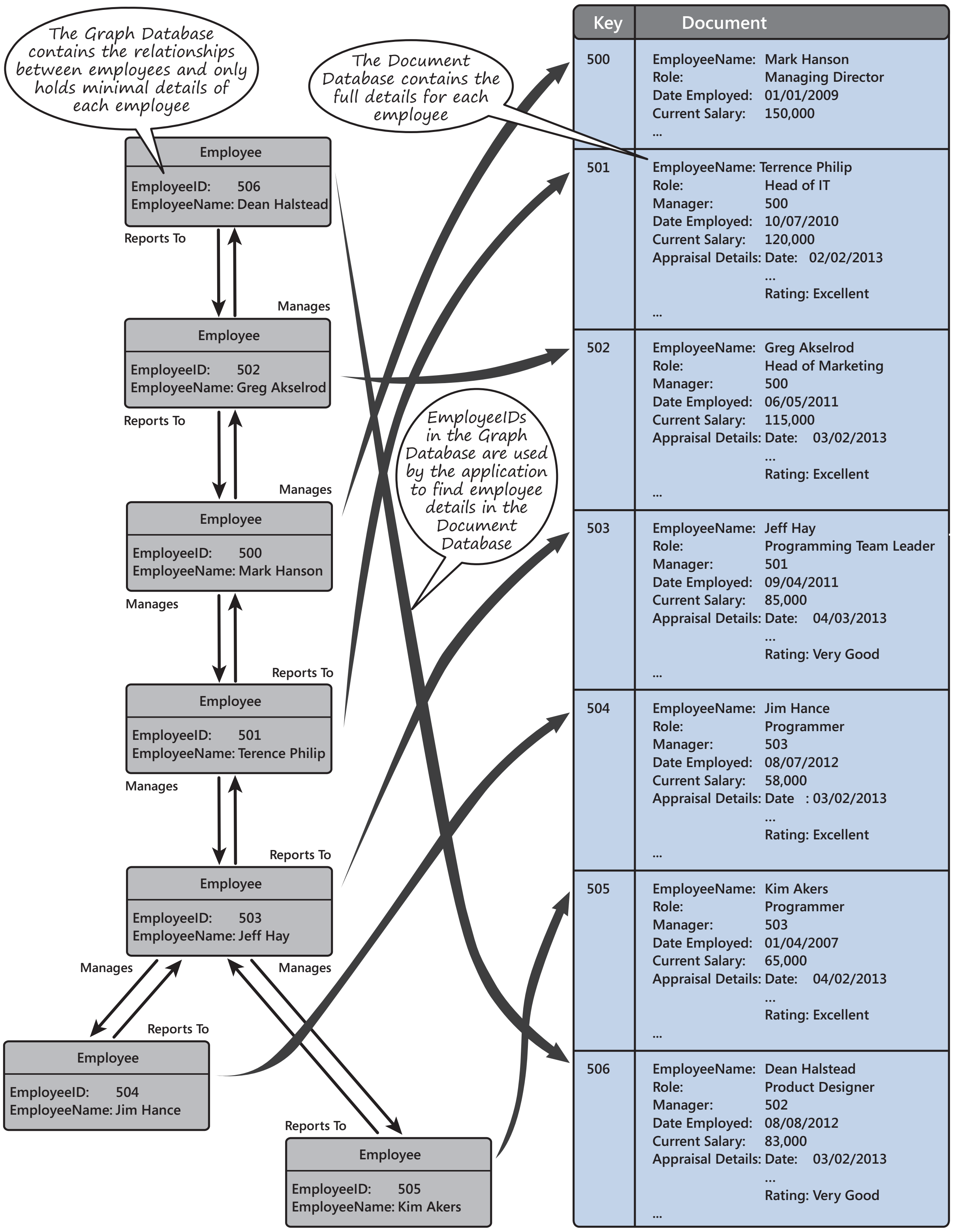


FIGURE 2
Storing employee details as a graph

A graph database such as this is ideal for performing queries that walk down hierarchical relationships. However, if the application needs to store highly structured and complex information about each employee (such as the details of their most recent review), then a graph database might not provide the capabilities necessary to define these structures.

The solution is to store the detailed information about each employee as a collection of documents in a document database, and maintain the information about the managers that they report to as a graph in a graph database. In this way, the data for each employee can be as complicated as the document database will allow, and the graph database only needs to hold the essential details of each employee needed to perform the various graph-oriented queries required by the application. Figure 3 shows this hybrid combination.


The Graph Database contains the relationships between employees and only holds minimal details of each employee
The Document Database contains the full details for each employee
EmployeeIDs in the Graph Database are used by the application to find employee details in the Document Database
Employee
EmployeeID: 506
EmployeeName: Dean Halstead
Reports To
Manages
Employee
EmployeeID: 502
EmployeeName: Greg Akselrod
Reports To
Manages
Employee
EmployeeID: 500
EmployeeName: Mark Hanson
Manages
Reports To
Employee
EmployeeID: 501
EmployeeName: Terence Philip
Manages
Reports To
Employee
EmployeeID: 503
EmployeeName: Jeff Hay
Manages
Manages
Reports To
Employee
EmployeeID: 504
EmployeeName: Jim Hance
Reports To
Employee
EmployeeID: 505
EmployeeName: Kim Akers

Key	Document
500	EmployeeName: Mark Hanson Role: Managing Director Date Employed: 01/01/2009 Current Salary: 150,000 ...
501	EmployeeName: Terrence Philip Role: Head of IT Manager: 500 Date Employed: 10/07/2010 Current Salary: 120,000 Appraisal Details: Date: 02/02/2013 ... Rating: Excellent ...
502	EmployeeName: Greg Akselrod Role: Head of Marketing Manager: 500 Date Employed: 06/05/2011 Current Salary: 115,000 Appraisal Details: Date: 03/02/2013 ... Rating: Excellent ...
503	EmployeeName: Jeff Hay Role: Programming Team Leader Manager: 501 Date Employed: 09/04/2011 Current Salary: 85,000 Appraisal Details: Date: 04/03/2013 ... Rating: Very Good ...
504	EmployeeName: Jim Hance Role: Programmer Manager: 503 Date Employed: 08/07/2012 Current Salary: 58,000 Appraisal Details: Date : 03/02/2013 ... Rating: Excellent ...
505	EmployeeName: Kim Akers Role: Programmer Manager: 503 Date Employed: 01/04/2007 Current Salary: 65,000 Appraisal Details: Date: 04/02/2013 ... Rating: Excellent ...
506	EmployeeName: Dean Halstead Role: Product Designer Manager: 502 Date Employed: 08/08/2012 Current Salary: 83,000 Appraisal Details: Date: 03/02/2013 ... Rating: Very Good ...

Figure 3
Storing employee details in a document database, and organizational details in a graph database

The employee documents and the employee nodes share a common set of employee IDs. The application can use the employee ID from a node in the graph database to find the details of that employee in the document database. Additionally, the graph database includes copies of the properties required to generate the organization chart. These copied properties help to optimize the application code that creates the organization chart as it does not need to retrieve this information from the document database, and it consists of information that does not change frequently (more dynamic data, such as salary and role information, is best left in the document database to avoid the overhead of maintaining it in two places).

This strategy is extremely flexible, enabling you to mix and match the databases necessary to the data storage and processing requirements of the application. For example, if you needed to store a complete salary and appraisal history for each member of staff, you could incorporate a column-family database. However, the cost of this approach is the additional logic that the application must implement when it needs to add, modify, or delete information because it must ensure that the data is consistent across all databases. This issue is discussed in the next section.

Managing Cross-Database Consistency

The primary reason for splitting information across different types of databases is to optimize the way in which applications store and retrieve data. However, each database is an independent repository and in most cases a database has no knowledge of any other databases used by the system. Maintaining consistency across databases is therefore an application issue, and it is the responsibility of the data access logic in the application to ensure that inserts, updates, and deletes that affect the data in multiple databases are actually propagated to those databases.

In the relational world, an application typically handles cross-database consistency by implementing distributed transactions that implement the Two-Phase Commit (2PC) protocol. However, as described in the section "*Performance and Scalability*" in Chapter 1, "Data Storage for Modern High-Performance Applications," the 2PC protocol can cause bottlenecks in an update-intensive system due to data being locked while the databases that participate in the transaction coordinate with each other, and is not ideally suited to running on slow or unreliable wide-area networks such as the Internet.

If your application performs OLTP operations, then a relational database is likely to be a better solution than a NoSQL database. However, you should be prepared to ask whether your application actually needs to implement operations as transactions, or whether you can trade immediate consistency for performance.

One of the main tradeoffs that NoSQL databases make is that of consistency against performance; they prefer to be quick rather than immediately accurate, as long as the system becomes consistent at some point. As a result, although most NoSQL databases support the notion of atomic writes to a single aggregate (a value in a key/value store, a document in a document database, or a collection of columns in a column family), most do not implement transactions that span multiple records (graph databases are an exception, due to the requirement that relationships must not reference nonexistent nodes, as described in Chapter 7, "*Implementing a Graph Database*"). In many cases, the immediate consistency of ACID transactions is not necessary and it may be perfectly acceptable for there to be a degree of lag while different databases are updated. In these situations, a solution can opt to implement eventual consistency, but the important point is that any changes must ultimately be reflected across every database, and the details of any change should never be lost.

You can implement eventual consistency in a variety of ways, and the following sections describe some possible strategies.

A few NoSQL databases provide limited transactional capabilities. Additionally, some NoSQL databases support integration with third-party libraries that enable an application to lock records during a series of operations to ensure that these records are not changed by another concurrent application. However, you should beware of possible performance problems and blocking that could occur if an application fails to release locks in a timely manner (or forgets to release them altogether). It is usually safer to design your solutions around the eventual consistency model and eliminate any need for locking.

Handling Consistency in the Business Logic of a Solution

The section "*Integrating NoSQL Databases into a Polyglot Solution*" in Chapter 1, "Data Storage for Modern High-Performance Applications" described an approach for hiding the data access logic for different NoSQL databases behind a web service façade. The façade provides an operational interface that exposes the business methods required by an application, and the implementation ensures that the requests that the business logic an application performs are directed towards the appropriate database(s). This is the approach that the designers at Adventure Works took when they built the MvcWebApi web service for the Shopping application.

In this architecture, the web service implementing the façade acts as the point of integration for the various NoSQL databases. The logic in the methods that the web services exposes handle the various database interactions, storing and retrieving data from one or more databases. Furthermore, these web methods take on the responsibility for detecting any failures when saving or fetching data, handling these failures, and if necessary implementing any compensation logic that undoes the work performed by an operation that cannot be completed successfully. How a method handles failures depends ultimately on the business scenario and the tolerance that an application has to these failures. In some cases, an error when saving data can be ignored if the data is transient or short lived while in other cases the failure may be more critical and require correction. For example, in an application that monitors the second by second changes in stock prices, the failure to save a single change to a price may not be critical as the price may change again very quickly (this scenario assumes that the data is being recorded for analytical purposes rather than as part of an online share trading system). On the other hand, in a banking system that transfers money between accounts, it is vital that every change is safely and accurately recorded.

Another key issue is idempotency. If possible, you should seek to eliminate any dependencies on the order in which the different repositories in a polyglot solution are updated. This strategy can help to increase throughput and alleviate some of the complex failure logic that you might otherwise have to implement. For example, if a business operation modifies information in several databases, it may be possible to perform the updates in parallel by using multiple threads. In a widely distributed environment performing a large number of concurrent operations, it is possible that an individual update may fail due to transient networking issues or timeouts. If the various updates performed by the business operation are idempotent, then a failing update against a specific repository can simply be retried without affecting any of the other updates. Only if an update fails irretrievably might it be necessary to undo the work performed by the other updates that comprise the operation.

How the Shopping Application Creates an Order

In the MvcWebApi web service used by the Shopping application, the data for an order spans three different databases; a key/value store, a SQL Server database, and a document database. The following steps describe the operations that affect an order:

- When a customer browses the product catalog, the customer can add items to their shopping cart. This shopping cart is persisted in a key/value store and is preserved between sessions. The customer can log out, and when the customer returns the shopping cart is restored.
- When the customer places an order, the items in the shopping cart are used to construct an order. The details of the order are passed to the orders service which records the order in the order history database with the status set to Pending and then stores the order information in the database used by the warehousing and dispatch systems inside Adventure Works (these systems are responsible for fulfilling the order and dispatching it to the customer). When the order is processed, the system writes another record to the order history database with the status set to Complete to indicate that that the order is complete.
- As the order is processed, any changes to the order are also recorded in the order history database.

The **Post** method in the **OrdersController** is responsible for creating an order from the information in a customer's shopping cart. It communicates with the order service, which saves a record of the order in the order history database and then passes the details of the order to the warehousing and dispatching systems inside Adventure Works. The order service implements the **IOrderService** interface. This interface exposes a method called **ProcessOrder** that takes an **Order** object as its parameter. The following code shows the relevant parts of the **Post** method in the **OrdersController** that invokes the order service:

```
public class OrdersController : ApiController
{
    private readonly IOrderHistoryRepository orderHistoryRepository;
    private readonly IShoppingCartRepository shoppingCartItemRepository;
    private readonly IPersonRepository personRepository;
    private readonly IProductRepository productRepository;
    private readonly IOrderService orderService;

    ...

    public HttpResponseMessage Post(OrderInfo order)
    {
        ...

        // Find the shopping cart for the user
        var shoppingCart = this.shoppingCartItemRepository.GetShoppingCart(
            order.CartId.ToString());
        ...
```

```
            // Create and populate a new Order object (called newOrder)
            // with the list of products in the shopping cart (most details omitted)

            var newOrder = new Order()
            {
                ...
                Status = OrderStatus.Pending;
            };

            ...

            try
            {
                this.orderService.ProcessOrder(newOrder);
            }
            catch (Exception ex)
            {
                // this save failed so we cannot continue processing the order
                return Request.CreateErrorResponse(
                    HttpStatusCode.InternalServerError, ex.Message);
            }

            // since we have captured the order
            // we can delete the users shopping cart now
            this.shoppingCartItemRepository.DeleteShoppingCart(
                shoppingCart.UserCartId);
            ...
        }
        ...
    }
```

The first part of the **Post** method retrieves the shopping cart from the **ShoppingCartRepository**, and uses this information to create a new **Order** object (most of this code is omitted in the example above). The status of this order is initially marked as **Pending**. The code in the **try … catch** block invokes the **ProcessOrder** method of the order service to save the order, and if this step is successful the shopping cart is deleted. Removing the shopping cart helps prevent the same order from being placed again accidentally.

If the shopping cart is not deleted correctly, the logic in the various controllers that respond to requests from the user interface is responsible for detecting this failure and handling it appropriately. The section "How the Shopping Application Prevents Duplicate Orders" describes this scenario in more detail.

The sample application implements the order service in the **OrderService** class. You can find this class in the **DataAccess.Domain.Services** project in the **Domain** folder of the solution. The **ProcessOrder** method in this class saves the order details (with a status of **Pending**) to the order history repository and then submits the order for processing. Handling the order may take some time, so the **ProcessOrder** method sends it to a queue. A separate service can take the order and process it in its own way. The following code example shows the relevant parts of the **ProcessOrder** method in the **OrderService** class:

```
public sealed class OrderService : ...
{
    private readonly IOrderHistoryRepository orderHistoryRepository;
    private readonly IOrderQueue orderQueue;
    ...

    public void ProcessOrder(Order newOrder)
    {
        ...
        if (newOrder.Status != OrderStatus.Pending)
        {
            // No need to do anything if the order is not Pending.
            return;
        }

        // Save the order to history as Pending to make sure
        // we capture the order
        this.orderHistoryRepository.SaveOrderHistory(
            new OrderHistory(newOrder));

        // At this point you have everything you need from the requestor.
        // What you can do next is to send a message with the trackingId
        // to a queue.
        this.orderQueue.Send(newOrder.TrackingId);
    }
    ...
}
```

Once the order has been sent to the queue, the **OrderService** class considers that the order has been placed and will be fulfilled. It is the job of the service that retrieves the order from the queue and that handles the order to ensure that this is actually the case.

How the OrderProcessor Service Handles an Order

The sample solution mimics a reliable queue by using the **SimulatedQueue** class. The **Send** method in this class simulates the process of reading the message that contains the order and passes it to an instance of the **OrderProcessor** class for handling. The **OrderProcessor** could fail for a variety of reasons. The cause of the error might be transient, and in the real world, the message containing the order should not be removed from the queue. Rather, it should remain on the queue so that it can be read and handled again.

*In theory, the **OrderProcessor** could continually fail to handle an order, and rather than repeatedly retrying the operation, after a certain number of attempts the system should raise an alert and allow an operator to intervene and rectifycorrect any problems. The **DeliverMessage** method in the **SimulatedQueue** class shows how you can structure your code to achieve this.*

The **OrderProcessor** class in the sample solution provides the **SaveOrderAndSaveOrderHistory** method that writes the details of the order in the orders database and adds a second record for the order in the order history database with the status set to **Completed**. The method has to save the order information to both databases for the operation to be successful. However, either or both of these actions could fail causing the method to abort and leaving the system in an inconsistent state. This eventuality should cause the processing in the **SimulatedQueue** object to fail and in turn cause the message containing the order to remain in the queue, as described earlier. The **SimulatedQueue** object can read the message again and pass it to the **SaveOrderAndSaveOrderHistory** method for another attempt at saving the order. However, one or possibly both of the writes performed by this method might actually have succeeded in the earlier attempt (the failure could be due to some other factor). The system should not allow the order to be duplicated, so the **SaveOrderAndSaveOrderHistory** method performs its actions as an idempotent process. Before each write, the **SaveOrderAndSaveOrderHistory** method reads the appropriate database to determine whether the order details have already been saved in that database. If they have, then there is no need to write this information again.

Eventually, the system will either become consistent, or an administrator will have been alerted to take the necessary corrective actions.

The following code example shows the code in the **SaveOrderAndSaveOrderHistory** method of the **OrderProcessor** class that saves the order details to the orders database and the order history database:

```
private void SaveOrderAndSaveOrderHistory(Guid trackingId)
{
    // This method writes to two different databases.
    // Both writes need to succeed to achieve consistency.
    // This method is idempotent.
    bool isOrderSavedInOrderRepo =
        this.orderRepository.IsOrderSaved(trackingId);

    bool isOrderSavedInOrderHistoryRepoAsCompleted =
        this.orderHistoryRepository.IsOrderCompleted(trackingId);

    if (!isOrderSavedInOrderRepo || !isOrderSavedInOrderHistoryRepoAsCompleted)
    {
        ...
        if (...)
        {
            if (!isOrderSavedInOrderRepo)
            {
                // Once the following orderRepository.SaveOrder method is
                // completed successfully, the flag
                // isOrderSavedInOrderRepo should return true the next time
                // the SaveOrderAndSaveOrderHistroy
                // method gets called for the same trackingId.
                this.orderRepository.SaveOrder(newOrder);
            }
            if (!isOrderSavedInOrderHistoryRepoAsCompleted)
            {
                newOrder.Status = OrderStatus.Completed;

                // Once the following SaveOrderHistory method is
                // completed successfully, the flag
                // isOrderSavedInOrderHistoryRepoAsCompleted should
                // return true the next time the
                // SaveOrderAndSaveOrderHistroy method gets called for
                // the same trackingId.
                this.orderHistoryRepository.SaveOrderHistory(
                    new OrderHistory(newOrder));
                }
            }
        }
    }
}
```

It is also possible that the order history contains orders that are marked as **Pending** but that never get processed for some reason. You can implement a separate process that periodically sweeps through the order history database to find such orders and add them to the queue for processing by the **OrderProcessor** class.

How the Shopping Application Prevents Duplicate Orders

It is important to understand that if the shopping cart is not deleted when an order is placed the order is still processed. However, it is still necessary to remove the shopping cart at some point to prevent the same order from being repeated. In the Shopping application, failure to delete the shopping cart is considered to be a user-interface issue rather than a business problem, and is handled outside of the Post method. For example, the MvcWebApi web service detects whether a stray shopping cart exists for a customer when the customer logs in and delete it.

```
public class AccountController: ApiController
{
    private IPersonRepository personRepository;
    private IShoppingCartRepository shoppingCartRepository;
    private ISalesOrderRepository orderRepository;;
    ...

    [HttpPost]
    public HttpResponseMessage Login(LoginInfo loginInfo)
    {
        ...
        var person = this.personRepository.GetPersonByEmail(loginInfo.UserName);

        ...

        // Check for an orphaned shopping cart
        var shoppingCart = this.shoppingCartRepository.GetShoppingCart(
                                    person.PersonGuid.ToString());
        if (this.orderRepository.IsOrderSubmitted(shoppingCart.TrackingId))
        {
            // Order is already submitted, delete the shopping cart;
            this.shoppingCartRepository.DeleteShoppingCart(
                                                    shoppingCart.UserCartId);
        }
        ...
    }
    ...
}
```

Each order is identified by a unique tracking ID by the internal systems inside Adventure Works. When a customer first places an item in a new shopping cart, the constructor for the **ShoppingCart** domain entity class generates this ID and stores it in the **TrackingId** field, as shown below:

```
public class ShoppingCart
{
    ...
    public ShoppingCart(string userCartId)
    {
        this.UserCartId = userCartId;
        this.TrackingId = Guid.NewGuid();
    }

    public string UserCartId { get; set; }
    ...
    public Guid TrackingId { get; set; }
    ...
}
```

When a customer logs in and has a shopping cart saved from a previous session, it queries the orders database to find out whether any existing orders have the same value for the tracking ID as that recorded in the shopping cart. If so, the cart had not been deleted correctly when the order was placed, so it removes it (this attempt may fail again, of course).

This check assumes that after a customer has placed an order they are likely to log off rather than place another order (this is the most common pattern of use in many merchant web systems). In situations where this is not the case, the check performed by the **login** method is not sufficient on its own to prevent duplicate orders from being submitted.sIn this case, it is necessary for the order processing logic to detect the duplicate order. In the sample solution, the **SaveOrderAndSaveOrderHistory** method in the **OrderProcessoet** class performs just such a check, and only attempts to save the order if it has not already been processed (it must have the status **Pending** rather than **Completed**) otherwise it is silently discarded.

```
private void SaveOrderAndSaveOrderHistory(Guid trackingId)
{
    ...
    if (!isOrderSavedInOrderRepo || !isOrderSavedInOrderHistoryRepoAsCompleted)
    {
        var newOrder = this.orderHistoryRepository.
            GetPendingOrderByTrackingId(trackingId);

        if (newOrder != null)
        {
           ...
        }
    }
}
```

Synchronizing Databases by Using a Batch Process

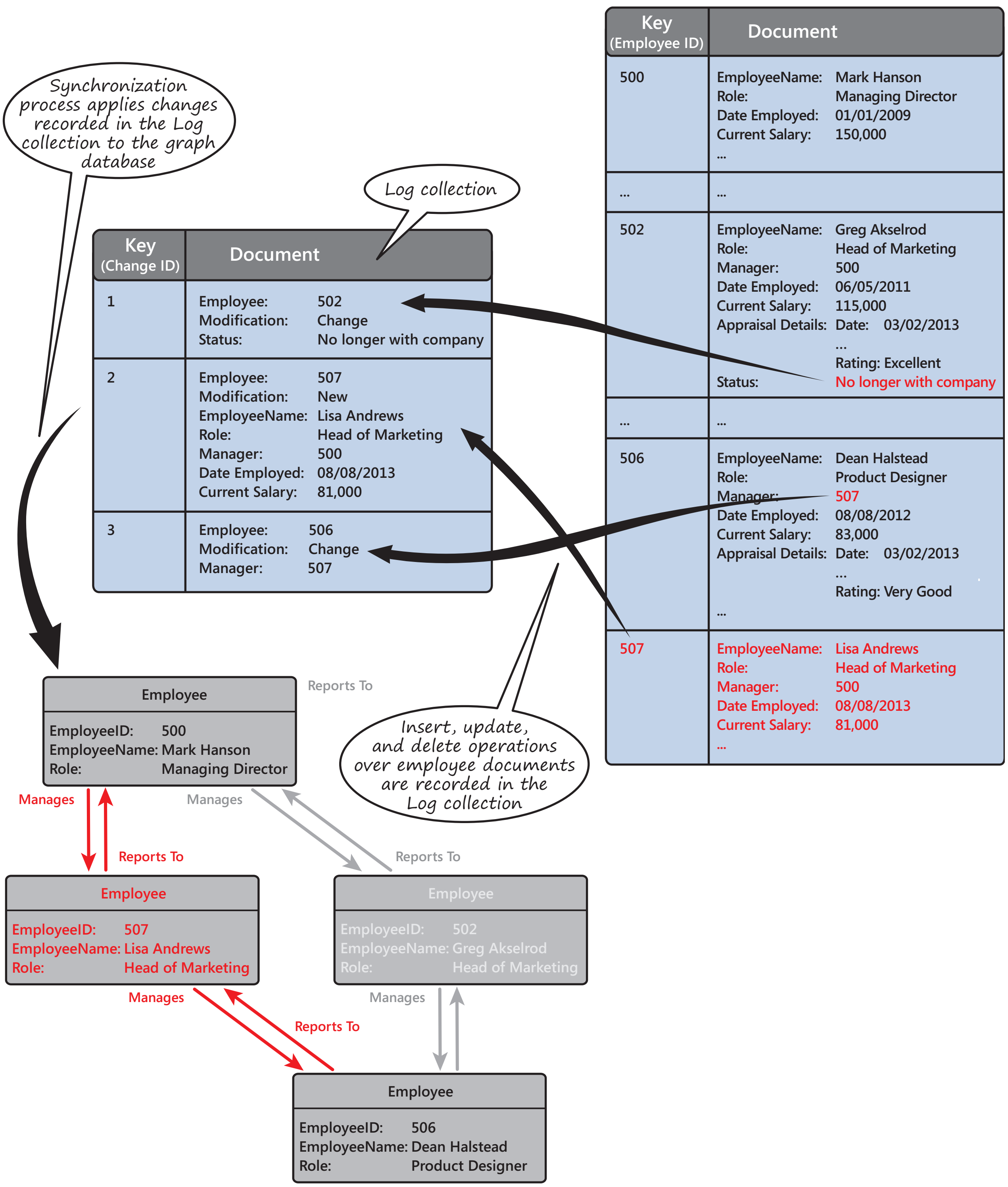
Depending on the business scenario, you can implement eventual consistency between databases by synchronizing them periodically. In the organization example described previously, you can arrange to run a script at regular intervals that detects the changes that have occurred to employee data in the document database since the last synchronization, and use this information to update the graph database.

The organization example depicts one-way synchronization and assumes that the document database is the authoritative source of information. If you need to implement bidirectional or even multi-way synchronization between a set of databases, then this solution might not be appropriate, and a better strategy could be to propagate updates as they occur by using events, as described in the next section.

This strategy requires that you can easily identify the modifications (including creates and deletes) that have been made recently. You may be able to extract this information from the database log file or journal. Many NoSQL databases are open source, and provide some information on the structure of the log files that you can use to develop your own applications that need to read this data.

The database log may be truncated or removed if the server is shutdown cleanly. Many NoSQL databases use the presence of a log file to detect whether the database was not shut down properly previously, and then use the information in the log file to recover any missing data from the database before deleting it. Additionally, many NoSQL databases also perform periodic checkpoints and truncate the database log after ensuring that all the changes it contains have been safely written to the database.

Alternatively, you can implement a collection of *Log* documents in the database and add the details of any changes to employee documents to this collection. The synchronization process can iterate through this collection to determine the changes that it needs to make, and then clear out all documents for records that have been synchronized when the process completes. Figure 4 illustrates this approach.



Key (Employee ID)	Document
500	EmployeeName: Mark Hanson Role: Managing Director Date Employed: 01/01/2009 Current Salary: 150,000 ...
...	...
502	EmployeeName: Greg Akselrod Role: Head of Marketing Manager: 500 Date Employed: 06/05/2011 Current Salary: 115,000 Appraisal Details: Date: 03/02/2013 ... Rating: Excellent Status: No longer with company
...	...
506	EmployeeName: Dean Halstead Role: Product Designer Manager: 507 Date Employed: 08/08/2012 Current Salary: 83,000 Appraisal Details: Date: 03/02/2013 ... Rating: Very Good ...
507	EmployeeName: Lisa Andrews Role: Head of Marketing Manager: 500 Date Employed: 08/08/2013 Current Salary: 81,000 ...

Key (Change ID)	Document
1	Employee: 502 Modification: Change Status: No longer with company
2	Employee: 507 Modification: New EmployeeName: Lisa Andrews Role: Head of Marketing Manager: 500 Date Employed: 08/08/2013 Current Salary: 81,000
3	Employee: 506 Modification: Change Manager: 507

Synchronization process applies changes recorded in the Log collection to the graph database
Log collection
Insert, update, and delete operations over employee documents are recorded in the Log collection
Employee
EmployeeID: 500
EmployeeName: Mark Hanson
Role: Managing Director
Reports To
Manages
Manages
Reports To
Reports To
Employee
EmployeeID: 507
EmployeeName: Lisa Andrews
Role: Head of Marketing
Employee
EmployeeID: 502
EmployeeName: Greg Akselrod
Role: Head of Marketing
Manages
Reports To
Manages
Employee
EmployeeID: 506
EmployeeName: Dean Halstead
Role: Product Designer

Figure 4
Using a collection of Log documents to synchronize databases manually

In this example, employee 502 has left the company and the **Status** field in the employee database is recorded as "No longer with company." The code that makes this change writes a corresponding document to the Log collection. A new employee is added (507), and the **Manager** field for employee 506 is changed to 507. The Log collection also contains a record of these changes. When the graph database is synchronized with the document database, the synchronization process reads the changes from the Log collection and makes the appropriate amendments to the graph before removing the records from the Log collection.

It is important to implement the synchronization process in an idempotent manner. If the synchronization process fails or the Log collection is not flushed correctly, then it should be possible to run the synchronization process again without worrying about which changes it had previously applied.

However, while this strategy is relatively simple, it imposes an overhead on the document database and requires additional logic in the application code that maintains the data in this database (it must record the details of changes in two places—in the Employees collection and in the Log collection). You should also remember that in most document databases, writes to multiple documents (such as an employee and a log document) are not atomic so there is a small possibility that an employee document could be updated but the change not recorded in the Log collection.

This strategy is only suitable if your application can tolerate a significant amount of latency (possibly several hours, or even days) in the consistency between databases. In scenarios such as the organization example where the data is not time critical, this latency would probably be perfectly acceptable. You may be able to automate the synchronization process to allow it to run as a timed job every few minutes, but if you require lower latency then this may not be the most suitable approach.

You can also use a pattern such as Event Sourcing to tackle this problem; make the Log collection the authoritative source of information and synchronize the Employees collection and the graph database with the data in the Log collection. For more information, read the section "*Introducing Event Sourcing* " in the guide "*CQRS Journey* ", available on the MSDN website.

How the Shopping Application Verifies Product Pricing and Availability

In the Shopping application, the data that describes products is held in two places; the SQL Server database used by the warehousing system inside Adventure Works, and the product catalog implemented by using MongoDB. The MongoDB database is replicated to spread the load across a collection of sites and servers, as described in the section "Implementing Scalability and Availability for the Product Catalog" in Chapter 5.

As far as the Shopping application is concerned, the product catalog is a read-only document database. However, outside of the Shopping application, it is updated periodically by a batch process with data from the warehousing database (it is not critical that the data in the product catalog is completely up to date). The approach taken is very straightforward; a script that uses a series of SQL statements to retrieve the most recent details of every product in the SQL Server database runs at monthly intervals. This script then connects to the master node in each MongoDB replication cluster, removes the existing category, subcategory, and product document collections, and then recreates them with the new data before rebuilding the indexes over these documents. The whole process takes no longer than one or two minutes to perform, and while it is ongoing users are still able to query the product catalog through the subordinate nodes in each replication cluster.

The developers adopted the same strategy for the graph database that holds the details of product recommendations (like the product catalog, it is not critical that this information is totally up to date). They implemented a monthly batch process that analyzes the recent orders in the SQL Server database used by the warehousing system, and extract the combinations of products that are most frequently bought together. The batch process then uses this data to update the graph database running at each Windows Azure datacenter.

It is possible that the price or availability of a product may change in the warehousing database, but this change will not be reflected until the next time the database is refreshed. The warehousing database is considered to be the authoritative source for product availability and pricing information. To counter possible inconsistencies between the two databases, when a customer places an order, the **Post** method in the **OrdersController** examines the items in the customer's shopping cart and cross-checks them against the warehousing database by calling the **InventoryAndPriceCheck** method of the **InventoryService** object that simulates the warehousing system inside Adventure Works (the section "*Retrieving Data from the SQL Server Database*" in Chapter 3 provides information on how the **InventoryAndPriceCheck** method works). If any item is discontinued or the price has changed since the product catalog was last updated, the **OrdersController** returns without placing the order and passes a response back to the user interface that enables it to alert the customer of the discrepancy. The following code example shows the relevant parts of the **Post** method:

```
public class OrdersController : ApiController
{
    ...
    private readonly IShoppingCartRepository shoppingCartItemRepository;
    private readonly IPersonRepository personRepository;
    private readonly IProductRepository productRepository;

    ...
    private readonly IInventoryService inventoryService;;
    ...]
    public HttpResponseMessage Pose(OrderInfo order)
    {
        ...

        var shoppingCart = this.shoppingCartItemRepository.GetShoppingCart(
            order.CartId.ToString());
        ...

        if (this.inventoryService.InventoryAndPriceCheck(shoppingCart))
        {
            // There was an inventory or price discrepancy between the
            // shopping cart and the product database
            // so we save the shopping cart and return an OK since
            // there is nothing RESTfully wrong with the message.
            // The UI will be responsible for handline this by
            // redirecting to an appropriate page.
            this.shoppingCartItemRepository.SaveShoppingCart(shoppingCart);

            var cartItems = new List<CartItem>();
            Mapper.Map(shoppingCart.ShoppingCartItems, cartItems);

            return Request.CreateResponse(HttpStatusCode.OK, cartItems);
        }

        ...
    }
    ...
}
```

The section "*Placing an Order*" in Chapter 2, "*The Adventure Works Scenario*" describes how the Shopping application handles this situation. The new details from the warehousing system are displayed and the user can either discard the contents of the shopping cart or proceed and place the order with the updated information.

Synchronizing Databases by Using Events

Event-based synchronization provides an approach that has much lower latency than batch synchronization, but is also architecturally more complex. Each database requires a separate process that maintains the data in the database in response to events.

In this strategy, as an application modifies the data in a database, it also triggers events. The processes that manage each database listen for these events and take the appropriate action. Therefore, this mechanism requires an infrastructure that can propagate these events in a distributed environment, such as a service bus. There are a number of commercial service bus implementations currently available including Windows Azure Service Bus. A variety of open source solutions are also available.

Many service bus implementations, including that provided by Service Bus, are based on reliable message queuing. In the case of Service Bus, an administrator creates a message queue in the cloud, and applications can connect to the message queue to send and receive messages through well-defined endpoints.

Service Bus supports transactional semantics when sending messages to a queue. An application can initiate a local transaction, post a series of messages, and then complete the transaction. If the application detects an error during the transaction it can abort, and the messages that it has posted will be removed. If the application completes the transaction, all the messages will be sent.

Service Bus transactions only effect operations that send or receive messages on a queue, and they cannot enlist other transactional stores such as a relational database. For example, if you start a Service Bus transaction, send several messages to a queue and update a database, and then abort the transaction, the messages will be removed from the queue but the database update will not be rolled back.

On the receiving end, an application can connect to the queue, read messages, and process them. When an application reads a message it is removed from the queue. When the application has finished with the message it can indicate that processing was completed successfully, but if an error occurs it can abandon the message, which causes it to be returned to the queue ready to be read again. Message queues can also have an associated timeout, and if the application that reads a message does not indicate that it has finished with the message before this timeout expires the service bus assumes that the application has failed and returns the message to the queue.

Once a message has been read and processed, it is no longer available to any other applications that may need to read the same message. In a NoSQL polyglot solution, it is possible that the details of a single update might need to be passed to several databases. In this case, you need a publish/subscribe model, where a single message can be sent to all interested parties, and not just the first one that happens to read it. Many service bus implementations provide just such a model. In the case of Windows Azure Service Bus, you can use topics and subscriptions. A topic is very similar to the *sending end* of a message queue since an application can post messages to it (they also provide the same transactional capabilities as Windows Azure Service Bus message queues). Different applications can then subscribe to this topic, and in effect they get their own private message queue from which they can read messages. Messages posted to the topic are copied to each subscription, and an application retrieves messages from the subscription in the same way as it fetches messages from a queue. Subscriptions also provide the *complete/abandon* mechanism with timeouts available with message queues.

By using the topic/subscriber model, an application that needs to update several databases can create a message that describes this update and post it to a topic. Subscribers that handle each of the databases can read the message and perform whatever processing is appropriate at their end. Figure 5 shows this approach:

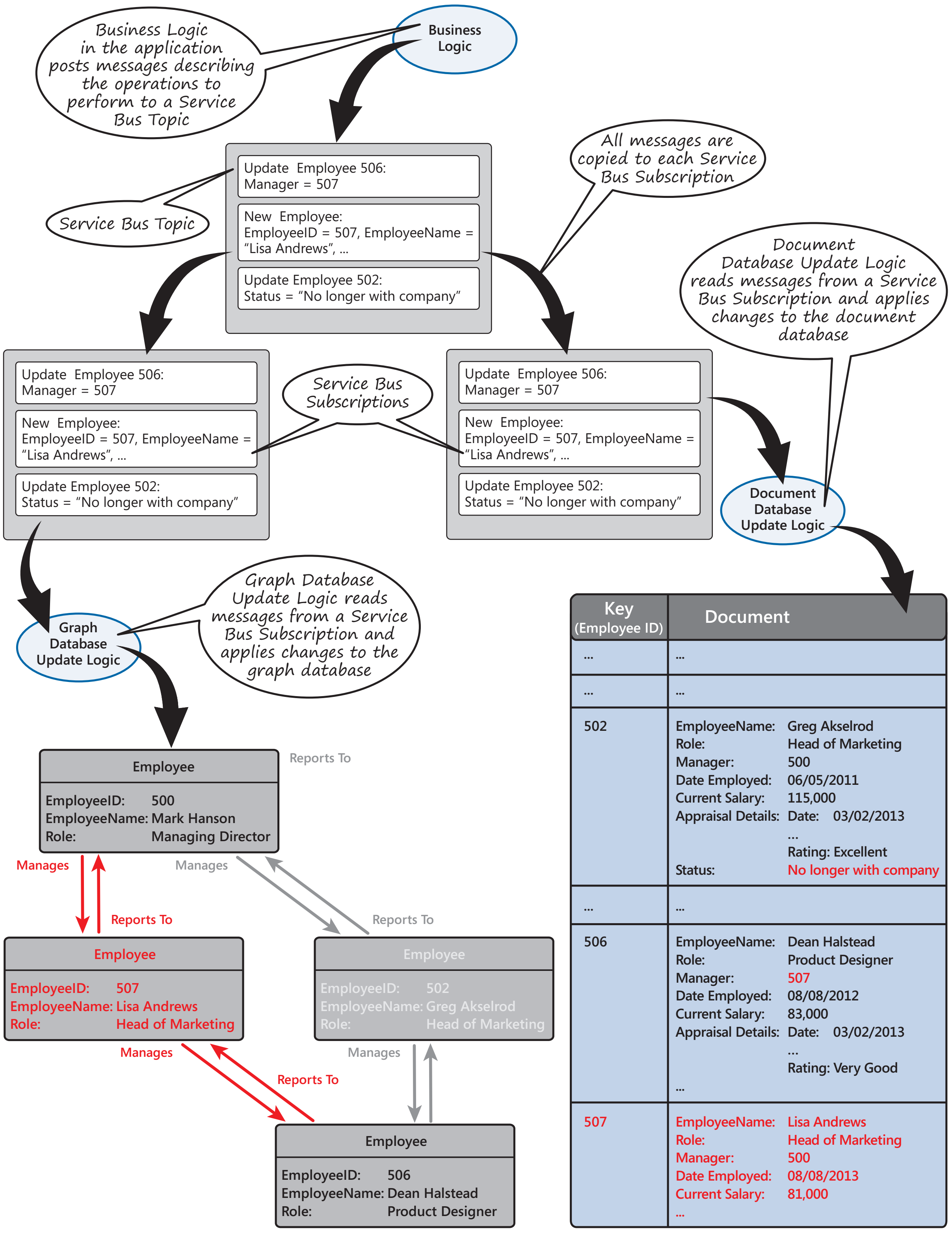

Business Logic
Business Logic in the application posts messages describing the operations to perform to a Service Bus Topic
Service Bus Topic
Update Employee 506: Manager = 507
New Employee: EmployeeID = 507, EmployeeName = "Lisa Andrews", ...
Update Employee 502: Status = "No longer with company"
All messages are copied to each Service Bus Subscription
Document Database Update Logic reads messages from a Service Bus Subscription and applies changes to the document database
Service Bus Subscriptions
Update Employee 506: Manager = 507
New Employee: EmployeeID = 507, EmployeeName = "Lisa Andrews", ...
Update Employee 502: Status = "No longer with company"
Update Employee 506: Manager = 507
New Employee: EmployeeID = 507, EmployeeName = "Lisa Andrews", ...
Update Employee 502: Status = "No longer with company"
Document Database Update Logic
Graph Database Update Logic
Graph Database Update Logic reads messages from a Service Bus Subscription and applies changes to the graph database
Employee
EmployeeID: 500
EmployeeName: Mark Hanson
Role: Managing Director
Reports To
Manages
Manages
Reports To
Reports To
Employee
EmployeeID: 507
EmployeeName: Lisa Andrews
Role: Head of Marketing
Employee
EmployeeID: 502
EmployeeName: Greg Akselrod
Role: Head of Marketing
Manages
Reports To
Manages
Employee
EmployeeID: 506
EmployeeName: Dean Halstead
Role: Product Designer
Key (Employee ID)
Document
... ...
... ...
502 EmployeeName: Greg Akselrod; Role: Head of Marketing; Manager: 500; Date Employed: 06/05/2011; Current Salary: 115,000; Appraisal Details: Date: 03/02/2013 ... Rating: Excellent; Status: No longer with company
... ...
506 EmployeeName: Dean Halstead; Role: Product Designer; Manager: 507; Date Employed: 08/08/2012; Current Salary: 83,000; Appraisal Details: Date: 03/02/2013 ... Rating: Very Good ...
507 EmployeeName: Lisa Andrews; Role: Head of Marketing; Manager: 500; Date Employed: 08/08/2013; Current Salary: 81,000 ...

Figure 5
Using a service bus to synchronize databases as changes occur

The reliable messaging capabilities of the topic, coupled with the guaranteed delivery semantics of each subscription ensure that updates are not lost, even if one or more of the subscribing processes should fail. In this case, when the subscriber restarts, the message describing the update will have been returned to the queue and can be read again. In this way, the service bus becomes the primary point of integration between databases, and you do not have to include complex failure detection and retry logic in the business logic of your application code. However, undoing an operation can be a challenge. In a typical SQL transaction, if the business logic detects that a transaction should not be allowed to complete for whatever reason, it can simply abort the transaction and all work performed will be rolled back. In a distributed environment based on messaging, if a subscriber determines that the operation should be undone, it may need to post one or more messages to the service bus topic that cause the subscribers to reverse the effects of the work that they have performed.

Summary

This chapter has examined some of the important issues that arise when you design and implement a polyglot solution. In particular, it has focused on strategies that you can employ to partition your data across different types of NoSQL databases, and techniques that you can use for implementing the application logic that needs to ensure consistency across different databases.

Different NoSQL databases are often designed and optimized to support specific data access scenarios. The advantage of using a collection of different NoSQL databases is that you can exploit their strengths and match them to the requirements of the various business scenarios that your applications need to support. The cost of this approach is the additional complexity and infrastructure required to combine the different technologies into a seamless solution. Using the Repository pattern can help to isolate the data access code from the business logic of your applications, but maintaining consistency across a collection of nonhomogeneous databases can be a significant challenge. If your data must be immediately and absolutely consistent all of the time, then you may need to revert to a transactional solution based on a single relational database, although this approach will inevitably limit the scalability of your solution. However, in many cases, immediate consistency is not a business requirement, and you can implement eventual consistency by synchronizing data periodically or by using events to propagate changes as they occur, depending on the latency that your business is willing to tolerate.

More Information

All links in this book are accessible from the book's online bibliography on MSDN at: *http://msdn.microsoft.com/en-us/library/dn320459.aspx*.

The *Repository* pattern is described in detail in "Patterns of Enterprise Application Architecture" website on Martin Fowler's website.

The article "*The Repository Pattern*" on MSDN describes an implementation of the Repository pattern by using SharePoint, but you can apply the general principles to almost any data store.

You can find more information about Windows Azure Service Bus on the "*Messaging*" page of the Windows Azure website.

The section "*Introducing Event Sourcing* " in the guide "*CQRS Journey* ", available on the MSDN website contains detailed information about using the Event Sourcing pattern to record the history of the changes made to the data held by an application and using this information to generate a view of the system state.

Appendix A How the MvcWebApi Web Service Works

The purpose of the MvcWebApi web service is to take REST requests sent from the user interface web application, validate these requests by applying the appropriate business logic, and then convert them into the corresponding CRUD (create, retrieve, update, and delete) operations in the various databases that contain information used by the Shopping application. This appendix describes how the developers designed and implemented the MvcWebApi web service to perform these tasks.

Understanding the Flow of Control Through the Web Service

This section is a high level overview of the design of the MvcWebApi web service. The remaining sections in this appendix describe in detail how the elements that comprise the MvcWebApi web service operate.

The developers at Adventure Works needed the web service to provide the functionality to support the core business cases described in Chapter 2, "The Adventure Works Scenario." The system uses a collection of SQL and NoSQL databases, but the developers wanted to design a system that decoupled the data storage technologies from the business logic of the web service. This approach enables Adventure Works to switch a particular repository to use a different data storage technology if necessary, without impacting the business logic in the web service. It also helps to ensure that the low level details of the various databases are not visible to the user interface web application, and which could result in accidental dependencies between the user interface and the databases.

The MvcWebAPI web service exposes its functionality through a series of controllers that respond to HTTP REST requests. The section "Routing Incoming Requests to a Controller" later in this appendix describes the controllers and the requests that they support.

The business logic inside each controller stores and retrieves information in the database by using the Repository pattern. This pattern enabled the developers to isolate the database-specific aspects of the code inside a repository class, potentially allowing them to build different repository classes for different databases and easily switching between them. Each repository class manages the details of its connection to the underlying database, and takes responsibility for formatting the data in the manner expected by the database. The section "Using the Repository Pattern to Access Data" later in this appendix provides the details on how Adventure Works implemented this pattern in the web service.

To enable the code to use different repositories without requiring that the web service is rebuilt and redeployed, the developers decided to use the Unity Application Block. The Unity Application Block allows a developer (or an administrator) to specify how an application should resolve references to objects at runtime rather than at compile time. The section "Instantiating Repository Objects" later in this appendix contains a description of how the developers at Adventure Works designed the MvcWebApi web service to support dynamic configuration by using the Unity Application Block.

The data that passes between the controllers and the repository classes are *domain entity objects*. A domain entity object is a database-neutral type that models business-specific data. It provides a logical structure that hides the way that the database stores and retrieves the data. The details of the domain entity objects used by the web service are described in the section "Decoupling Entities from the Data Access Technology" later in this appendix.

The response messages that the controllers create and send back to the user interface web application are HTTP REST messages, in many cases wrapping the data that was requested by the user interface web application, following the Data Transfer Object (DTO) pattern. The section "Transmitting Data Between a Controller and a Client" later in this appendix contains more information about the DTO objects created by the MvcWebApi web service.

Figure 1 illustrates the high level flow of control through the MvcWebApi web service. This figure depicts the user interface web application sending a request that is handled by the **Products** controller in the MvcWebApi web service. This controller retrieves information about products and product recommendations by using **Product** repository and **ProductRecommendation** repository objects. The repository objects can connect to the same database or to different databases (in the Shopping cart application, product information is held in a document database, and product recommendations are stored in a graph database). The data access logic is incorporated into a series of database-specific classes. The repository objects convert the information from a database-specific format into a neutral entity objects. The controller uses the entity objects to construct DTOs that it passes back in the REST responses to the user interface web application.

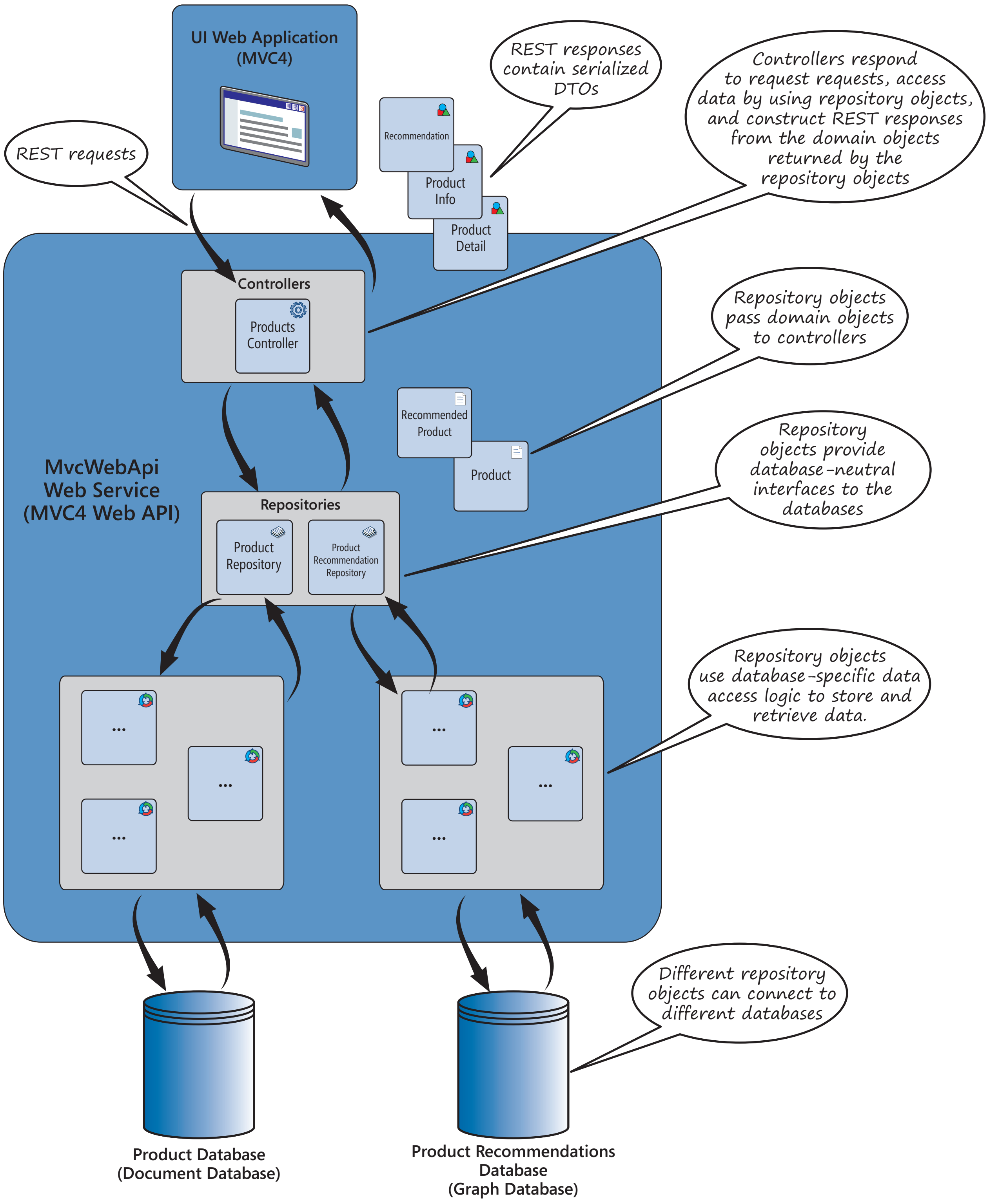

UI Web Application (MVC4)
REST requests
REST responses contain serialized DTOs
Recommendation
Product Info
Product Detail
Controllers respond to request requests, access data by using repository objects, and construct REST responses from the domain objects returned by the repository objects
Controllers
Products Controller
Repository objects pass domain objects to controllers
Recommended Product
Product
MvcWebApi Web Service (MVC4 Web API)
Repositories
Product Repository
Product Recommendation Repository
Repository objects provide database-neutral interfaces to the databases
Repository objects use database-specific data access logic to store and retrieve data.
...
Different repository objects can connect to different databases
Product Database (Document Database)
Product Recommendations Database (Graph Database)

Figure 1
The high-level flow of control through the MvcWebApi web service

IMPORTANT

The UI web application generates anti-forgery tokens that help to protect critical operations such as registering as a new customer, logging in, and placing an order (other operations, such as viewing the product catalog, expose data that is in the public domain and are not protected). However, the MvcWebApi web service currently implements only minimal security measures. Any application that knows the URL of the web service can connect to it and send requests. A real world implementation of this system would include more robust authentication and prevent unauthorized use, but in this reference implementation the additional code would obscure the data access guidance that this application is designed to convey.

Additionally, although the structure of the MvcWebApi separates the business logic of the web service out into discrete functional areas, this separation is driven by the implementation of the controllers and is not an attempt to implement a strict domain driven design (DDD).

Routing Incoming Requests to a Controller

The MvcWebApi web service is implemented in the DataAccess.MvcWebApi project in the solution provided with this guide.

In common with most ASP.NET MVC4 applications, all incoming REST requests are routed to a controller based on the URI that the client application specifies. The controllers are implemented by using the controller classes defined in the Controllers folder of the DataAccess.MvcWebApi project. The following table describes each controller class and the public methods that they each contain:

Controller Class	Description
AccountController	Implements services that authenticate users and provide access to customer accounts and related information. This controller exposes the following operations: • **Get**. Returns the details for the customer that matches a specified customer ID. • **Login**. Authenticates the specified username and password. • **Register**. Adds the details of a new customer to the database.
CartController	Provides the following services that implement shopping cart functionality: • **Get**. Returns the list of items in the shopping cart that has a specified shopping cart ID. • **Add.** Adds the specified item to the shopping cart. • **DeleteCartItem**. Removes the specified item from the shopping cart.
CategoriesController	Implements services that provides access to the product categories in the database: • **GetAll**. Returns a list containing the details of every product category.
OrdersController	Provides the following operations that enable an application to retrieve order information and place an order for a customer: • **Get**. Returns the details of the specified order. • **Post**. Creates an order from the items in the customer's shopping cart and submits it to the order service which processes and handles the order. *Note that in the sample application, the order service simulates the process of handling an order but it illustrates good practice for handling idempotency and eventual consistency when you need to implement operations that span non-transactional data stores. The order service is described in detail in Chapter 8, "Building a Polyglot Solution."* • **GetHistory**. Returns the list of order history documents for the customer. The document database maintains a full audit trail of each order for every customer. • **GetHistoryDetails**. Returns the details contained in an order history document.

Controller Class	Description
ProductsController	Implements functionality that provides access to the product information in the database: • **GetProducts**. Returns a list containing the details of every product in a specified product subcategory. • **Get**. Returns the details of the product with a specified product ID. • **GetRecommendations**. Returns a list of products that customers who purchased the specified product also bought.
StatesController	Provides the following service that is used to return information about states and provinces (referenced by customer addresses) in the database: • **Get**. Returns a list containing the details of every state and province in the database.
SubcategoriesController	Implements the following operation that provides access to the subcategory information in the database: • **GetSubcategories**. Returns a list containing the details of every subcategory in a specified category.

*The MvcWebApi web service also defines the **HomeController** class. This class is the controller that is associated with the root URI of the web service, and it generates the default **Home** view for the website. This view displays a welcome page describing the ASP.NET Web API. The **HomeController** class and the **Home** view are not described further in this appendix.*

The routes exposed by the MvcWebApi web service are defined in the static **Register** method of the **WebApiConfig** class, in the App_Start folder of the DataAccess.MvcWebApi project. The code example below shows the **Register** method of the **WebApiConfig** class, and illustrates how requests are routed to the appropriate controller:

```
public static void Register(HttpConfiguration config)
{
    // Subcategories
    config.Routes.MapHttpRoute(
        name: "Subcategories",
        routeTemplate: "api/categories/{categoryId}/subcategories",
        defaults: new {controller="Subcategories", action="GetSubcategories"});

    // Products
    config.Routes.MapHttpRoute(
        name: "Products",
        routeTemplate: "api/subcategories/{subcategoryId}/products",
        defaults: new { controller = "Products", action = "GetProducts" });

    config.Routes.MapHttpRoute(
        name: "Recommendations",
        routeTemplate: "api/products/{productId}/recommendations",
        defaults: new { controller = "Products", action = "GetRecommendations" });

    // Orders
    config.Routes.MapHttpRoute(
        name: "OrdersHistory",
        routeTemplate: "api/account/{personId}/orders",
        defaults: new { controller = "Orders", action = "GetHistory" });
        config.Routes.MapHttpRoute(
            name: "OrderHistoryDetails",
            routeTemplate: "api/orders/history/{historyId}",
            defaults: new { controller = "Orders", action = "GetHistoryDetails" });

        // Default
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{action}/{id}",
            defaults: new { action = RouteParameter.Optional,
                            id = RouteParameter.Optional },
            constraints: new { action = "[a-zA-Z]+" });

        config.Routes.MapHttpRoute(
            name: "DefaultApiFallback",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional });

        ...
    }
```

The following table summarizes these routes and the data that they pass to the relevant controller, together with the expected response from the controller. Note that all routes begin with the literal “api”.

Route name	Controller	Examples	Description
Subcategories	**Subcategories**	api/categories/1/ subcategories	Returns a JSON array containing a list of subcategories for the specified category, or the **Not Found** HTTP status code if there is no subcategory with the specified ID. The example returns an array containing the subcategories for the Bikes category (in the database, Bikes is category 1.)
Products	**Products**	api/subcategories/ 3/products	Returns a JSON array containing a list of products for the specified subcategory, or the **Not Found** HTTP status code if there is no matching subcategory. The example returns an array of bicycles from the Touring Bikes subcategory (subcategory 3 in the database.)
Recommendations	**Products**	api/products/710/ recommendations	Returns a JSON array containing a list of products and ratings that other customers who purchased the specified product also bought, or the **Not Found** HTTP status code if there is no matching product. The example returns the list of products (and their ratings) that were bought by customers who also bought product 710.
OrdersHistory	**Orders**	api/account/ 156dfc64-abb3- 44d4-a373- 1884ba077271 / orders	Returns a JSON array containing a list of orders placed by the customer with the specified ID. If there is no matching customer, then the request returns the **Not Found** HTTP status code. The example returns the list of orders for customer 156dfc64-abb3-44d4-a373-1884ba077271.

Route name	Controller	Examples	Description
OrderHistoryDetails	**Orders**	api/orders/history/8d068e00-824d-4dff-8afe-552e367a7d6e	Returns a JSON object containing the details of the specified order history record. If there is no matching order history record, then the request returns the **Not Found** HTTP status code. In the example, note that the ID of the order history record is a GUID.
DefaultApi and DefaultApiFallback	*All*	api/categories api/products/706	These routes capture all requests that do not match any of the preceding routes. The request is routed to the named controller and invokes the specified action. Without the optional trailing ID, these routes respond with a JSON array containing the objects returned by the specified *GetAll* action of the specified controller. With the optional trailing ID, this route responds with a JSON object containing the object returned by the *Get* action of the specified controller, passing the ID as the parameter to this action. The first example returns the list of product categories in the Adventure Works database. The second example returns the details of product 706.
Subcategories-Controller			Implements the following operation that provides access to the subcategory information in the database:

The order in which the routes are configured in the **Register** method defines their precedence for cases where the same URI matches more than one route.

For more information about defining routes for Web API web services, see "Routing in ASP.NET Web API" on the ASP.NET website.

Transmitting Data Between a Controller and a Client

The controllers receive REST requests and send REST responses. The data received in these requests and sent in these responses can contain structured information, such as the details of a product, the items in a shopping cart, or the address of a customer. The MvcWebApi web service defines a series of serializable data types that define the shape of this structured information. These types act as data transfer objects (DTOs), and the information that they contain is transmitted as JSON objects and JSON arrays. These DTOs are implemented in the Models folder in the DataAccess.MvcWebApi project. The following table briefly describes these classes:

The MvcWebApi web service is configured to accept and handle cross-origin resource sharing (CORS) requests. This feature enables the user interface web application to run in a separate web domain from the MvcWebApi web service.

Model Class	Description
AddressInfo	Contains a street address, including the city, state, country, and postal code.
CartItem	Specifies the details of an item in the shopping cart. The properties include the shopping cart ID, the product ID, the product name, the quantity, and the price of the product.
CartItemDelete	Contains a reference to an item to remove from a shopping cart.
Category	Holds the name of the category and its ID.
CreditCardInfo	Contains the credit card type and credit card number, as strings.
LoginInfo	Contains the credentials of a user (user name and password).
OrderDetail	Specifies the complete details of an order. This class includes a list of **OrderItemInfo** objects that define each line item for the order (for details, see the **OrderItemInfo** class described later in this table).

Model Class	Description
OrderHistoryInfo	Contains a reference to an order together with the date and time that the order was placed or last modified.
OrderInfo	Specifies the summary details of an order. This class simply contains a set of IDs that reference the shopping cart containing the products being ordered, the shipping address and billing address of the customer, and the credit card used to pay for the order.
OrderItemInfo	Contains the information for a line item in an order; the product ID, product name, unit price, and quantity.
ProductDetail	Holds the details of a product, such as the product name, product number, color, list price, size, and the weight.
ProductInfo	Specifies the price of a product together with the category and subcategory to which the product belongs.
Recommendation	Holds the name and ID of a product, together with its recommendation rating.
RegistrationInfo	Contains the information needed to register a new customer, including the first name, last name, email address, password, address, and credit card details.
Subcategory	Holds the name of the subcategory, its ID, and the category to which it belongs.

Note: *Chapter 7, "Implementing a Graph Database," describes how the recommendation ratings for a product are calculated.*

*The **Category**, **ProductDetail**, **ProductInfo**, **Recommendation**, and **Subcategory** classes inherit from the abstract **ModelBase** class that provides the fields (**Id**, **Name**, and **ParentId**) that are common to these classes.*

The MvcWebApi web service uses AutoMapper to convert the domain entity objects that it receives from the various repository objects into DTOs. AutoMapper simplifies the process of translating objects from one format into another by defining a set of mapping rules that can be used throughout an application. In the MvcWebApi web service, these rules are defined in the AutoMapperConfig.cs file in the App_Start folder in the DataAccess.MvcWebApi project.

As an example, the **Product** domain entity type that contains the information returned by method in the **ProductRepository** class looks like this:

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string ProductNumber { get; set; }
    public string Color { get; set; }
    public decimal ListPrice { get; set; }
    public string Size { get; set; }
    public string SizeUnitMeasureCode { get; set; }
    public decimal? Weight { get; set; }
    public string WeightUnitMeasureCode { get; set; }
    public string Class { get; set; }
    public string Style { get; set; }
    public Subcategory Subcategory { get; set; }
}
```

The **Product** domain entity type contains a reference to the subcategory in which it is contained. In turn, the subcategory contains a reference to the category to which it belongs. The **Subcategory** and **Category** domain entity types look like this:

```
public class Subcategory
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Category Category { get; set; }
}

public class Category
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

The **GetProducts** method of the **ProductsController** transmits **ProductInfo** objects back to the client as DTOs. The **ProductInfo** class looks like this:

```
public class ProductInfo : ModelBase
{
    public decimal ListPrice { get; set; }
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public string SubcategoryName { get; set; }
}
```

A **ProductInfo** DTO is a flattened structure that contains information from **Product**, **Subcategory**, and **Category** domain entity objects. The AutoMapperConfig.cs file contains mappings that specify how to populate a **ProductInfo** DTO from a **Product** entity object, as shown by the following code example:

```
public static void SetAutoMapperConfiguration()
{
    ...
    // define the mapping from the Product domain entity to the ProductInfo DTO
    Mapper.CreateMap<DE.Catalog.Product, DTO.ProductInfo>()
        .ForMember(dest => dest.CategoryId, src => src.MapFrom(
            dest => dest.Subcategory == null ? 0 :
                (dest.Subcategory.Category == null ? 0 :
                    dest.Subcategory.Category.Id)))
        .ForMember(dest => dest.CategoryName, src => src.MapFrom(
            dest => dest.Subcategory == null ? null :
                (dest.Subcategory.Category == null ? null :
                    dest.Subcategory.Category.Name)))
        .ForMember(dest => dest.SubcategoryName, src => src.MapFrom(
            dest => dest.Subcategory == null ? null : dest.Subcategory.Name));
    ...
}
```

This mapping specifies that the **CategoryId** property in the DTO should be read from the **Id** property of the **Category** object referenced by the **Subcategory** property of the domain entity object as long as the Category and **Subcategory** properties reference valid objects. If not, the default value of 0 is used instead. In a similar manner, the **CategoryName** property is populated from the **Name** property of the **Category** object referenced by the **Subcategory** property of the domain entity object, or is set to **null** if the **Category** or **Subcategory** objects are **null**. The **SubcategoryName** property is copied from the **Name** property of the **Subcategory** property of the domain object. Again a null value is used if this property does not reference a valid object. The values of any properties not explicitly mapped (such as the **ListPrice**) are automatically copied from the domain entity object to the DTO as long as they have the same name in the domain entity object and the DTO.

The **GetProducts** method in the **ProductsController** uses the static **Map** method of the **Mapper** class (this is the AutoMapper that invokes the rules in the AutoMapperConfig.cs file) to translate a collection of **Product** domain entity objects returned by the product repository into a collection of **ProductInfo** DTOs that it sends back in the REST response to the client:

```
public HttpResponseMessage GetProducts(int subcategoryId)
{
    ...
    var products = this.productRepository.GetProducts(subcategoryId);
    if (products == null)
    {
        return Request.CreateErrorResponse(HttpStatusCode.NotFound,
             string.Format(CultureInfo.CurrentCulture,
                           Strings.SubcategoryNotFound));
    }

    var productInfos = new List<ProductInfo>();

    Mapper.Map(products, productInfos);

    return Request.CreateResponse(HttpStatusCode.OK, productInfos);
}
```

For more information about AutoMapper, see [AutoMapper]() on github.com.

The other controllers use the same approach for generating DTOs.

Using the Repository Pattern to Access Data

The Repository pattern enables you to separate the code that accesses a data store from the business logic that uses the data in this store, minimizing any dependencies that the controller might have on the data store. Each of the controllers creates one or more repository objects to connect to the various databases and retrieve, create, update, or delete data. For example, the following code shows the **GetAll** method in the **CategoriesController** class that uses the **GetAllCategories** method of a **CategoryRepository** object to fetch category information:

```
public class CategoriesController : ApiController
{
    private ICategoryRepository categoryRepository;

    public CategoriesController(ICategoryRepository categoryRepository)
    {
        this.categoryRepository = categoryRepository;
    }

    public HttpResponseMessage GetAll()
    {
        var categories = this.categoryRepository.GetAllCategories();
        var result = new List<Category>();

        Mapper.Map(categories, result);

        return Request.CreateResponse(HttpStatusCode.OK, result);
    }
    ...
}
```

Each repository object is an instance of a database-specific repository class. The repository classes implement a set of database-agnostic interfaces. The methods defined by the repository interfaces return or manipulate collections and instances of domain entity objects. The section "Decoupling Entities from the Data Access Technology" later in this appendix describes these types. The following table summarizes the repository interfaces:

The repository interfaces are defined in the DataAccess.Repo.Interface project in the Repo folder of the solution code provided with this guide.

Repository Interface	Methods	Description
ICategoryRepository	**GetAllCategories** **GetSubcategories**	This repository interface provides access to product category and subcategory information. The **GetAllCategories** method returns a list containing the details of every category as a collection of **Category** domain entity objects. The **GetSubcategories** method returns a collection of **Subcategory** objects, listing all the subcategories in the specified category. Category and subcategory information is read only in the Shopping application, so this class does not provide any means to save changes made to **Category** domain entity objects back to the database.
IInventoryProduct-Repository	**GetInventoryProduct**	This repository interface defines methods that retrieve inventory information about products. The **GetInventoryProduct** method takes a product ID and returns an **InventoryProduct** object that contains the current price of the product from the inventory database.
IOrderHistoryRepository	**GetOrderHistoryBy-TrackingId** **GetOrdersHistoriesy** **GetOrderHistoryBy-HistoryId** **IsOrderCompleted** **GetPendingOrderBy-TrackingId** **SaveOrderHistory**	This repository interface defines methods that store and retrieve order history information. The **GetOrderHistoryByTrackingId** method returns an **Order** domain entity object that matches the specified order code. Each order has its own order code that identifies the order. The **GetOrdersHistories** method returns a collection of **OrderHis-tory** domain entity objects containing complete audit history of all orders for a customer. The **GetOrderHistoryByHistoryId** method returns an **OrderHis-tory** domain entity object containing the details from a single order document. The **IsOrderCompleted** method examines the status of the specified order and returns true if the status indicates that the order has been completed, false otherwise. The **GetPendingOrderByTrackingId** method returns an **Order** domain entity object containing the details of the order that matches the specified tracking ID and that has a status of **Pending**, or null if there is no such order. The **SaveOrderHistory** method writes an **OrderHistory** domain entity object to the database.
ISalesOrderRepository	**SaveOrder** **UpdateOrderStatus** **IsOrderSaved**	This repository interface manages order information in the database. The **SaveOrder** method takes an **Order** domain entity object and uses it to create and populate an order in the database. The **UpdateOrderStatus** method takes the ID of an order and a new order status, and updates the specified order to have this status in the database. The **IsOrderSaved** method takes the ID of an order and returns a Boolean value indicated whether an order with this ID has already been submitted for processingsaved to the database.

Repository Interface	Methods	Description
IPersonRepository	**GetPerson** **GetPersonByEmail** **SavePerson**	This repository interface provides access to the details of customers. The **GetPerson** method returns a **Person** domain entity object containing the details of the customer that matches the ID specified as the parameter to this method. The **GetPersonByEmail** method performs a similar task, except that it matches customers by using their email address. The **SavePerson** method is used when a new customer is registered. This method saves the registration details (held in a **Person** domain entity object) to the database.
IProductRecommendation-Repository	**GetProduct-Recommendations**	This repository interface enables an application to retrieve product recommendations from the database. The **GetProductRecommendations** method takes a product ID as the parameter and returns a collection of **RecommendedProduct** domain entity objects. This collection contains information about products that other customers who bought the specified product also purchased.
IProductRepository	**GetProducts** **GetProduct** **ProductExists**	This repository interface defines methods that return information about the products that customers can order. The **GetProducts** method retrieves a list of all products in a specified subcategory specified as a parameter to this method. The list is returned as a collection of **Product** domain entity objects. The **GetProduct** method returns a single **Product** domain entity object that matches the product ID specified as the parameter to this method. The **ProductExists** method takes a product ID and returns a Boolean value indicating whether a product with this ID exists in the product catalog. In the Shopping application, product information is read only so this repository does not provide any means to save changes to products back to the database.
IShoppingCartRepository	**GetShoppingCart** **SaveShoppingCart** **DeleteShoppingCart**	This repository interface provides access to the contents of a customer's shopping cart. The **GetShoppingCart** method returns the contents of the specified shopping cart as a **ShoppingCart** domain entity object. The **SaveShoppingCart** method saves the shopping cart to the database. The **DeleteShoppingCart** method removes the shopping cart items from the database specified by the shopping cart ID passed as the parameter to this method. Note that the shopping cart ID is actually the same as the customer's user ID because a customer can only have one shopping cart.
IStateProvinceRepository	**GetStateProvince** **GetStateProvinces**	This repository retrieves the information about the states and provinces that can be included in customers' addresses. The **GetStateProvince** method returns a **StateProvince** domain entity object with an ID that matches the parameter value passed in. The **GetStateProvinces** method returns a collection of **State-Province** domain entity objects, listing every state or province in the database.

The sample solution supports a number of different databases, and each database provides its own implementation of one or more of these interfaces containing database-specific code. Note that not all databases implement every repository; repositories are only provided where it is useful to store the data made available through that repository in a particular database.

The following section summarizes the repository classes that the developers at Adventure Works created for storing and retrieving order, person, and inventory data from SQL Server. These classes connect to the database by using the Entity Framework 5.0.

How the Entity Framework Repository Classes Work

The repository classes for the Entity Framework, and the data types that these repository classes use to retrieve and modify data in the SQL Server database, are defined in the DataAccess.Repo.Impl.Sql project in the Repo folder.

The DataAccess.Repo.Impl.Sql project contains four repository classes named **SalesOrderRepository, PersonRepository**, **InventoryProductRepository**, and **StateProvinceRepository**. These classes provide the business methods that the controller classes use to store and retrieve data. Internally, these repository classes save and fetch data by using a *context* object (**SalesOrderContext**, **PersonContext**, **InventoryProductContext**, or **StateProvinceContext** as appropriate). Each context object is responsible for connecting to the database and managing or retrieving the data from the appropriate tables.

How the Context Classes Store and Retrieve Data from the Database

The context classes are defined in the DataAccess.Repo.Impl.Sql project, in the Repo folder in the solution code (the code is stored in the Order, Person, and StateProvince folders in this project). They are all custom Entity Framework context objects, derived indirectly from the **DbContext** class (via the **BaseContext** generic class) that exposes the data from the SQL Server database through a group of public **IDbSet** collection properties. The following code example shows how some of the properties for handling customer information are defined in the **PersonContext** class:

```
public class PersonContext : BaseContext<PersonContext>, IPersonContext
{
    ...
    public IDbSet<Person> Persons { get; set; }
    public IDbSet<PersonEmailAddress> EmailAddresses { get; set; }
    public IDbSet<PersonAddress> Addresses { get; set; }
    public IDbSet<PersonCreditCard> CreditCards { get; set; }
    ...
}
```

The type parameters referenced by the **IDbSet** collection properties are entity classes that are specific to the Entity Framework repositories; they correspond to individual tables in the SQL Server database. These classes are defined in the same folders of the DataAccess.Repo.Impl.Sql project (Order, Person, and StateProvince) as the various context classes. The following table briefly summarizes these entity classes:

Folder\Entity Class	**Description**
Person\Person	This class contains the data that describes a customer. The data is exposed as public properties. Many of the properties correspond to fields in the **Person** table in the SQL Server database, and relationships between the **Person** table and other tables such as **PersonCreditCard**, **EmailAddress**, and **Address** are implemented as collections of the appropriate entity object.
Person\PersonAddress	This class contains the data that describes the address of a customer.
Person\PersonBusinessEntity and **Person\PersonBusinessEntityAddress**	These two classes provide the connection between **Person** entities and **Address** entities. A **PersonBusinessEntity** represents a **Person**, and the **PersonBusinessEntityAddress** entity implements a one-to-many relationship between a **PersonBusinessEntity** object and a **PersonAddress** object.
Person\PersonCreditCard	This class holds the data that describes a credit card for a customer.
Person\PersonEmailAddress	This class describes the email address of a customer.
Person\PersonPassword	This class contains the password for the customer. For security reasons, the password is held as a hash in the **PasswordHash** property rather than as clear text. The **PasswordSalt** property contains the random salt string appended to the password before it is hashed.
StateProvince\StateProvince	This class contains information about the states and provinces that can be included in customer addresses.
Order\SalesOrderHeader	This class contains the information that describes a customer's order. It includes data such as the billing address and shipping address of the customer (held as **PersonAddress** objects), a reference to the credit card that was used to pay for the order, the total value of the order, and a collection of **SalesOrderDetail** objects describing the line items for the order.
Order\SalesOrderDetail	This class describes a line item for an order, indicating the product, quantity, and unit price amongst other details.
Order\InventoryProduct	This class contains the inventory information for a product required by the order service. This class contains the current price of a specified product. The **InventoryService** object used by the **Orders** controller retrieves an **InventoryProduct** object that contains the current price for each item in a new order before the order is placed. The **Orders** controller uses this information to determine whether the price of an item has changed since the customer added it to their shopping cart.

The developers at Adventure Works followed a code first approach to implementing the entity model for the repositories and they defined each of the entity classes manually. For example, the **Person** and **PersonEmailAddress** classes look like this:

```
public class Person : ...
{
    public string PersonType { get; set; }
    public bool NameStyle { get; set; }
    public string Title { get; set; }
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string Suffix { get; set; }
    public int EmailPromotion { get; set; }
    public Guid PersonGuid { get; set; }
    public virtual PersonPassword Password { get; set; }
    public virtual ICollection<PersonEmailAddress> EmailAddresses { get; set; }
    public virtual ICollection<PersonBusinessEntityAddress> Addresses {get; set;}
    public virtual ICollection<PersonCreditCard> CreditCards { get; set; }
}
```

```
public class PersonEmailAddress
{
    public int BusinessEntityId { get; set; }
    public int EmailAddressId { get; set; }
    public string EmailAddress { get; set; }
    public virtual Person Person { get; set; }
}
```

The **PersonContext** class uses the Entity Framework Fluent API to map the entity classes to tables in the SQL Server database and specify the relationships between tables. The following sample shows the code in the **OnModelCreating** event handler method for the **PersonContext** class that populates the **Persons** and **EmailAddresses IDbSet** collections. This event handler runs when the **PersonContext** object is created, and it passes a reference to the in-memory data model builder that is constructing the model for the Entity Framework as the parameter:

```
public sealed class PersonContext : ...
{
    public IDbSet<Person> Persons { get; set; }
    public IDbSet<PersonEmailAddress> EmailAddresses { get; set; }
    ...
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        ...
        modelBuilder.Configurations.Add(new PersonMap());
        modelBuilder.Configurations.Add(new PersonEmailAddressMap());
        ...
    }
}
```

In this example the **OnModelCreating** method creates new **PersonMap** and **PersonEmailAddressMap** objects. The code in these types describes how to associate each of the entity classes with the corresponding tables in the database. They are defined in the PersonMap.cs and PersonEmailAddressMap.cs files in the Person folder of the DataAccess.Repo.Impl.Sql project. The following code sample shows how the **PersonEmailAddressMap** class is implemented:

```
public class PersonEmailAddressMap : EntityTypeConfiguration<PersonEmailAddress>
{
    public PersonEmailAddressMap()
        : base()
    {
        this.ToTable("EmailAddress", "Person");
        this.HasKey(e => e.EmailAddressId);

        this.HasRequired(e => e.Person)
            .WithMany()
            .WillCascadeOnDelete(true);
    }
}
```

The call to the **ToTable** method in this code specifies that the **DbModelBuilder** object should map instances of the **PersonEmailAddress** type to rows in the **EmailAddress** table in the **Person** schema in the SQL Server database. The call to the **HasKey** method specifies that the **EmailAddressId** field contains the primary key used to establish relationships with other objects (such as instances of the **Person** type). The **WithMany** method defines the type of relationship that a **PersonEmailAddress** object has with a **Person** object. In the Adventure Works database, a **Person** row in the database can have many associated **EmailAddress** rows, and if a **Person** row is deleted then all associated **Email-Address** rows should be deleted as well.

The **PersonMap** class shown in the following code example is more complicated due to the relationships that the underlying **Person** table has with the **Email-Address**, **Password**, **PersonCreditCard**, and **BusinessEntity** tables in the SQL Server database. Specifically, the **PersonMap** class stipulates that:

- The **Person** type maps to rows in the **Person** table in the **Person** schema in the SQL Server database,
- The **BusinessEntityId** field contains the primary key,
- The **Person** table in the database has a one-to-one relationship with rows in the **Password** table, and that the password for a person should be removed if the person is deleted,
- The **Person** table has a one-to-many relationship with the **EmailAddress** table, and that the email addresses for a person should be removed if the person is deleted, and
- The **Person** table has a many-to-many relationship with the **CreditCard** table implemented as a one-to-many relationship between the **Person** and **PersonCreditCard** tables, and a many-to-one relationship between the **PersonCreditCard** table and **CreditCard** table (this is a common technique for implementing many-to-many relationships in a relational database).

*See the section "How Adventure Works Designed the Database for the Shopping Application" in Chapter 3, "*Implementing a Relational Database*" for more information about the structure of the* **Person**, **EmailAddress**, **Password**, **PersonCreditCard**, *and* **BusinessEntity** *tables in the SQL Server database.*

```
public class PersonMap : EntityTypeConfiguration<Person>
{
    public PersonMap()
        : base()
    {
        this.ToTable("Person", "Person");
        this.HasKey(p => p.BusinessEntityId);

        this.HasRequired(p => p.Password)
            .WithRequiredPrincipal(p => p.Person)
            .WillCascadeOnDelete(true);

        this.HasMany(p => p.EmailAddresses)
            .WithRequired(e => e.Person)
            .WillCascadeOnDelete(true);

        this.HasMany(p => p.CreditCards)
            .WithMany(c => c.Persons)
            .Map(map => map.ToTable("PersonCreditCard", "Sales")
                .MapLeftKey("BusinessEntityID")
                .MapRightKey("CreditCardID"));
    }
}
```

For more information about constructing an entity model at runtime by using the Fluent API, see "Configuring/Mapping Properties and Types with the Fluent API" and "Configuring Relationships with the Fluent API" on MSDN.

How the Repository Classes Read and Save the Data for Objects

The repository classes read and save data through the context objects described in the previous section. To ensure consistency, all read operations are performed within the scope of a SQL Server transaction that uses the **ReadCommitted** isolation level. The repository classes inherit from the **BaseRepository** class that defines the **TransactionOptions** property and the **GetTransactionScope** method shown in the following code example:

```
public abstract class BaseRepository
{
    private static TransactionOptions transactionOptions =
        new TransactionOptions()
    {
        IsolationLevel = IsolationLevel.ReadCommitted
    };

    ...

    protected virtual TransactionScope GetTransactionScope()
    {
        return new TransactionScope(TransactionScopeOption.Required,
                                    transactionOptions);
    }
}
```

When a repository object retrieves data, it calls the **GetTransactionScope** method to create a **Transaction-Scope** object and reads the data within this scope as shown in the next code example.

In the context classes, each **IDbSet** property is an enumerable collection. This means that a repository object can retrieve data from a context object by performing LINQ queries. As an example, the following code shows the statements in the **GetPerson** method of the **PersonRepository** class that uses a **PersonContext** object to retrieve customer data from the database through the **Persons IDbSet** collection:

```
public class PersonRepository : BaseRepository, IpersonRepository
{
    public DE.Person GetPerson(Guid personGuid)
    {
        using (var context = new PersonContext())
        {
            Person person = null;

            using (var transactionScope = this.GetTransactionScope())
            {
                person = context.Persons
                    .Include(p => p.Addresses)
                    .Include(p => p.CreditCards)
                    .Include(p => p.EmailAddresses)
                    .Include(p => p.Password)
                    .SingleOrDefault(p => p.PersonGuid == personGuid);

                transactionScope.Complete();
            }
            ...
        }
    }
    ...
}
```

> *The data for a **Person** object includes address, credit card, email address, and password information that are retrieved from separate tables in the SQL Server database. The **Include** method ensures that these fields are populated when the data for the **Person** object is retrieved from the database (the Entity Framework performs lazy evaluation by default).*

Additionally, because they are descended from the Entity Framework **DbContext** class, the context classes implement the Unit of Work pattern, enabling the repository classes to combine operations that span multiple entities into a single transaction. All changes made to the collections attached to a context object are packaged up and converted into the corresponding SQL commands by the **SaveChanges** method inherited from the **DbContext** class. For example, the **SavePerson** method in the **PersonRepository** class creates and populates **PersonEmailAddress** objects and **PersonBusinessEntityAddress** objects containing the email addresses and business addresses of a customer that it adds to the **EmailAddresses** and **Addresses** collections in the **Person** object. The **Person** object is then added to the **Persons** collection in the **DbContext** object. The call to the **SaveChanges** method ensures that the new person, email address, and business entity address details are saved to the tables in the SQL Server database as an atomic transaction. The transaction runs by using the **Read-Committed** isolation level using a **TransactionScope** object created by the **GetTransactionScope** method. The code for the **SavePerson** method looks like this (the important statements are highlighted):

```
public class PersonRepository : BaseRepository, IPersonRepository
{
    ...
    public DE.Person SavePerson(DE.Person person)
    {
        var newPerson = new Person()
        {
            Addresses = new List<PersonBusinessEntityAddress>(),
            CreditCards = new List<PersonCreditCard>(),
            EmailAddresses = new List<PersonEmailAddress>(),
            Password = new PersonPassword()
        };

        Mapper.Map(person, newPerson);

        // add email addresses
        foreach (var emailAddress in person.EmailAddresses)
        {
            newPerson.EmailAddresses.Add(new PersonEmailAddress()
                { EmailAddress = emailAddress });
        }

        // add addresses
        foreach (var address in person.Addresses)
        {
            var personAddress = new PersonAddress();
            Mapper.Map(address, personAddress);

            newPerson.Addresses.Add(new PersonBusinessEntityAddress()
            {
                Address = personAddress,
                AddressTypeId = 2 // static value
            });
        }
```

```
            // since the PersonGuid is a storage implementation,
            // we create it now instead of earlier when the DE was created
            newPerson.PersonGuid = Guid.NewGuid();

            try
            {
                using (var context = new PersonContext())
                {
                    using (var transactionScope = this.GetTransactionScope())
                    {
                        context.Persons.Add(newPerson);
                        context.SaveChanges();
                        ...
                        transactionScope.Complete();
                    }
                }
            }
            catch (DbUpdateException ex)
            {
                ... // Handle errors that may occur when updating the database
            }

            return person;
        }
    }
```

How the Repository Classes for Other Databases Work

The MvcWebApi web service also contains repository classes for a variety of NoSQL databases. At a high level, they all function in a similar manner to the Entity Framework repository. They implement one or more of the repository interfaces, and they pass domain entity objects back to the controllers that use them. Internally, each of the repository classes can be quite different, and they use database-specific APIs to connect to the database to store and retrieve data.

Like the Entity Framework repositories, the developers implemented the repositories for each of the databases in separate projects under the Repo folder in the solution. The following table summarizes these projects and the repositories that they contain.

Data Storage Technology	Repository Classes	Description
Windows Azure Table service	**ShoppingCartRepository**	This repository class implements the **IShoppingCartRepository** interface to store and retrieve shopping cart information from the Windows Azure Table service. The data access logic is provided by **ShoppingCartContext** class in the DataAccess.Storage.Impl. TableService project. The data structures that define the storage classes for this repository are defined in the ShoppingCart folder. For more information, see the section "How Adventure Works Used a Key/Value Store to Hold Shopping Cart Information" in Chapter 4, "Implementing a Key/Value Store."
MongoDB	**CategoryRepository, OrderHistoryRepository,** and **ProductRepository**	These repository classes implement the **ICategoryRepository**, **IOrderHistoryRepository**, and **IProductRepository** interfaces to store product and order history information in a MongoDB document database. The Order and Catalog folders in the DataAccess.Repo.Impl.Mongo project contains the data types that the context classes use to store information in the database. For more information, see the section "How Adventure Works Used a Document Database to Store the Product Catalog and Order History Information" in Chapter 5, "Implementing a Document Database."
Neo4j	**ProductRecommendation-Repository**	This repository class implements the **IProductRecommendation-Repository** interface to retrieve product recommendation information from a Neo4j graph database. The Catalog folder in the DataAccess.Repo.Impl.Neo4j project defines the **ProductGraphEntity** class that specifies the structure of the information that the context class retrieves from the database. For more information, see the section "How Adventure Works Used a Graph Database to Store Product Recommendations" in Chapter 7, "Implementing a Graph Database."
In-Memory Storage	**CategoryRepository, OrderHistoryRepository, ProductRecommendation-Repository, Product-Repository,** and **Shopping-CartRepository**	These repository classes, defined in the DataAccess.Repo.Impl. InMemory folder, simulate NoSQL document and graph databases by using in-memory collections. They are provided to enable you to run the application without installing MongoDB or Neo4j if you do not have access to this software. They are not described further in this guide.

The details of the ShoppingCartRepository class for theWindows Azure Table service are described in Chapter 4, "Implementing a Key/Value Store," the details of the repository classes for MongoDB are described in Chapter 5, "Implementing a Document Database,", and the details of the ProductRecommendationRepository classe for Neo4j are described in Chapter 7, "Implementing a Graph Database."

Decoupling Entities from the Data Access Technology

As described earlier in this appendix, the data that passes between the controllers and the repository classes are instances of *domain entity objects*. These are database-neutral types that help to remove any dependencies that the controller classes might otherwise have on the way that the data is physically stored. The classes that implement these entities are organized into the Catalog, Order, Person, and ShoppingCart folders in the DataAccess.Domain project under the Domain folder. These types incorporate the business logic necessary to validate the data that they hold, also defined in a database-agnostic manner (this validation occurs before the data is saved to the database). For example, the **Person** class makes use of validation annotations, read only collections, and regular expressions, like this:

```
public class Person
{
    private readonly IList<string> emailAddresses = new List<string>();
    ...
    public int Id { get; set; }

    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    ...

    public IReadOnlyCollection<string> EmailAddresses {
        get { return new ReadOnlyCollection<string>(this.emailAddresses); } }
    ...

    public void AddEmailAddress(string emailAddress)
    {
        if (string.IsNullOrWhiteSpace(emailAddress))
        {
            throw new ArgumentNullException(
                "Email Address parameter cannot be null or white space");
        }

        var regex =
            new Regex(@"^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}$");
        var match = regex.Match(emailAddress);
        if (!match.Success)
        {
            throw new ValidationException(string.Format(
                "Invalid email address: {0}", emailAddress));
        }

        this.emailAddresses.Add(emailAddress);
    }
    ...
}
```

The repository classes in the MvcWebApi web service pass domain entity objects to the controllers. Therefore, an important task of the repository classes is to take the data that they retrieve from a database and reformat it as one or more of these domain entity objects. The repository classes for the Entity Framework and MongoDB use AutoMapper for this purpose. Note that this is the same technology that is used by the controllers to convert domain entity objects into DTOs—see the section "Transmitting Data Between a Controller and a Client" earlier in this appendix for more information. The repository classes for the Windows Azure Table service and Neo4j implement their own manual mapping functions; the nature of the database-specific types for these repositories is not best suited to using AutoMapper; the data is retrieved in a serialized form and has to be deserialized before it can be converted.

How the Entity Framework and MongoDB Repository Classes use AutoMapper to Convert Data

The projects that implement repositories for the Entity Framework and MongoDB provide their own mappings in a file called AutoMapperConfig.cs. The following code example shows part of the AutoMapper configuration for the Entity Framework repository classes in the DataAccess.Repo.Impl.Sql project. This code handles the mapping from an Entity Framework **InventoryProduct** object to a domain **InventoryProduct** entity object, and from an Entity Framework **Person** object to a domain **Person** object. Note that in this code sample, **DE** is an alias for the DataAccess.Domain namespace that contains the domain entity types:

```
private static void SetAutoMapperConfigurationPrivate()
{
    // define the mapping from the InventoryProduct entity
    // to the InventoryProduct domain entity
    Mapper.CreateMap<Order.InventoryProduct, DE.Order.InventoryProduct>();
    ...

    // define the mapping from the Person entity to the Person domain entity
    Mapper.CreateMap<Person.Person, DE.Person.Person>()
        .ForMember(dest => dest.Id,
                   src => src.MapFrom(val => val.BusinessEntityId))
        .ForMember(dest => dest.PasswordHash,
                   src => src.MapFrom(val => val.Password.PasswordHash))
        .ForMember(dest => dest.PasswordSalt,
                   src => src.MapFrom(val => val.Password.PasswordSalt));
    ...
}
```

*The **SetAutoMapperConfigurationPrivate** method is invoked by the public static **SetAutoMapperConfiguration** method that ensures that the configuration is performed only once.*

The **InventoryRepository** class calls the static **Map** method of the **Mapper** class to convert product information that it has retrieved from the SQL Server database into an **InventoryProduct** object, as shown in the following code example:

```
public class InventoryProductRepository : BaseRepository, IInventoryProductRepository
{
    public DE.InventoryProduct GetInventoryProduct(int productId)
    {
        using (var context = new InventoryProductContext())
        {
            ...
            var inventoryProduct = context.InventoryProducts.SingleOrDefault(
                p => p.ProductId == productId);
            ...

            var result = new DE.InventoryProduct();
            Mapper.Map(inventoryProduct, result);
            return result;
        }
    }
}
```

Similarly, the **PersonRepository** class uses calls the **Map** method after it has retrieved the customer information from the SQL Server database. The following code example highlights the statement in the **GetPerson** method that performs this task:

```
public class PersonRepository : BaseRepository, IPersonRepository
{
    public DE.Person GetPerson(Guid personGuid)
    {
        ...
        using (var context = new PersonContext())
        {
            person = context.Persons
                ...
                .SingleOrDefault(p => p.PersonGuid == personGuid);
            ...
            var result = new DE.Person();
            ...
            Mapper.Map(person, result);
            ...
            return result;
        }
    }
    ...
}
```

Where appropriate, the repository classes also use AutoMapper to convert from domain entity objects into database-specific types. For example, the AutoMapper configuration for the Entity Framework repository classes includes the following mapping that translates an **Order** domain entity object into a **SalesOrderHeader** object. Note that the **Order** class does not include a value for every property in the **SalesOrderHeader** class, and the mapping generates static values for these properties:

```
private static void SetAutoMapperConfigurationPrivate()
{
    ...
    // define the mapping from the Order domain entity
    // to the SalesOrderHeader entity
    Mapper.CreateMap<DE.Order.Order, Order.SalesOrderHeader>()
        .ForMember(dest => dest.OnlineOrderFlag,
                   src => src.MapFrom(val => true)) // static value
        .ForMember(dest => dest.RevisionNumber,
                   src => src.MapFrom(val => 3)) // static value
        .ForMember(dest => dest.SalesOrderDetails,
                   src => src.MapFrom(val => val.OrderItems))
        .ForMember(dest => dest.ShipMethodId,
                   src => src.MapFrom(val => 5)) // static value
        .ForMember(dest => dest.Status,
                   src => src.MapFrom(val => (byte)val.Status))
        .ForMember(dest => dest.TaxAmt,
                   src => src.MapFrom(val => 0)) // static value
        .ForMember(dest => dest.BillToAddressId,
                   src => src.MapFrom(val => val.BillToAddress.Id))
        .ForMember(dest => dest.ShipToAddressId,
                   src => src.MapFrom(val => val.ShippingAddress.Id))
        .ForMember(dest => dest.CreditCardId,
                   src => src.MapFrom(val => val.CreditCard.Id))
        .ForMember(dest => dest.BillToAddress, src => src.Ignore())
        .ForMember(dest => dest.CreditCard, src => src.Ignore());
    ...
}
```

This mapping is used when a **SalesOrderRepository** object saves an order to the SQL Server database. The following code example shows the **SaveOrder** method of the **SalesOrderRepository** class:

```
public class SalesOrderRepository : BaseRepository, ISalesOrderRepository
{
    public DE.Order SaveOrder(DE.Order order)
    {
        var salesOrderHeader = new SalesOrderHeader();
        Mapper.Map(order, salesOrderHeader);

        using (var context = new SalesOrderContext())
        {
            ...
            context.SalesOrderHeaders.Add(salesOrderHeader);
            context.SaveChanges();
        }

        return order;
    }
    ...
}
```

How the Windows Azure Table Service Repository Converts Data

The **ShoppingCartRepository** stores and retrieves shopping cart data from the Windows Azure Table service using the following types:

```
public sealed class ShoppingCartTableEntity: ...
{
    ...
    public string ShoppingCartItemsJSON { get; set; }
    public Guid TrackingId { get; set; }
}

public class ShoppingCartItemTableEntity
{
    public int Quantity { get; set; }
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public decimal ProductPrice { get; set; }
    public string CheckoutErrorMessage { get; set; }
}
```

The **ShoppingCartTableEntity** represents a shopping cart, while the **ShoppingCartItemTableEntity** contains a single item held in the shopping cart. The **ShoppingCartItems** property in the **ShoppingCartTableEntity** class is a string that contains one or more **ShoppingCartItemTableEntity** objects serialized as a JSON string. The section "How Adventure Works Used a Key/Value Store to Hold Shopping Cart Information" in Chapter 4 provides more information on why the data is stored in this format.

The **ShoppingCartRepository** class converts data between **ShoppingCartTableEntity** objects and **Shopping-Cart** domain objects by using a custom mapper class called **ShoppingCartMapper**. This class provides two static **Map** methods—one that converts a **ShoppingCartTableEntity** object into a **ShoppingCart** domain object, and another that performs the opposite operation. Chapter 4 describes how these methods work.

The **ShoppingCartRepository** class invokes the **Map** methods as it retrieves and stores shopping cart information. The code example below highlights the statements that convert from a **ShoppingCartTableEntity** object to a **ShoppingCart** domain object in the **GetShoppingCart** method of the **ShoppingCartRepository** class, and from a **ShoppingCart** domain object to a **ShoppingCartTableEntity** object in the **SaveShoppingCart** method:

```
public class ShoppingCartRepository : IShoppingCartRepository
{
    public ShoppingCart GetShoppingCart(string shoppingCartId)
    {
        var storedCart = new ShoppingCartContext().Get(shoppingCartId);
        return storedCart != null ? ShoppingCartMapper.Map(storedCart)
                                  : new ShoppingCart(shoppingCartId);
    }

    public ShoppingCart SaveShoppingCart(ShoppingCart shoppingCart)
    {
        new ShoppingCartContext().Save(ShoppingCartMapper.Map(shoppingCart));
        return shoppingCart;
    }
    ...
}
```

For more information about how the ***ShoppingCartRepository*** *is implemented, see the section "Storing and Retrieving Data in the Windows Azure Table Service" in Chapter 4.*

How the Neo4j Repository Converts Data

The product recommendations information stored in the Neo4j database is stored as serialized JSON data. The Neo4j **ProductRecommendationRepository** class retrieves data from the database and deserializes it as **ProductGraphEntity** objects. The following code example shows the **ProductGraphEntity** class:

```
public class ProductGraphEntity
{
    [JsonProperty("productId")]
    public int ProductId { get; set; }

    [JsonProperty("name")]
    public string Name { get; set; }

    [JsonProperty("percentage")]
    public decimal Percentage { get; set; }
}
```

The **GetProductRecommendations** method of the **ProductRecommendationRepository** class converts **Product-GraphEntity** objects into **RecommendedProduct** domain objects that have a similar structure except that the properties are not tagged as JSON-serializable. This is the type expected by the **GetRecommendations** method in the **ProductsController**. The following code example highlights how the **GetProductRecommendations** method manually converts a **ProductGraphEntity** object into a **RecommendedProduct** domain object:

```
public class ProductRecommendationRepository : IProductRecommendationRepository
{
    ...
    public ICollection<RecommendedProduct> GetProductRecommendations(
        int productId)
    {
        var recommendedProducts = new List<RecommendedProduct>();
        ...
        var graphResults = ... // fetch data from the graph database
                               // as a collection of ProductGraphEntity objects
        ...

        foreach (var graphProduct in graphResults)
        {
            recommendedProducts.Add(new RecommendedProduct()
            {
                Name = graphProduct.Name,
                Percentage = graphProduct.Percentage,
                ProductId = graphProduct.ProductId
            });
        }

        return recommendedProducts;
    }
    ...
}
```

*For more information about how the **ProductRecommendationRepository** works, see the section "Retrieving Product Recommendations from the Database" in Chapter 7.*

Instantiating Repository Objects

The repository classes implement a common set of interfaces that enabled the developers at Adventure Works to decouple the business logic in the controller classes from the database-specific code. However, a controller still has to instantiate the appropriate repository classes, and this step can introduce dependencies back into the code for the controller classes if it is not performed carefully.

To eliminate these dependencies, the developers chose to use the Unity Application Block to inject references to the repository classes dynamically rather than coding them statically. This approach enables the developers to more easily switch to a different set of repository classes that reference an alternative data access mechanism without requiring that they rebuild or even redeploy the entire application.

Each controller references the repositories that it uses through the repository interface. The following code example shows how the **AccountController** class references the **PersonRepository** type through the **IPerson-RepositoryInterface**:

```
public class AccountController : ApiController
{
    private IPersonRepository personRepository;
    ...

    public AccountController(IPersonRepository personRepository, ...)
    {
        this.personRepository = personRepository;
        ...
    }

    public HttpResponseMessage Get(string id)
    {
        Guid guid;
        if (!Guid.TryParse(id, out guid))
        {
            ...
        }

        var person = this.personRepository.GetPerson(guid);
        ...
    }
    ...
}
```

The constructor is passed a reference to a **PersonRepository** object that is used by the methods in the **AccountController** class. At runtime, the MvcWebApi web service uses the Unity Application block to re-solve the **IPersonRepository** reference passed to the **AccountController** constructor and create a new **PersonRepository** object. The web.config file for the DataAccess.MvcWebApi project contains the following configuration information:

```
<?xml version="1.0" encoding="utf-8"?>
...
<configuration>
  <configSections>
    ...
    <section name="unity"
type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
Microsoft.Practices.Unity.Configuration"/>
  </configSections>
  ...
  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    ...
    <alias alias="IPersonRepository"
type="DataAccess.Repo.Interface.IPersonRepository,
DataAccess.Repo.Interface" />
    ...
    <container>
      ...
      <register type="IPersonRepository" mapTo="DataAccess.Repo.Impl.Sql.
Person.PersonRepository,
DataAccess.Repo.Impl.Sql">
        ...
       </register>
      ...
    </container>
  </unity>
</configuration>
```

This configuration enables the Unity Application Block to resolve references to the **IPersonRepository** interface as an instance of the **PersonRespository** class in the DataAccess.Repo.Impl.Sql assembly. To switch to a different repository that uses a different data access technology, update the reference in the **<container>** section of the configuration file to reference the appropriate repository.

For more information about using the Unity Application Block to resolve dependencies, visit the "Unity" page on MSDN.

More Information

All links in this book are accessible from the book's online bibliography on MSDN at: *http://msdn.microsoft.com/en-us/library/dn320459.aspx*.

The page "Routing in ASP.NET Web API" is available on the ASP.NET website at *http://www.asp.net/web-api/overview/web-api-routing-and-actions/routing-in-aspnet-web-api*.

You can find details about AutoMapper on GitHub, at *https://github.com/AutoMapper*.

The Repository pattern is described in detail on the "*Patterns of Enterprise Application Architecture*" website at *http://martinfowler.com/eaaCatalog/repository.html*.

Information describing how to use the Fluent API in the Entity Framework 5.0 to map types to database tables is available on MSDN at *http://msdn.microsoft.com/data/jj591617*. Information describing how to configure relationships between entities by using the Fluent API is available at *http://msdn.microsoft.com/data/jj591620*.

You can find more information about using the Unit of Work pattern with the Repository pattern on MSDN at *http://blogs.msdn.com/b/adonet/archive/2009/06/16/using-repository-and-unit-of-work-patterns-with-entity-framework-4-0.aspx*.

You can download the Unity Application Block from the Microsoft Download Center at *http://www.microsoft.com/en-us/download/details.aspx?id=38788*.

Unity documentation is available on MSDN at *http://msdn.com/unity*.

Appendix B How the Developers Selected the NoSQL Databases

The developers at Adventure Works recognized that the different types of data required by the Shopping application would likely have different data storage and retrieval requirements. They could have implemented the system using a single relational database, but this approach would have entailed designing a collection of relational tables that might not be as well optimized for each business use case in the Shopping application as a non-relational approach would be. Therefore, the developers decided to adopt a strategy that utilized the most appropriate data storage mechanism for each form of data access performed by the application. The chapters in this guide describe how and why the developers at Adventure Works selected the various NoSQL databases that they used to store the information that the application uses, but the following list summarizes the decisions that the developers made:

- When a customer registers with the Shopping application, their details must be held in a safe and reliable database, and the login process that identifies the customer must be performed in a secure manner. The developers chose to implement the business functionality for the Shopping application as an ASP.NET web service built by using the MVC4 Web API. ASP.NET includes support for using SQL Server to manage and protect user account information, so the developers chose to store customer details in a SQL Server database.

  The Shopping application has to integrate with the warehousing and dispatching systems that handle inventory and order processing inside Adventure Works. These are transactional systems also based on SQL Server. They are outside the scope of the Shopping application, but the orders placed by customers have to be made available to them. Therefore, when a customer places an order, the details of the order are saved in the SQL Server database that these systems use.

  In the sample solution provided with the guide, the MvcWebApi web service writes the data directly to the SQL Server database used by the warehousing and dispatching systems. In the production environment described in Chapter 3 the details of orders are stored by using Windows Azure SQL Database in the cloud and Windows Azure Data Sync propagates this information to the SQL Server database running on-premises.

- When a customer logs in and adds items to their shopping cart, the application needs to retrieve and store the shopping cart in data storage. When the customer places an order, the shopping cart is no longer required and the application has to delete it from storage. The actual content of a shopping cart is immaterial to the data store, which can treat it as an opaque value. The important point is that the data store has to enable a shopping cart to be accessed quickly by using a unique key. For this reason, the developers chose to use a key/value store in the form of the Windows Azure Table service.

  You can also configure the user interface to retrieve product images from the Windows Azure Blob service. This is another example of a key/value store, although the MvcWebApi web service does not implement any specific functionality to support this mechanism; it is handled completely by the UI web application.

- When a customer browses products in the in the product catalog, the application has to quickly retrieve the data that describes category, subcategory, and product information. The application has to be able to filter this data by nonkey fields. For example, it has to fetch products by using the subcategory ID. The developers could have stored this information in a relational database, but the information held for each product could vary resulting in a complex schema, and retrieving data could result in the application having to perform a number of relational join operations across tables. For these reasons, the developers decided to structure the product and category information as a set of documents and store the information in a document database.

  For similar reasons, the developers chose to use a document database to store order history information. In this case, the application needs to be able to retrieve order history documents by using the **OrderCode** field. This field is not the key, and all order history documents that relate to the same order have the same value in this field.
- Although not part of the initial implementation, in the future the developers at Adventure Works may need to store information about website traffic to help assess how customers use the application and address possible performance issues. This data is likely to be high-growth and should support analytical processing to enable an administrator to determine the frequency with which customers visit each page in the Shopping application. These requirements are easily met by using a column-family database.
- When the application displays the details of a product, it also fetches production recommendations (other products that the customer may also be interested in). The application needs to retrieve this information based on potentially complex relationships between entities. Using a graph database helps to reduce the complexity of the queries that the application runs.

Index