



3 - Implementing a Relational Database

patterns & practices

proven practices for predictable results

On this page:	Download:
<p> Designing a Relational Database to Support Fast Transactions Normalizing Tables Implementing Efficient Transactions Implementing Distributed Transactions Designing a Relational Database to Optimize Queries Denormalizing Tables Maintaining Query-Only Data Designing Indexes to Support Efficient Queries Partitioning Data How Adventure Works Designed the Database for the Shopping Application Registering a New Customer and Logging In Placing an Order Verifying the Price of an Item Maintaining Data Integrity Implementing a Relational Database to Maximize Concurrency, Scalability, and Availability Scaling Out and Sharding Minimizing Network Latency Improving Availability Implementing a Relational Database in the Cloud by Using Microsoft Windows Azure Why Adventure Works Used Windows Azure SQL Database for the Shopping Application Accessing Data in a Relational Database from an Application Connecting to a Relational Database Abstracting the Database Structure from Application Code - <i>Using an Object-Relational Mapping Layer, Using the Entity Framework, Using a Micro-ORM</i> How the Shopping Application Accesses the SQL Server Database Retrieving Data from the SQL Server Database Inserting, Updating, and Deleting Data in the SQL Server Database Summary More Information </p>	<p> Download code samples </p> <p>  Download book (PDF) </p> <p>  Order paperback </p>

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

Note:

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 your 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 (**SalesOrderHeaderDueDate**). 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.

Note:

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.

Note:

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.

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

Normalized Tables**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

 Poe says:

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.

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.

 Poe says:

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.

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.

Note:

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 UNCOMMITTED** 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.

Poe says:



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.

- **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.

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.

Note:

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.

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.

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.

 **Jana says:**



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.

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 **SalesOrderHeaderTaxAmount** table for this order, so it is necessary to phrase the query by using a right outer join which generates a temporary *ghost* row:

```
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	08/07/2013
44280	08/10/2013
44544	08/11/2013



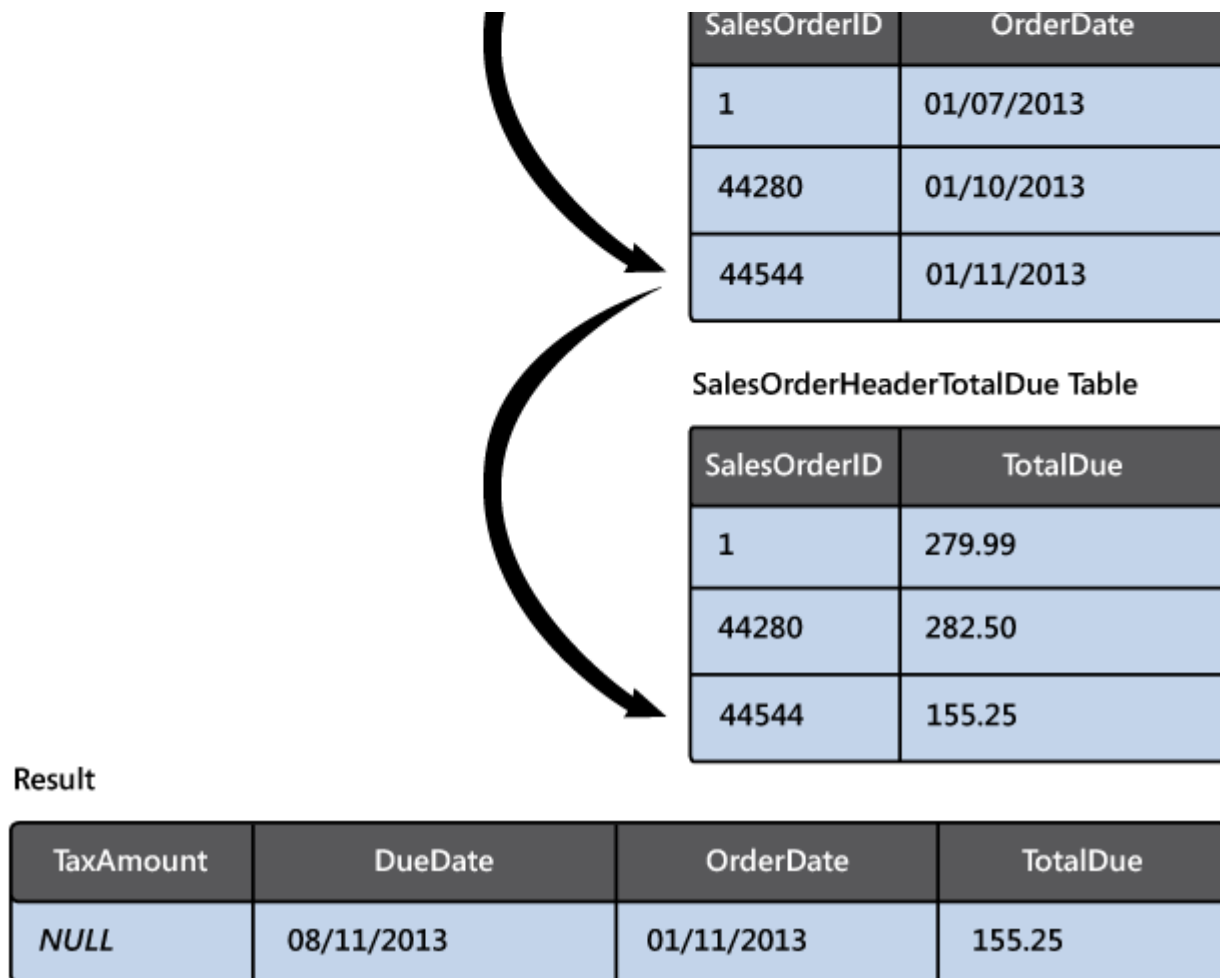


Figure 2 - Reconstructing the details of an order from the normalized tables holding order information
This section describes some techniques that you can follow to optimize your databases for query access.

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):

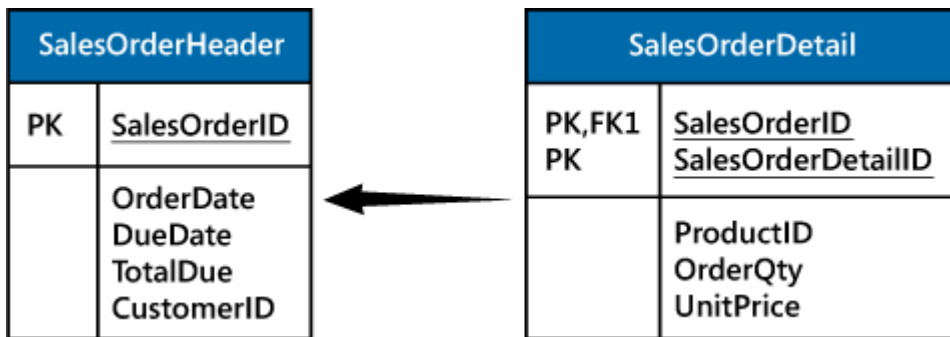


Figure 3 - The structure of the SalesOrderHeader and SalesOrderDetail tables
There are at least two ways that enable you to eliminate this join:

- 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.

SalesOrderDetail	
PK	<u>SalesOrderID</u>
PK	<u>SalesOrderDetailID</u>
	ProductID OrderQty UnitPrice OrderDate DueDate TotalDue CustomerID

Example 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

Duplicated Sales Order Header Data

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.

SalesOrderHeader	
PK	<u>SalesOrderID</u>
	OrderDate DueDate TotalDue CustomerID ProductID_1 OrderQty_1 UnitPrice_1 ProductID_2 OrderQty_2 UnitPrice_2 ProductID_3 OrderQty_3 UnitPrice_3 ProductID_4 OrderQty_4 UnitPrice_4

Example Data

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.

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.

Poe says:



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.

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
```

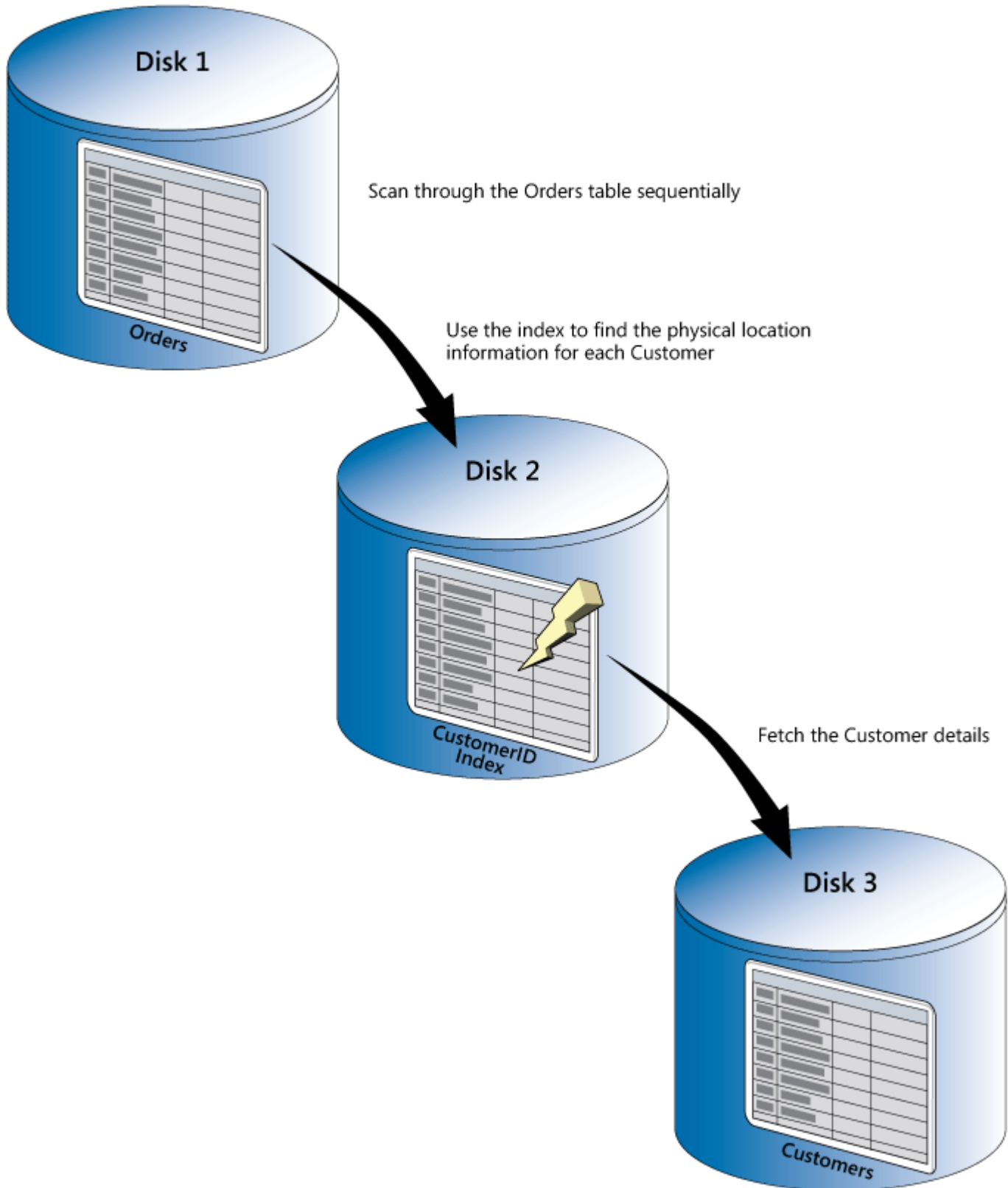


Figure 6 - Maximizing parallel I/O by placing tables and indexes on separate physical disks

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.

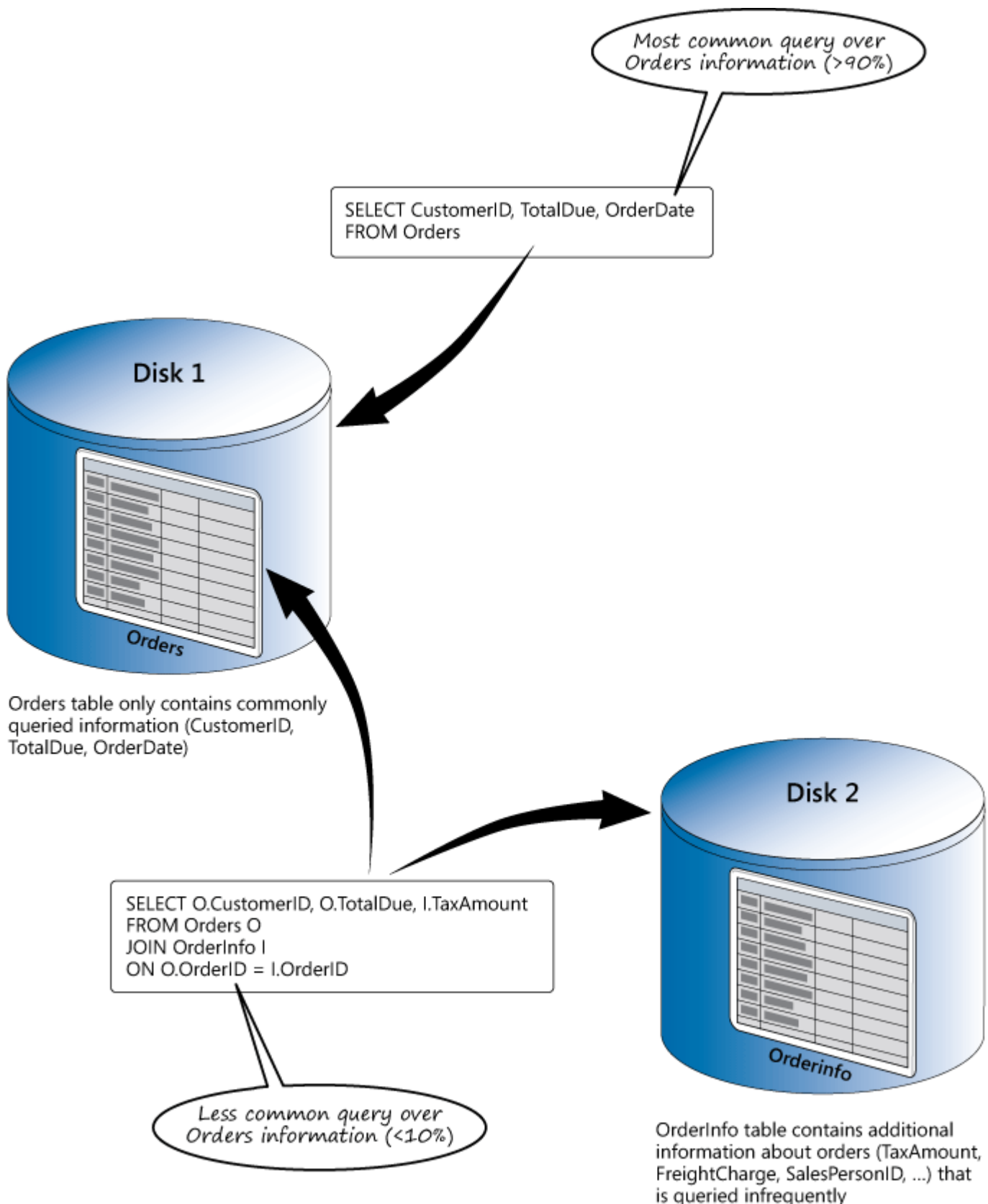


Figure 7 - Partially normalizing a table and implementing vertical partitioning

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.

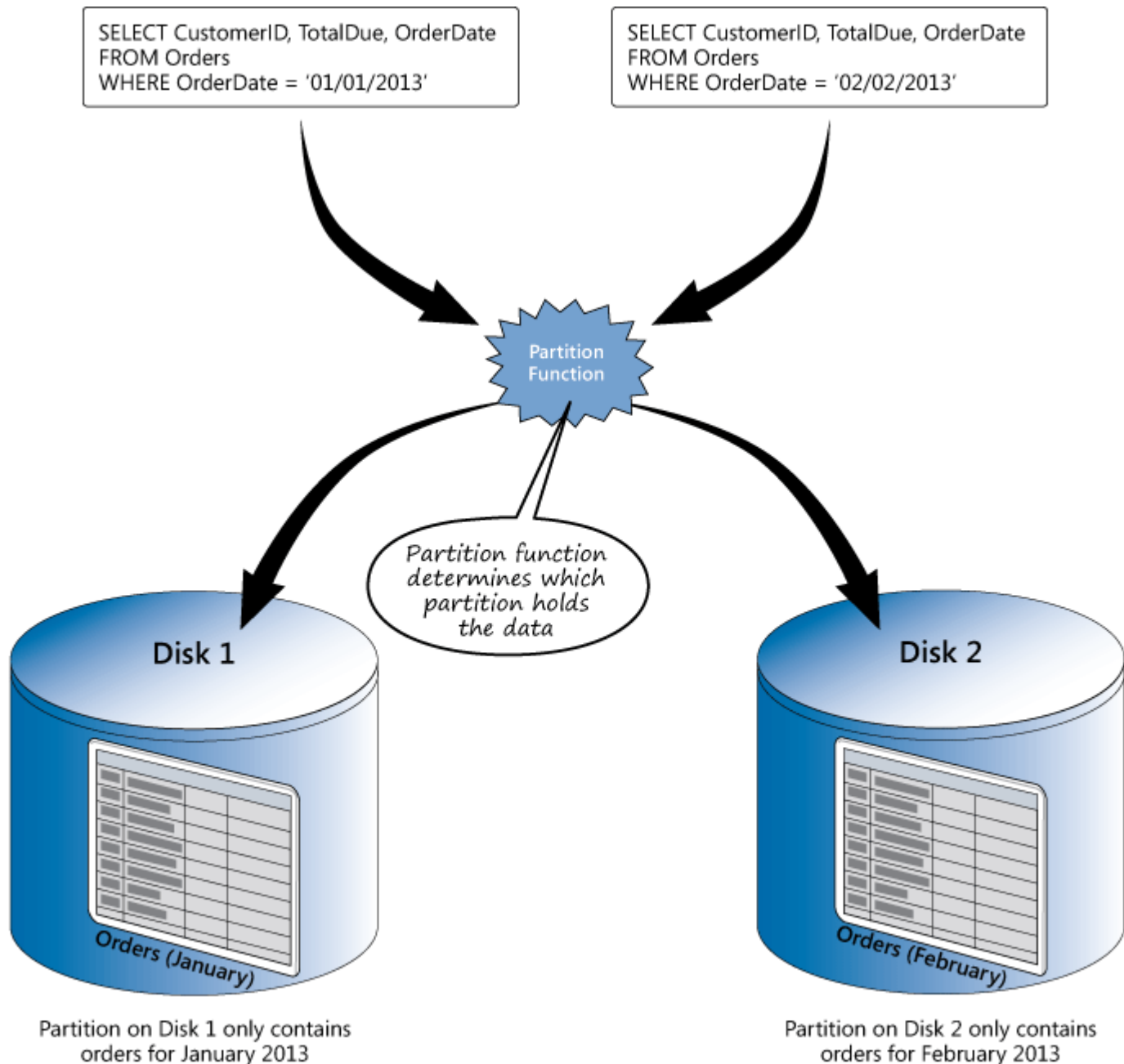


Figure 8 - Implementing horizontal partitioning

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.

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.

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.

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:

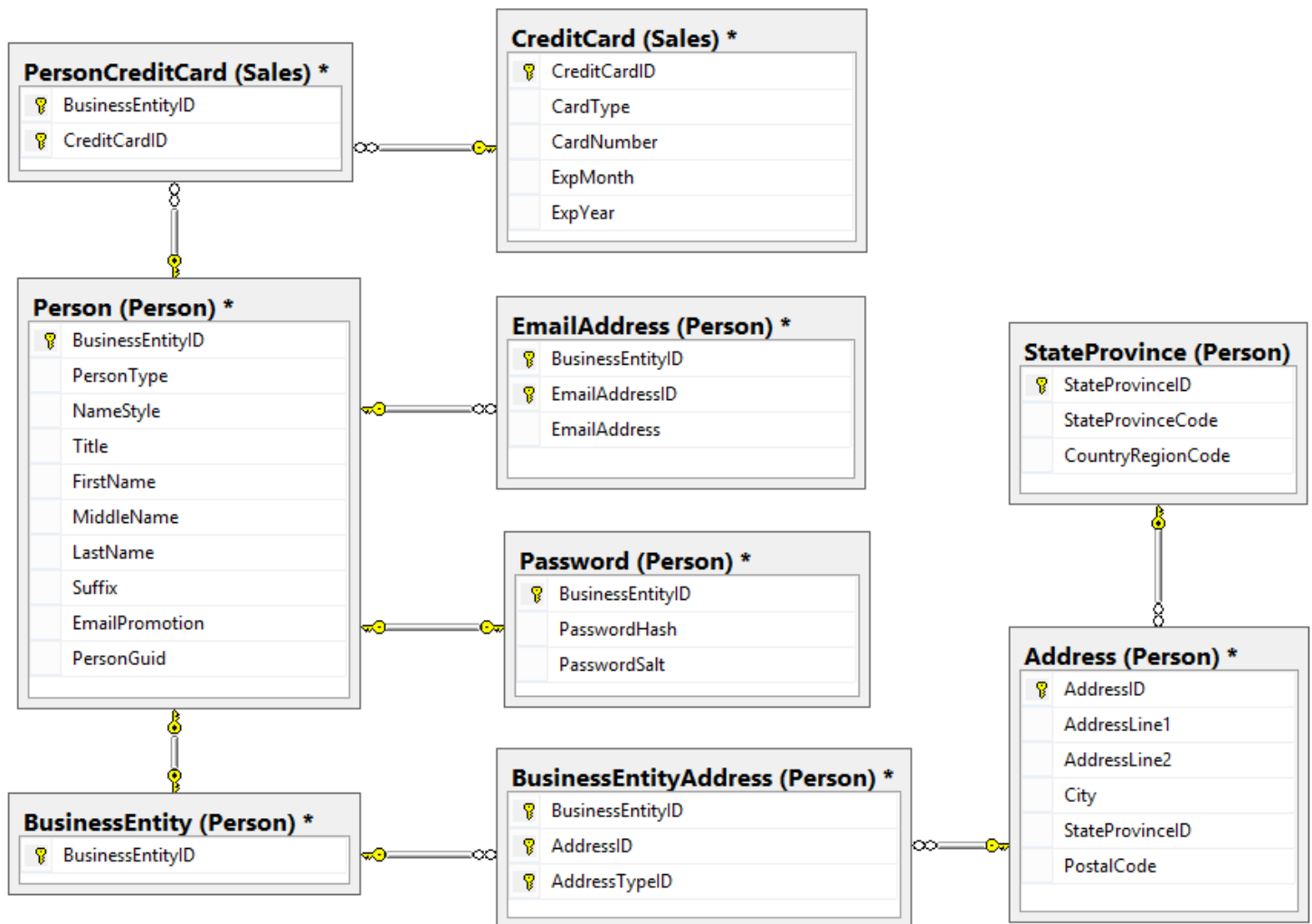


Figure 9 - Tables used by the Shopping application to store customer information

Note:

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:

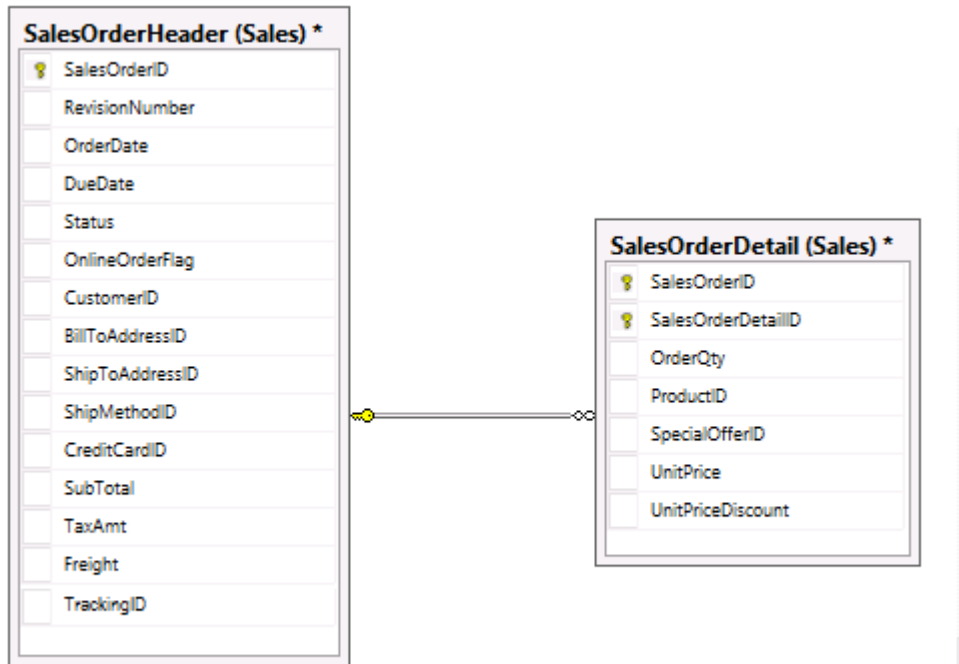


Figure 10 - Tables used by the Shopping application to record the details of an order

Note:

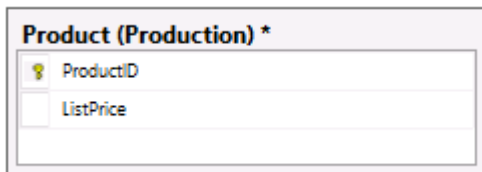
This diagram only shows the columns in these tables that are relevant to the Shopping application. The warehousing and dispatch systems also use these tables and maintain additional information about orders (such as whether the items have been picked, the order has been dispatched, and so on) which are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

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.

Verifying the Price of an Item

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.



Product (Production) *	
ProductID	
ListPrice	

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 **BusinessEntity** 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 **PersonCreditCard** 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 **SalesOrderHeader** 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.

Note:

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.

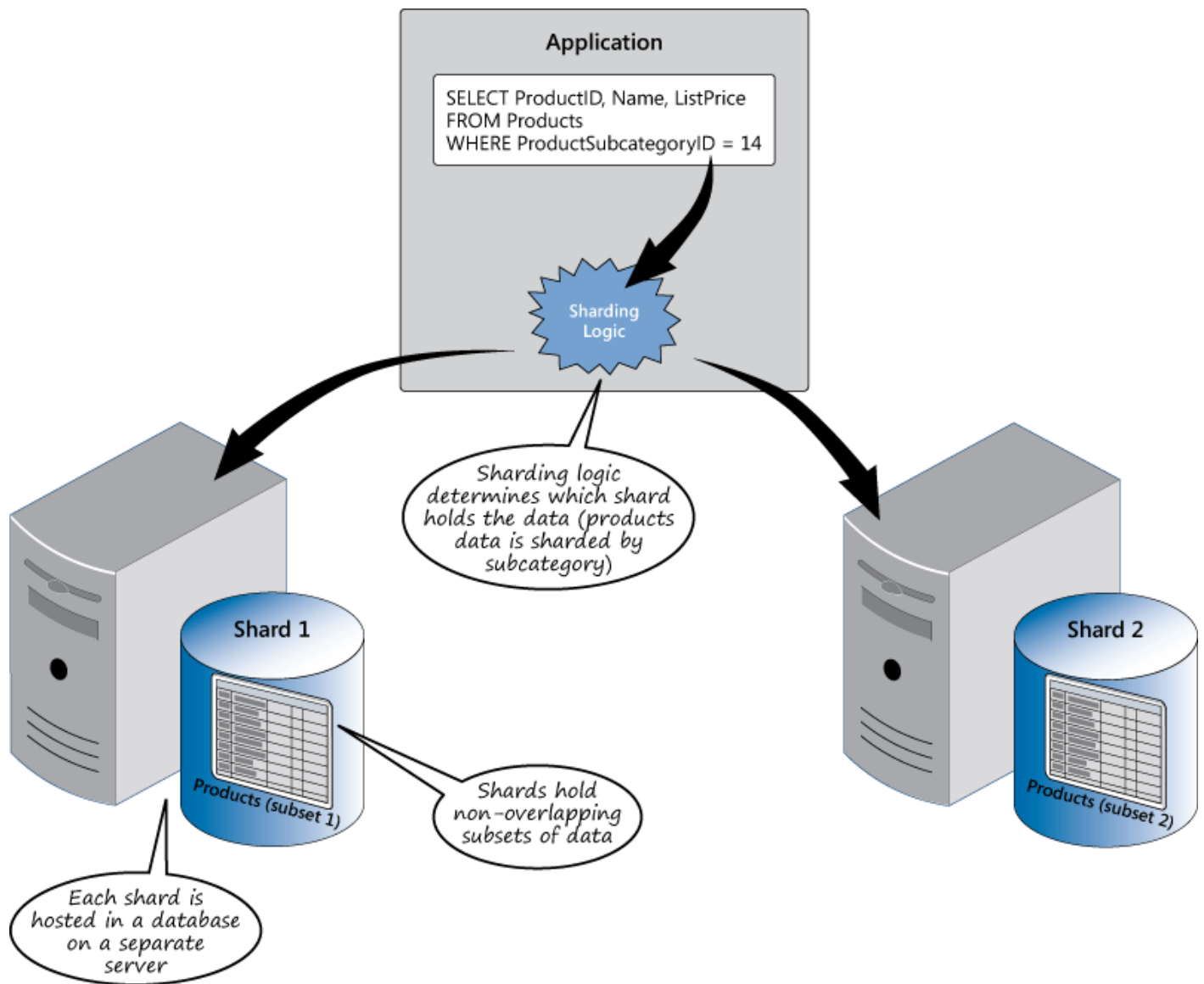


Figure 12 - Scaling out by implementing the Shared Nothing approach

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.

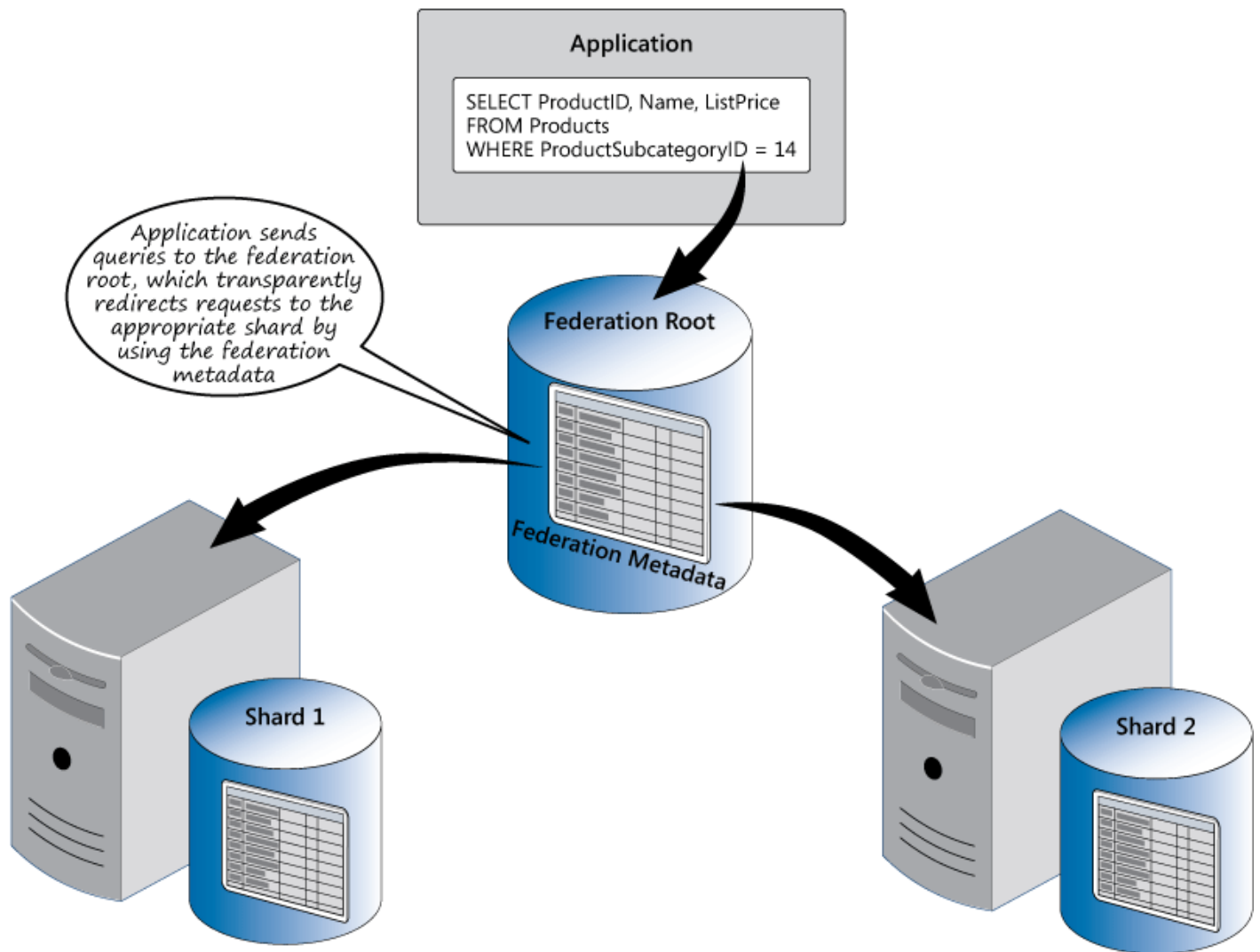


Figure 13 - Scaling out by implementing the Federation pattern

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.

Note:

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.

 **Note:**

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.

 **Poe says:**



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 "<http://msdn.microsoft.com/library/hh871440.aspx> 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 it can connect to the database, or you might need to reconfigure the application manually to reference a different node.

 **Poe says:**



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.

- **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](#)."

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.

 **Bharath says:**

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.

 **Note:**

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.

 **Bharath says:**

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.

 **Note:**

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. 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 logins that you can use (you cannot log in as **sa**, **guest**, or **administrator**, for example).

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.

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.

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.

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

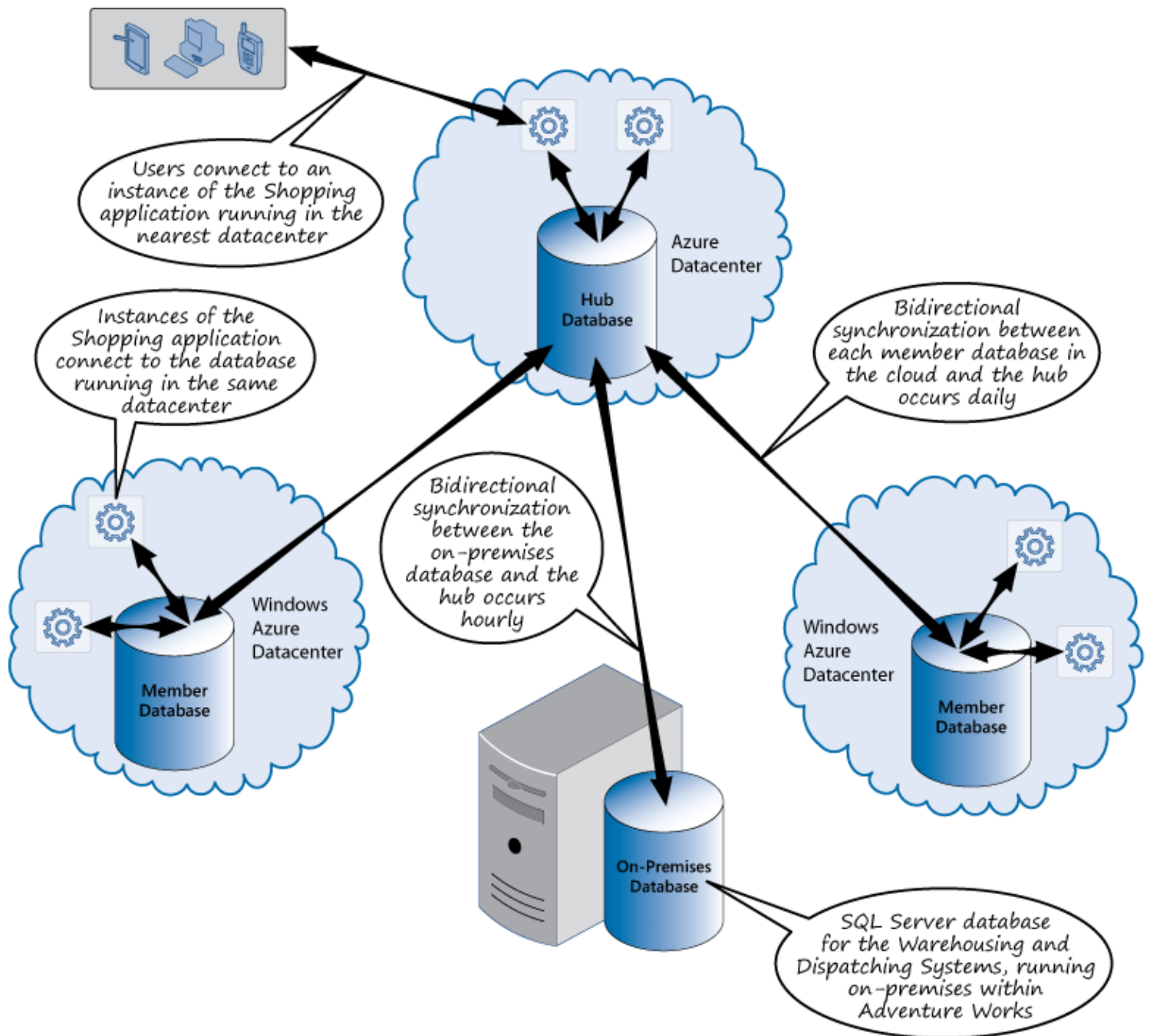


Figure 14 - How Adventure Works deployed the databases and synchronize the data

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.

Note:

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.

Markus says:

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 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:

The screenshot shows the Entity Model Designer in Visual Studio 2012. The top pane displays the Entity Model Designer interface with two entities: **SalesOrderHeader** and **SalesOrderDetail**. A 1-to-many relationship is shown between them, with a cardinality of 1 on the SalesOrderHeader side and * on the SalesOrderDetail side. The SalesOrderHeader entity has properties: SalesOrderID, RevisionNumber, OrderDate, DueDate, ShipDate, Status, OnlineOrderFlag, SalesOrderNum..., PurchaseOrderN..., AccountNumber, CustomerID, SalesPersonID, TerritoryID, BillToAddressID, ShipToAddressID, ShipMethodID, CreditCardID, CreditCardAppr..., CurrencyRateID, SubTotal, TaxAmt, Freight, TotalDue, and Comment. The SalesOrderDetail entity has properties: SalesOrderID, SalesOrderDetail..., CarrierTracking..., OrderQty, ProductID, SpecialOfferID, UnitPrice, UnitPriceDiscount, LineTotal, rowguid, and ModifiedDate. The SalesOrderHeader entity has a navigation property named SalesOrderHeader.

The bottom pane shows the **Mapping Details - SalesOrderHeader** window. It displays a table with columns: **Column**, **Operator**, and **Value / Property**.

Column	Operator	Value / Property
Tables		
Maps to SalesOrderHeader		
<Add a Condition>		
Column Mappings		
SalesOrderID : int	↔	SalesOrderID : Int32
RevisionNumber : tinyint	↔	RevisionNumber : Byte
OrderDate : datetime	↔	OrderDate : DateTime
DueDate : datetime	↔	DueDate : DateTime
ShipDate : datetime	↔	ShipDate : DateTime
Status : tinyint	↔	Status : Byte
OnlineOrderFlag : bit	↔	OnlineOrderFlag : Boolean

The bottom of the window shows tabs for **Error List**, **Output**, **Find Results 1**, **Find Symbol Results**, and **Mapping Details**.

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.

Note:

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.

Markus says:



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.

XML

```
<connectionStrings>
```

```
...
```



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

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.

For detailed information about the Entity Framework, see "[Entity Framework](#)" page in the Data Developer Center, on MSDN.

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.

Note:

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:

C#

```
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
    }
    ...
}
```

 Markus says:



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.

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.

 Markus says:

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.



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.

Note:

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

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 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:

C#

```
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;
    }
}
...
}

```

Note:

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:

C#

```
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 **InventoryAndPriceCheck** method in the **InventoryService** class to verify the price of items in the order and also check whether Adventure Works still stocks these items.

Note:

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 **InventoryAndPriceCheck** method:

C#

```
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);

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

Note:

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.

Markus says:

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:

C#

```

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:

C#

```

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 **OrdersController** 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):

C#

```
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 high-volume 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 on MSDN 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.
- The "[DML Triggers](#)" page describing how to implement database triggers in SQL Server 2012 is available on MSDN.
- The "[Stored Procedures \(Database Engine\)](#)" page describing the advantages of using stored procedures in SQL Server 2012 is available on MSDN.
- The page "[Online Transaction Processing vs. Decision Support](#)", available on MSDN, summarizes common design decisions for meeting the requirements of databases that support OLTP and DSS operations.
- You can download the guide "[Building Hybrid Applications in the Cloud on Windows Azure](#)" from MSDN.
- The "[SQL Server Integration Services](#)" section in Microsoft SQL Server 2012 Books Online, is available on MSDN.
- You can find detailed information on SQL Server indexes in Books Online for SQL Server 2012. The "[Indexes](#)" section is available on MSDN.
- The "[Partitioned Tables and Indexes](#)" section of Books Online for SQL Server 2012 is available on MSDN.
- 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.
- 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 web site.
- The section "[AlwaysOn Failover Cluster Instances \(SQL Server\)](#)" in Books Online for SQL Server 2012.
- The [Windows Azure calculator](#) is available online.
- The page "[Federations in Windows Azure SQL Database](#)," describing how to use federations to scale out a database, is available on MSDN.
- You can find the "[General Guidelines and Limitations \(Windows Azure SQL Database\)](#)" page on MSDN.
- The article "[Data Series: SQL Server in Windows Azure Virtual Machine vs. SQL Database](#)" is available on the MSDN blogs.
- The article "[Choosing between SQL Server in Windows Azure VM & Windows Azure SQL Database](#)" is available on the MSDN blogs.
- The "[SQL Data Sync](#)" page, which provides information about Windows Azure SQL Data Sync, is available on MSDN.
- Information about the [Microsoft Enterprise Library](#) is available on MSDN at <http://msdn.com/entlib>.
- You can read about LINQ to SQL on the "[LINQ to SQL: .NET Language-Integrated Query for Relational Data](#)" page on MSDN.
- You can find information about the [Entity Framework](#) in the Data Developer Center, available on MSDN.
- You can download the [Unity Application Block](#) from MSDN at <http://msdn.com/unity>.
- The [Repository pattern](#) is described on MSDN.
- Information about the Dapper dot net package is available on NuGet <http://www.nuget.org/packages/Dapper/>.

[Next Topic](#) | [Previous Topic](#) | [Home](#) | [Community](#)

© 2018 Microsoft