

Non-Relational Databases

Chapter Outline

Introduction, 589	Additional Reading, 621
Two-Minute Chapter, 591	
Non-Relational Databases: Background, 592	
<i>Key-Value Pairs, 594</i>	
<i>Sparse Data and Flexible Columns, 595</i>	
<i>Distributed Data, 597</i>	
<i>Consistency and Integrity, 598</i>	
<i>Optimizing Data Storage for Queries, 600</i>	
Cassandra, 601	
<i>Installation Issues, 601</i>	
<i>Pet Store Web Example, 602</i>	
<i>Database Design, 603</i>	
<i>Primary Keys, 605</i>	
<i>Initial Queries, 607</i>	
<i>Indexes, 609</i>	
<i>Querying Tables with Compound Keys, 612</i>	
<i>INSERT and UPDATE, 613</i>	
Cloud Databases, 614	
Summary, 616	
Key Terms, 618	
Review Questions, 618	
Exercises, 619	
Web Site References, 621	

What You Will Learn in This Chapter

- Why would anyone need a non-relational database?
- What are the main features of non-relational databases?
- How are databases designed and queried using Cassandra?
- How does cloud computing benefit key-value pair databases?

A Developer's View

Ariel: Why the puzzled look, Miranda?

Miranda: Well, my company built this great Web site that lets customers post comments, rate products, and interact with each other...

Ariel: Yes, that sounds fairly standard today.

Miranda: But, there are millions of customers! We ran some tests with sample data and it runs really slowly. Plus, it looks like we would need a huge server, and the DBMS license fees will be enormous—just to provide a free service to customers.

Ariel: Wow! That does sound like a problem. Are there any other options.

Miranda: Yes, that is the confusing part. Big companies like Facebook have developed non-relational DBMSs that emphasize scalability and speed. Some are even open-source so we don't have to pay license fees for each copy. All of which is great, but these things are really new and they keep changing, so it is difficult to figure out how to structure the data and write the code.

Ariel: Hmm. That does sound tricky. But it sounds like it is useful to learn the basics to help you decide when to use the tools.

Getting Started

Web applications tend to handle data in a relatively unique pattern: Most data is exchanged as key-value pairs. Starting with data on forms passed to a Web server, the data is coded with a key (such as the textbox name) and the corresponding value. Large Web sites also struggle with handling data for millions of users. So new systems have been defined to store and retrieve these individual pieces of data as quickly as possible. The Cassandra project is one of the most popular. It focuses on the ability to store data across multiple servers; both for performance and to minimize disruptions if one node fails. The data design for these systems is not normalized, and Joins are not supported. This chapter presents the basic elements of design and data retrieval in non-normal databases.

Introduction

Why would anyone need a non-relational database? The importance of the Web has three major impacts on technology systems: (1) The need to run 24-7 (24-hours a day, 7-days a week); (2) The ability to handle relatively complex object data (pictures, long text, and so on); and (3) Handling data for millions of users. In terms of Web 2.0 (Web services and social interaction), the other key aspect is that companies provide these services with minimal or no fees on customers. The large scale of the applications causes several problems with performance, reliability, and cost. This last element has led large Web firms to develop open source tools to handle many computing aspects in an attempt to hold down licensing costs. Along the way, they decided to use a different data model to im-

prove performance and reliability. These newer systems do not use relational data storage. The early versions had minimal support for data storage and retrieval, so some people referred to them as **NoSQL** databases. But it is better to refer to them as **non-relational**, because the major differences lie in the data model and storage, not in the query language. Of course, as Chapter 1 points out, non-relational systems have been around longer than relational, so how are these new systems different from the old tools? The answer to that question revolves around two key features: distributed databases and key-value data storage.

The new DBMSs are evolving rapidly. Most of the tools were originally developed by specific companies for their particular needs (such as BigQuery (BigTable) and MapReduce by Google, Hadoop (HBase) by Yahoo, and DynamoDB by Amazon). The open-source community then developed variations on the tools. Some similarities exist across the tools, but in the end they are all different. As of 2013, some Web sites list 150 variations of NoSQL tools. To illustrate the concepts, this chapter explains the general concepts but focuses on one DBMS: Cassandra. Cassandra is in the top 3 list for non-relational DBMSs in terms of popularity or usage. It is also one of the best performing and has ongoing development with installation files for several operating systems (including Amazon EC2). It also has an interactive query language which makes it possible to explore the data without detailed programming. However, any real-world application would require programming, which is also supported through several languages.

Chapter 11 introduces how distributed databases can improve reliability and scalability. Data storage in the non-relational systems was built from the ground up to run as distributed systems. In particular, Cassandra operates as **peer-to-peer** distributed nodes. The system is designed to be installed on multiple servers, in multiple clusters, and across multiple data centers—which could be located anywhere in the world. Any piece of data is replicated across multiple servers, and typically each server holds only a portion of the data. If any node fails, the data is available from other nodes. As the database grows, more server nodes can be added to the clusters to scale the operations linearly.

The other defining aspect to the new systems is that data is stored as **key-value** pairs. The key can be any type of data, but must be unique. The data value can be any data element and might be a complex object or collection of items. The key-value concept is important in Web-based applications. For example, each text box on a Web-browser form has a unique ID. When the user submits the form to the server, the browser packages the data and sends it as pairs of the form: ID=value. The browser and Web server programming tools are designed to handle these pairs of data. So, it made sense to build a DBMS that uses this same concept to store and retrieve the data from files. The key-value concept is similar to the primary key in a relational DBMS, but most of the new systems are far more flexible. In particular, the new systems routinely violate the definition of first normal form (storing atomic, single-valued data in one cell). The data stored can be a single item (such as a last name), but it can also contain repeating items such as multiple e-mail addresses.

Retrieving data is quite different from the SQL approach. The most important limitation is that the query systems do not support any type of table JOIN—which is the main reason for the NoSQL name. Data can be retrieved from only one table at a time—by providing the key data. Some additional queries can be supported by predefining indexes on the desired search columns. But queries are limited to improve performance.

This chapter explores these fundamental differences between Cassandra and traditional systems. It explains how the database model design is different and how to handle common situations. A small example of the Pet Store is used to illustrate data storage for a simple Web application. You can download and install a copy of Cassandra from the Web (www.DataStax.com is recommended), and then download and install the sample Pet Store Web database to test the basic concepts.

Two-Minute Chapter

Data storage is the defining difference between traditional relational DBMSs and the new non-relational systems. To improve performance, the new systems require that indexes be defined for every item that needs to be queried later. So the data storage model must be defined in terms of the queries that will be used. The strongest limitation is that the query systems do not support any type of JOIN statement. Data is retrieved from one table at a time. Retrieving matching data from a second table requires writing code to extract a key for the second table and then writing another query to obtain the data from the second table. For even faster performance, data is often duplicated. For example, to avoid a lookup for Customer Name, many designers would store the Customer Name column along with the transaction data so it can be retrieved immediately. The assumption is that disk space is cheap, but Web response delays are expensive because people will leave a slow site.

The DBMSs do not provide referential integrity, so data entered in one table might not exist or match in a second table (unless application code is written to maintain integrity). Similarly, there is no guarantee that each node in the system has the exact same data for each item. A node or connection might fail or updates might be slow so a node might have older data. The goal of the new systems is to emphasize performance over strict data integrity. In their defense, an argument has been put forward that it is probably impossible to guarantee data integrity in a distributed system—without severe performance issues. Interestingly, Cassandra provides the ability to specify the level of consistency desired with each query.

The issue of handling one-to-many relationships is still important, and in many cases it makes sense to create separate tables to handle them. But the new systems often encourage storing multi-valued items in a column or cell (which violates the first normalization rule). Cassandra supports the definition of sets and lists within a single column. Again, the point is that anything stored with the original row key will be retrieved immediately. So any data that is used together should be stored together.

In Cassandra, a keyspace is similar to a schema in that it holds all of the tables for a single application. A Table holds a collection of rows, and each table must have a primary key—preferably based on a single column. One-to-many relationships are handled by defining two columns in a compound primary key. A non-key column can hold single-valued data, or sets, lists, and maps can be used to store repeating data.

Database design is more flexible than the relational model, with few strict rules. Start by normalizing the data into tables and then decide how to duplicate or combine data to improve performance. Ultimately, the design is based on the queries that will be needed by the application.

Cassandra uses the CQL query language to define the database structure and retrieve data. Additional programming (CLI) tools are available but CQL provides

commands that are similar to SQL—without the JOINS and with severe restrictions on retrieving data. The CREATE TABLE command is similar to SQL (with different options). The SELECT command is used to specify columns and the table name. A WHERE condition can be used but it can only include columns that are in the primary key or supported by a secondary index.

Non-Relational Databases: Background

What are the main features of non-relational databases? Technically, a non-relational database could be any data storage method that does not support the data normalization rules. In fact, the earliest data storage methods were flat files and hierarchical databases, which were then expanded into network databases (which had nothing to do with LANs or distributed databases). In some ways, the resurgence of non-relational systems is a continuation of the arguments for those earlier data storage methods. A primary argument was that data stored in non-relational systems could be retrieved faster. And in some cases, that answer was true. What Codd successfully argued is that the relational system separated the data so that it was stored more efficiently and provided support for **ad hoc queries**. The relational model also provides tools to ensure data integrity and consistency. So overall, it provides the best performance across a wide range of uses.

However, the relational approach is not necessarily the absolute fastest way to store and retrieve individual pieces of data. In particular, if data is always stored and retrieved in a specific way, it can be considerably faster to optimize the data storage to match the application needs. For instance, Chapter 12 explains the hashed or direct key access method. Given a unique key, its value can be hashed or converted into a specific location address and the data associated with that key can be stored and retrieved almost instantly. But, the application has to always store and retrieve the data using the key. The current non-relational systems utilize this hashed-key approach to store and retrieve data—with a few additional twists.

In a Web application environment data is often collected and transferred in key-value pairs, so it makes sense to store data using the key. Data related to individuals or to specific items is also easy to identify with unique keys. Data storage and retrieval can be optimized for these transactions. The data storage will fail if a manager wants to do a more complex search; but data could be extracted and stored in a data warehouse for purposes of data mining or complex searches.

The problem with Web applications is that users expect instantaneous results. In the early years, Google asked users if they would prefer 10 results or 30. Most people opted for the larger number. Until Google ran actual tests and found that people were dissatisfied with larger results page—as much as a 20 percent drop in usage. The reason: it took a half-second less time to generate the smaller page. Marissa Mayer (then at Google) gave a couple of talks with summaries on the Web such as <http://www.zdnet.com/blog/btl/googles-marissa-mayer-speed-wins/3925>. The point is that timing can be critical on Web pages, and as the number of users and amount of data increase, delay times can increase exponentially. It is far better to scale up the servers linearly as the number of users increases.

To improve speed, non-relational systems emphasize hashed-key data access, and storing data on distributed servers. Multiple servers are important for scale—adding new servers should improve the overall performance of the data storage. The Web also has geographic implications because of the location of users and bandwidth constraints. Distributed systems are useful because the data can be placed around the world where local sites can respond faster to user requests.

CID	LastName	FirstName	Email
101	Brown	Bobby	BBrown@gmail.com
102	Jones	Jackie	JJackie@live.com
103	Piste	Paula	SkiFast@yahoo.com

Relational table: Primary key (with index).

Atomic cell data, JOINS to other tables.

Fixed columns, all columns searchable.

Key	Value
91e83b31...	LN=Brown, FN=Bobby, E=BBrown@gmail.com
4f763ab4...	LN=Jones, FN=Jackie, E=JJackie@live.com
754d4a...	LN=Piste, FN=Paula, E=SkiFast@yahoo.com

Key-value pairs. Row key is unique and defines storage partition.

Row key is the default way to retrieve a row.

Searching by other columns requires a secondary index.

Data value can be almost anything.

Columns are treated as more key-value pairs and are flexible by row.

Figure 13.1

Relational v. Key-value pair table. Both use indexed primary keys to locate a row, but columns in a relational table are fixed and hold single, atomic values. With key values, rows are retrieved with a row key and the value can be almost anything; including more key-value pairs that are essentially columns.

The challenge with distributed systems lies with maintaining data integrity. Specifically, how can the DBMS ensure that all nodes in the system have the same up-to-date copy of data? Relational systems emphasize the importance of data integrity. Most use locking mechanisms and transaction logs to ensure that data is always accurate across the entire system. But these mechanisms add delays to processing data—particularly storing or updating values. And, in the end, it is still hard to guarantee that every node will always maintain consistent data—particularly if network connections fail.

So, the simple difference with non-relational systems is that they focus on performance and worry less about data consistency. To improve performance, they also limit the ways in which the data can be queried. Consequently, the database design ultimately must be based on identifying exactly how the data will be used and queried. Figure 13.1 shows the main conceptual differences between a table in a relational database and in a key-value pair database. Relational tables have fixed columns with atomic, single-valued cells. Data for rows is retrieved via an indexed primary key, but keys can use several columns. For key-value tables, a primary key is almost always a single column and data is primarily retrieved only by a specific key. The rest of the row value can hold almost anything. In table terms, the columns are treated as another set of key-value pairs. So new columns can be added to any row at any time, simply by adding new key-value pairs to the row data. But, searching for data by any column other than the key requires creating a new index on that column.

Some non-relational systems also provide more flexibility in defining tables—particularly columns. Primary keys identify rows of data, but with some tools it is possible to put anything into a row. Which means that each row might hold different columns and even different types of data. Early proponents of the non-relational approach argued that this flexibility made it easy to expand the database to add new columns and new data later. In a world of Web applications that start small and then add features with each new version, there is some appeal to this flexibility. On the other hand, putting different columns and data into every row is a programmer’s nightmare because the code has to continually check to see what data exists for each row. In most cases, it is safer to simply add new columns to tables and ensure that each row is at least somewhat consistent.

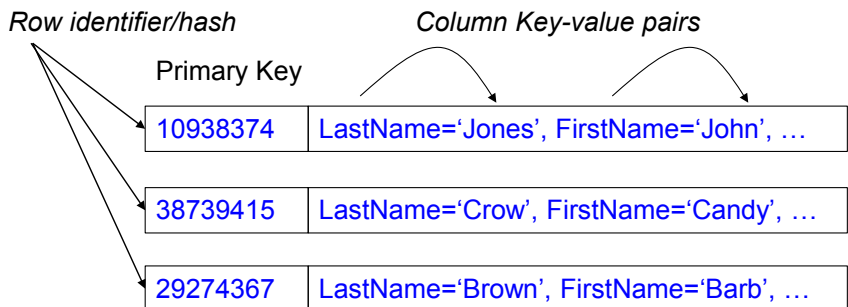
Key-Value Pairs

A key-value pair is probably the simplest data structure available for storing and locating data. Each item to be stored is identified with a key and the key is the only data needed to find the value item. The key could be any data type. A long integer is probably best to ensure that the values are unique, but text values can be converted into numbers through a hashing algorithm. As noted, Web forms make heavy use of key-value pairs, so most Web servers and other tools also use them. For instance, a basic customer Web form would send pairs of data to the server based on the text boxes, of the form: LastName=’Jones’, Email=’John@Jones.com’, and Category=’Student’.

For storing data in a database, the concept is similar; but a database table has rows and columns. So each row needs to have a unique key entry. In the case of Customer data, it would make sense to invent a CustomerID to use as the **primary key**. So a data row for a specific customer might be stored with a value for CustomerID of 10938374. The remaining “columns” of associated data (FirstName, LastName, Email, and so on) would be stored in the space identified by the primary key value. Figure 13.2 gives an example of the key-value concepts. The primary key is similar to most other lookups, where the key is converted into a storage location. Some tools use indexes, others might use a hashed-key conversion directly to the physical address.

Figure 13.2

Key-value pairs for identifying rows and for extracting column data within a row. The row primary key is just a direct/hashed-key lookup. The column storage shows how key mappings are used to support flexible rows that can hold different data.



The interesting twist with non-relational systems is that they might also store the column or cell data using key values where the key is the name of the column. Instead of allocating space for each column, the data storage consists of a mapping array that retrieves a value based on the key. With the sample data, data might be retrieved by specifying `CustomerID=10938374`. Then the application requests the value on that row associated with the key of 'LastName' or 'FirstName'.

Obviously, the row keys need to hold unique values. The column names also have to be unique, but in most cases those are predefined. The challenge with row keys becomes more difficult in the distributed environment of most non-relational systems. Think about the challenge of inventing a key number that has to be unique across all of the servers. A relational DBMS probably has a key-generation method that creates incremental values. It might store the latest ID values in a table and then generate the next value on demand. However, this approach requires that all servers have access to the same consistent table data. The non-relational approach avoids enforcing consistency, so a different method is needed to create ID values. The most common approach is to use a number known as a **universally unique identifier (uuid)**. These numbers have been used for several years for similar purposes, so software exists to generate them reliably on almost any device. Microsoft has used a variation known as globally unique identifier or GUID, but an ISO standard now exists. By the standard, a uuid is a 128-bit number represented by 32 hexadecimal digits and written in standard form with hyphens, such as:

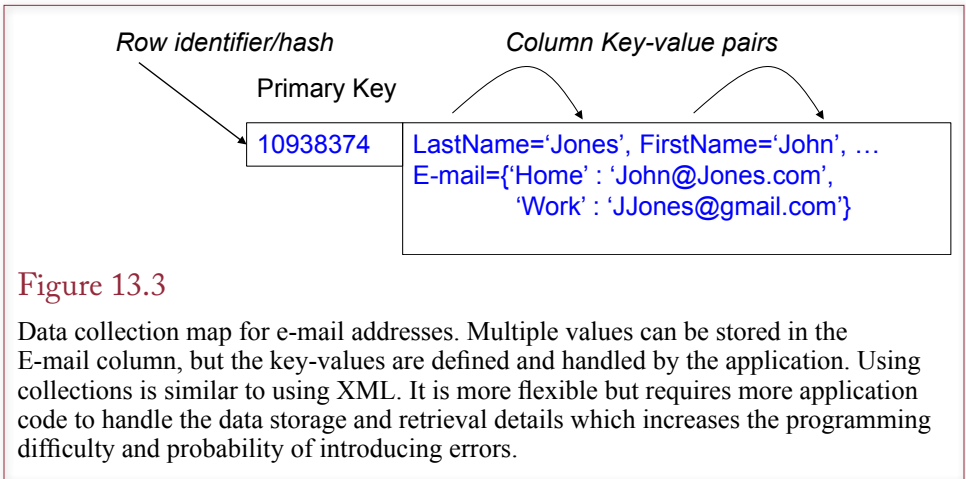
```
71c1da88-88af-4217-aa41-332ea3d33ae9
```

Several methods have been defined to generate uuid values. The earliest ones (type: Version 1) used the MAC address of the computer's network card and a measure of time. Because each network card is assigned a unique MAC address by the vendor, the UUID generated is known to be different from one generated by any other computer. Other methods also exist, including purely random numbers (Version 4), which could result in duplicates with a tiny probability; or (Version 5) numbers generated with security hash algorithms. In any case, primary programming languages all have algorithms to generate uuid values. The drawback to uuids is that they are a pain to type if you want to manually test a query. But they are necessary in distributed computing environments, and most of the time a programming language generates the value or retrieves it from an existing table.

Sparse Data and Flexible Columns

The second part of Figure 13.2 hints at how columns in non-relational systems are different from those in relational tables. In a relational DBMS, each table has a fixed set of columns and each row/column cell has exactly one value. Once the relational rules are discarded, it is easier to think of a row as just a collection of bytes. The row key retrieves those bytes, but the application can store or retrieve almost anything in that space. Most systems define the column space as a data map which is just another set of key-value pairs.

One benefit to the mapping approach arises for data with many missing entries, or **sparse** tables. Each row only stores the data columns that exist so if much of the data is missing no space is wasted. For instance, one Customer entry might have values only for LastName and FirstName, so the map contains only those two entries. Another row might have several items, including a photograph. Application do have to be slightly cautious and test for missing values when requesting items within a row.



Some systems and some developers take this approach to the extreme and claim that the flexibility enables them to store different key items in every row. For example, one Customer row might contain entries for LastName, FirstName, and Phone, while another could hold data for FamilyName, Nickname, and Skype address. Even if a system does support this level of flexibility, it should be avoided. Changing column names/keys means that every application needs to know all of the possible values and test for them within the code. Making data storage more “flexible” at the cost of making program code harder to write, read, and test is a bad tradeoff in most situations.

Another approach to flexible column data is the ability to store complex data within a single cell (or column). For example, Cassandra supports set, list, and map data types. A **set** is an unordered collection, a **list** has an index order (1, 2, ...), and a **map** is a collection of key-value pairs. Each of these can be used to store multiple entries in a single column for one row of data. They should be used only for small lists because the query system will retrieve every value at the same time.

Figure 13.3 shows an example where a set or list might be used to enable customers to enter multiple e-mail addresses or multiple phone numbers. For instance, to store two e-mail addresses, a set could define

```
E-mail = {'John@Jones.com', 'JJones@gmail.com'}
```

A map uses a key to define the difference between the two addresses such as

```
E-mail = {'Home': 'John@Jones.com', 'Work': 'JJones@gmail.com'}.
```

Collections support any data type and the map can contain any key definitions needed by the application. But, the application programmers need to remember all of the keys and handle them within the program, so the programming can become more complex and subject to more errors. In some ways, the collection types are not radically different from the relational DBMS. Most relational systems today support an XML data type which makes it possible to store complex data, including lists and collections in a single column. And the same warnings apply to using the XML data types—they make the application programming more complex and harder to test and debug.

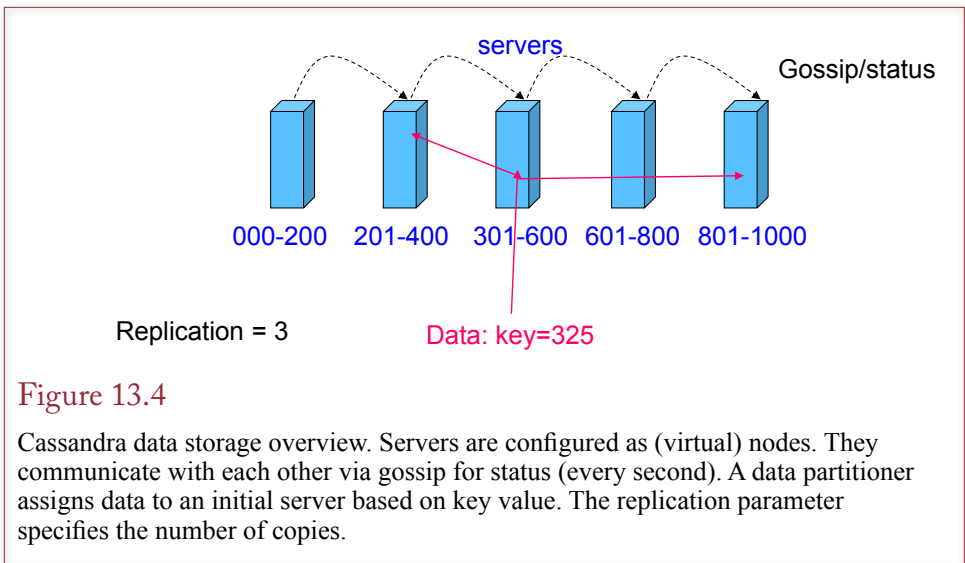


Figure 13.4

Cassandra data storage overview. Servers are configured as (virtual) nodes. They communicate with each other via gossip for status (every second). A data partitioner assigns data to an initial server based on key value. The replication parameter specifies the number of copies.

Distributed Data

The non-relational systems are built from the ground up to handle highly distributed data. Cassandra has a particularly interesting version because it is peer-to-peer instead of using a central server approach. Every server node in the Cassandra network is independent and shares data directly with other nodes. No single node coordinates or controls the others. Figure 13.4 shows the basic elements of the Cassandra network. A data partitioner defines a range of key values for each node. When the database is created, the designer specifies a replication level—3 in this example. Data is then written to the appropriate server based on the value of the key and replicated to the specified number of servers. Each node communicates with the others via a **gossip** channel to share status information. If a server fails to respond, it is moved from the active list and others pick up the lost key range. Similarly, when a new server is added, the key ranges are redefined and gossip is used to synchronize the data updates across the new server.

Distributed data in Cassandra is actually much more sophisticated, but most of the details are not important to the design or queries so they are not covered in this chapter. Basically, servers can support virtual nodes which simplifies replication assignments. More interestingly, it is possible to incorporate the physical layout of the servers into the replication design. For example, servers located in the same rack are connected by high-speed networks and can quickly share data; but they are more at risk for collective failure (e.g., power or network). Nodes in a different data center might be a different geographical location, so data is more protected if spread across centers, but updates are slower. Cassandra data models support defining these characteristics and the data partitioner optimizes the data replication.

Other systems have different features, and some rely on a central server to coordinate data storage and status messages. With Cassandra, data queries and write operations can be connected to any node and the system will function the same way each time, even if one node crashes or becomes inaccessible. Most of the non-relational systems use some form of distributed data storage; both to provide data protection through replication and to improve performance by having multiple servers and multiple drives handling the data.

In terms of physical computers, the server processing is important, but the data storage methods are more important. Because Cassandra automatically handles replication of the data, RAID 1+0 drives are not recommended. RAID 1+0 drives make physical replicas of data being written so if one drive fails the others can rebuild the content. But Cassandra already handles the replication so using RAID 1+0 just wastes space. RAID 0 drives are still useful because they multiply the access speeds with physically independent disks. But high-end Cassandra implementations still recommend even faster **solid state drives (SSDs)** for all data storage.

Other than performance and backup, the nice feature of the distributed systems is that they are invisible to the application. The application (writing and retrieving data) just issues queries and the DBMS handles all of the details automatically. Of course, setting up and monitoring the distributed network takes additional time. But, application transparency is important because the data storage can be rescaled at any time without altering the application.

Consistency and Integrity

Largely because of the distributed structure, one of the key aspects of non-relational databases is the limitations on consistency and integrity. A key strength of traditional relational systems are the built-in controls to ensure data consistency; which makes them valuable for business transactions. The problem is that absolute consistency is difficult to guarantee in a widely distributed system. It would require that all nodes maintain communication during all updates. Worse, strict consistency can require that some transactions (reads and writes) be delayed until all nodes are consistent. But the point of a distributed system is that it should be able to handle short-term failures in some nodes and connections; and maintain high performance even under heavy load.

Figure 13.5

Cassandra tunable consistency. Developers can choose a consistency level for any write (or read) operation. The lowest level (ANY) has the least delays. The ALL level requires all replicas to be updated before continuing.

Level	Nodes	Description
ANY (lowest)	1	Write will still succeed if a hinted handoff has been written.
ONE, TWO, THREE	1, 2, or 3	Write must be logged and committed to the specified number of replica nodes.
QUORUM	Replication/2 + 1	Write logged and committed to at least half the replication nodes.
LOCAL_QUORUM	Same data center	Same as quorum within the local data center.
EACH_QUORUM	All data centers	Same as quorum within all data centers.
ALL (highest)	All replicas	Write must be logged and committed to all replicas.

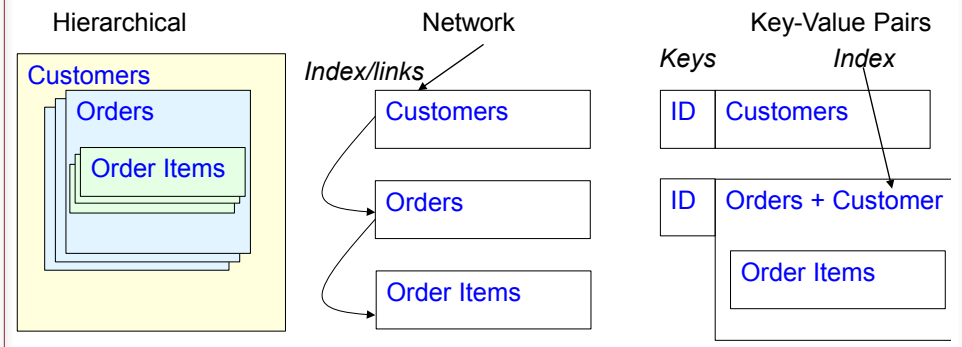
Non-relational systems relax the constraints on absolute consistency and allow nodes to be inconsistent—at least for a short period of time. Actually, in terms of read and write transactions, Cassandra provides the ability for developers to specify the desired level of consistency, calling it **tunable consistency**. As shown in Figure 13.5, write consistency specifies the number of replicas that need to return an acknowledgement of success. The lowest level (ANY) operates with the least delays. The highest level of consistency (ALL) requires all replicas to be updated and committed before continuing. It is similar to the consistency requirements in traditional relational systems.

However, several consistency issues exist beyond read and write transactions. First, non-relational systems do not support referential integrity. The DBMS does not have a method to verify that foreign keys are valid. For example, a Customer-ID entered into a Sale table could be wrong. Similarly, when a row is deleted from the Customer table, there is no automatic mechanism to delete the corresponding data in tables that use that data (cascade delete). So the programmer is responsible for maintaining data integrity.

A second consistency issue arises because non-relational designs often duplicate data to improve performance. Consider the standard Sale and Customer tables again. Instead of relying on the CustomerID to look up the customer name in the Customer table, many designers will duplicate and store the customer name in the Sale table. That way the name (and other data) can be retrieved at the same time the sale data is read, without requiring an additional lookup in the Customer table. But again, the DBMS has no method to ensure consistency of data. Changing the address in one location does not affect the others. This action might have some use—the sales data could contain different addresses for a customer depending on when the sale was made. Many Web databases rely on keeping different values of data at different points in time. But users do need to remember that the data can be inconsistent at times.

Figure 13.6

Non-relational storage affects how data can be retrieved. Hierarchical systems stored and located data by starting at the top level and working down. Network allowed more flexibility by separating the tables and linking them through indexes that had to be built to support queries. Key-Value combines elements of both by using indexes on keys to locate individual rows. Any other searches require additional indexes.



Optimizing Data Storage for Queries

Figure 13.6 shows that the original non-relational DBMSs (hierarchical and network) were relatively rigid in the way data was retrieved. Hierarchical models adopted features of paper filing cabinets. A cabinet (database) would hold folders of Customers stored alphabetically. Each folder would contain the individual orders, and the orders would contain the detailed items purchased. As long as you only wanted to retrieve data by Customer and then find individual orders, the system was relatively fast. But, if you wanted to find customers who ordered a specific product, the system would have to start at the top and go through every customer and every order. The Network model attempted to support these additional searches by separating the storage of each table and then building indexes and links to all of the data. So, if the developer knew in advance that someone might want to search for customers who ordered a specific product, an index could be built on the ItemID, and then the back-links could be traced to identify the specific customers.

The newer key-value systems adopt some elements from both of these models (as well as a couple from the relational model). Data stored in separate tables is indexed by the primary key. Using multi-level indexes and the power of distributed data, the storage and retrieval of the associated row data is fast. For even faster access, data is often duplicated. For example, designers might include the Customer name and shipping address with the Order data. Likewise, the Order Item data might be stored within the Orders row, similar to the way it would be handled with a hierarchical model. The critical design concept to remember is that the DBMS can only retrieve data using the primary key. In fact, queries probably cannot use other data in WHERE conditions. This limitation is demonstrated in the query section. However, a few systems, particularly Cassandra, support the creation of additional indexes that can be used for searching. In the example, if the developer knows that someone will want to search for customers who ordered a specific product, a separate index can be created using ItemID on the Orders table.

The most important point of this section: Unlike the relational data normalization rules, there is no fixed method for defining data storage using key-value pairs. Instead, the designer must know how the data will be generated and queried and then design the data storage to optimize the overall performance. Figure 13.7

Figure 13.7

Non-relational storage affects how data can be retrieved. Hierarchical stored and located data by starting at the top level and working down. Network allowed more flexibility by separating the tables and linking them through indexes that had to be built to support queries. Key-Value combines elements of both by using indexes on keys to locate individual rows. Any other searches require additional indexes.

1. Identify the basic data to be stored.
2. Do a base data normalization to identify potential tables.
3. Identify all the ways an application will need to query the data.
4. Identify the primary key-value pairs (base tables).
5. If needed, duplicate data to improve performance.
6. Create additional indexes to support queries not covered by primary keys.
7. Test performance, combine data and reduce indexes if needed.

defines the basic steps that can be used to design data storage for a key-value DBMS. But, each design will be unique and require experimentation to find the best storage approach. The basic rule is that any data that can be accessed via a key will be relatively fast. Storing all related data in one row is faster—even if it means duplicating some information. Searching on non-key items requires creating additional indexes, but adding indexes slows down performance on updates and inserts because the indexes have to be rebuilt. If a design for a transaction system starts to require dozens of indexes, it will probably be better to eliminate all but the essential indexes and create a data warehouse to enable managers to perform additional searches on a copy of the database (see Chapter 9).

Ultimately, getting the best performance out of a key-value pair database requires experimentation with the design. Eventually, as the system software grows and stabilizes, perhaps computer scientists will be able to develop rules to improve the designs.

Cassandra

How are databases designed and queried using Cassandra? All of the key-value pair DBMSs are slightly different and each application requires a custom database structure. Although the general elements are similar, it is important to look at a specific DBMS and a specific problem to understand the features and constraints of the tools. Cassandra is one of the leading non-relational DBMSs, with strong developer support including a company (www.DataStax.com) that specializes in advancing the software and providing support. This level of support is useful to help ensure the DBMS will survive for at least a few years. Remember that these tools are relatively new and many companies are experimenting with different approaches. A second useful feature of Cassandra is that it has an interactive query system that makes it possible to experiment with the database without needing to write code for each example. Ultimately, each application still has to be written in some programming language, but it is helpful to be able to test designs and queries before writing code.

Installation Issues

One of the strengths of the non-relational systems, particularly Cassandra, is the ability to run as a distributed database on multiple server nodes. A drawback to running multiple servers is the cost of the servers—both hardware and software licensing costs. Consequently, most of these tools are open-source projects that are also designed to run on open-source systems—reducing the licensing costs. (Some of the hardware costs can be handled by using cloud-based computing as described in the last section of this chapter.) A challenge with open-source operating systems is that they can be harder to install and manage than Windows-based systems. Also, several variations exist, leading to differences in installation and operating procedures. Cassandra is written in the Java programming language, which means that versions have been compiled to run on most systems (including Windows—but it is not recommended). Cassandra also requires the Python programming language (for the query tool).

It is highly recommended that Cassandra be installed on a virtual machine running an open-source operating system. The Debian version is relatively straightforward to install and it uses packages to install most software which simplifies installation of applications and tools. DataStax has packaged versions and instruc-

1. Install Virtual Machine Server—open source: Debian
<http://www.debian.org/releases/stable/installmanual>
2. Sun/Oracle version of Java: at least JRE and JNA
 - a. Java –version (default is open source Java)
 - b. Download and install from Oracle, then set as default
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
3. Download and install Cassandra from DataStax (Community edition)
4. Several configuration steps for production are not needed for the sample and testing. And only one node is needed.
 - a. Download and install the PetStoreWeb files.
 - b. Unzip and copy them to a folder
 - c. In terminal mode, run the cql command to install:
`cqlsh -f PetStoreWeb.txt`

Figure 13.8

Summary installation steps. Follow the detailed installation steps in the Apache Cassandra Documentation from DataStax: <http://www.datastax.com/docs>. Be certain to install the recommended version of Java before attempting to install Cassandra.

tions for a couple of the more-popular Linux variants including Debian. Figure 13.8 outlines the basic steps, but several details can be required for each step so you might want to obtain assistance from a local Linux user. Installing the proper version of Java and setting it as the default version is one of the more complex steps.

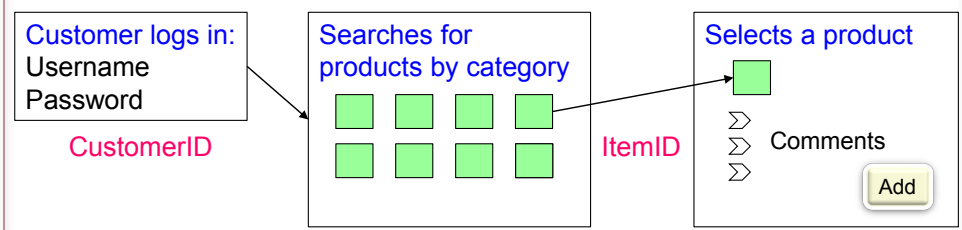
Cassandra has several useful configuration options for production systems. These options are used to initialize and coordinate the multiple nodes in the distributed system. They are not needed for the sample demonstration files—which are tiny. They are critical for optimizing performance in large production databases, but they focus on the distributed networking issues so are not covered in this chapter.

Pet Store Web Example

Many of the non-relational benefits arise when millions of people need to store and access data—particularly on the Web. Figure 13.9 shows a common extension to the Pet Store case. Customers will choose a category and see items in that

Figure 13.9

Pet Store Web Site Usage. Customers see merchandise items based on a selected Category. When an item is selected, the page displays the product, description, price, and a set of comments from other customers. Customers who are logged in can add their own comments.



- Find CustomerID given the Username
- List Merchandise given a Category
- Display Merchandise data given an ItemID
- List all comments and customer screen name for a specified ItemID
- Insert a new comment given ItemID and CustomerID

Figure 13.10

Initial application queries. These queries will affect the database design. Lookups by ID are handled as primary keys. Other lookups will require additional indexes.

category. When they select a specific item, the details of the product are shown along with a set of comments entered by other customers. Once the customer is logged in, he or she can add a comment to the list. A customer can make only one comment on a given product, but some might want to change the comment later. The basic process should be familiar, because many Web sites support comments by users. With potentially millions of customers and their comments, the database could become large. It is also important that the page displaying a product be retrieved and generated quickly—customers will not tolerate delays.

Think about the usage display for a minute in terms of data. The existing (relational) database already has tables for Customer and Merchandise. Those basic tables will probably transfer cleanly—but the ID values will have to be changed to uuids. A new table will probably have to be defined for the Comments, and the details are covered in the next section. But also think about the potential queries needed by the application. Figure 13.10 shows the main queries that will be needed by the application. These queries are important because they will affect the design of the database. Queries using IDs will become key-value pairs or primary keys, and the others will have to be handled by separate indexes.

Database Design

Figure 13.11 shows the three tables needed for the Pet Store Web application. The Customer and Merchandise tables are essentially copied from the relational design, except that the primary keys are uuids. The ItemComments table is new. Notice that the keys are the same as they would be in a relational database: ItemID + CustomerID. Also, notice the duplication of the ScreenName in the ItemComments table (from the Customer table). By placing this small piece of data that is displayed with the comment inside the comment table, it saves a lookup into the Customer table so it can be retrieved significantly faster. But the application will be responsible for maintaining data consistency.

To understand the challenges of design, consider the ItemComments table. The first question to ask: Why is it a separate table? Why not just store the comments within the Merchandise table? It would be straightforward to add a column to the Merchandise table that could hold a repeating set of data as comments. The drawback to storing all comments in one row of the Item table is that retrieving a row of data always retrieves the entire row. If a popular product gets thousands of comments, it could take too long to retrieve that one row. Additionally, it is more difficult to include the other attributes (CustomerID, Date, Rating, and so on). It can be done—but only by creating a fairly large mapped object inside each row, which slows down retrieval and processing. Using both ItemID and CustomerID as keys also makes it straightforward to search for comments by Customer.

Customer	Merchandise	ItemComments
<u>*CustomerID</u> FirstName LastName ScreenName Username Password Email	<u>*ItemID</u> Description QuantityOnHand ListPrice Category	<u>*ItemID</u> <u>*CustomerID</u> CommentDate ScreenName Title Comment Rating

Figure 13.11

Data tables for Pet Store example. Customer and Merchandise are base tables and the ID key columns are uuids. ItemComments are new and each customer can comment once on a given item (but can change the comments later). Notice the duplication of ScreenName in the ItemComments table.

Before creating the tables, Cassandra requires that all tables be defined within a **keyspace**, which is similar to a schema in Oracle or SQL Server. It simply separates one collection of tables from another by assigning a name. Generally, a keyspace is defined for each application. The syntax to create a keyspace is straightforward, as is the command to switch to a new (or different) key space:

```
CREATE Keyspace PetStoreWeb;
USE PetStoreWeb;
```

Tables are created within a keyspace. In earlier versions, and in some existing documentation and error messages, a table was called a **column family**. In current versions, the two terms are synonyms, but it is easier to think of the data as a table than a family. Tables hold **columns**, which are not exactly the same as SQL columns; but in most situations, they are similar. The differences are greater in terms of data storage (and keys) because the columns are actually stored as key-value pairs within each row.

Note that keyspace, table, and column names in Cassandra are normally not case sensitive. The key words (CREATE) are also not case sensitive. But this book uses case to highlight the key words and names to make them easier to read. There are two catches in Cassandra: (1) Double quotes placed around a name make it case sensitive and quotes are then required in all future usage. (2) Cassandra automatically converts all names to lower-case when it stores them. A few procedures (notably COPY) seem to automatically use quotes, so if a command does not work, try entering all names in lower-case.

Figure 13.12 shows the CREATE TABLE commands used to define the tables in Cassandra. The syntax of the command is similar to that in SQL but it has different data storage options, which are not shown here. The default options are fine for the sample database. Also, notice the uuid data type for each of the ID columns. These values will have to be generated by the application. Note the specification of both columns as the PRIMARY KEY for the new ItemComments table. Although this syntax looks similar to SQL, the effects are quite different as will be explained shortly. Finally, notice the use of the set<text> definition for the e-

```

CREATE TABLE Customer(
  CustomerID      uuid,
  FirstName      varchar,
  LastName       varchar,
  ScreenName     varchar,
  Username       varchar,
  Password       varchar,
  Email          set<text>,
  PRIMARY KEY (CustomerID)
);

CREATE TABLE Merchandise (
  ItemID         uuid,
  Description     varchar,
  QOH            int,
  ListPrice      decimal,
  Category       varchar,
  PRIMARY KEY(ItemID)
);

CREATE TABLE ItemComments(
  ItemID         uuid,
  CustomerID     uuid,
  CommentDate    timestamp,
  ScreenName     varchar,
  Title          varchar,
  Comment        varchar,
  Rating         int,
  PRIMARY KEY (ItemID, CustomerID)
);

```

Figure 13.12

Data tables for Pet Store example. Customer and Merchandise are base tables and the ID key columns are uuids. ItemComments are new and each customer can comment once on a given item (but can change the comments later). Notice the duplication of ScreenName in the ItemComments table.

mail address in the Customer table. Defining it as a set means that the table can hold multiple e-mail addresses for each customer. It is still up to the application to handle the collection and editing of that data and it slightly complicates the syntax for storing them, but it demonstrates the additional flexibility of collections.

Figure 13.13 shows the primary data types available in Cassandra. For business applications, the most common data types are the standard uuid, int, varchar (or text), decimal (for currency values), timestamp, boolean, and possibly float (for percentages). Avoid the more exotic types of counter and varint, and you should almost always use varchar instead of ascii, which does not support international characters. Collections are defined with the set, list, and map keywords followed by the type of data that will be stored. Most collections will use the text data type.

Primary Keys

The Primary Key definition syntax is similar to that used in SQL. However, primary keys are considerably different in Cassandra than in relational databases;

Data Type	Description	Data Type	Description
ascii	US ASCII text string	inet	IP address as string
bigint	64-bit signed integer	int	32-bit integer
Blob	Binary object/picture	text or varchar	UTF-8 string
boolean	true/false	timestamp	Date+ time, 8 bytes
counter	64-bit integer, but...	uuid	Type 1 or 4 uuid
decimal	variable precision decimal	varint	Arbitrary-precision int
double	64-bit floating point	Java classes	Optional classes in Java
float	32-bit floating point		

Figure 13.13

Cassandra data types. The most commonly used types in business applications should be: uuid, int, varchar (or text), decimal (for currency), and timestamp.

particularly when the primary key contains more than one column. Recall that data is stored as a key-value pair. Specifically, each row must have a key that is used in the index to find a specific location. In case you are curious, that key cannot be a counter type, and uuid is by far the most common.

A critical difference with Cassandra is that when the primary key consists of multiple columns or a **compound primary key**, only the first column is used as the partition key—which determines where data is stored. The other columns are clustering columns and data is stored together. In fact, the clustering columns are used to sort the data. In the ItemComments example of PRIMARY KEY (ItemID, CustomerID), the ItemID determines where the data row is stored, and the comments are stored sorted by the CustomerID. Depending on the application goals, it might be useful to change the keys to: PRIMARY KEY (ItemID, CommentDate). This definition would store the comments in order of date, making it easy to retrieve and display them in that order. However, the timestamp data type only splits time down to seconds, so the application would have to be careful to ensure that no two comments are ever written with the exact same date and time.

In some situations, it can be useful to partition the row data on more than one key value. The **composite primary key** is used to define multiple columns as the partitioning key by using a second set of parentheses, such as PRIMARY KEY (ItemID, CustomerID), optional columns). The difference with a composite key is important. Think of it as defining the data storage by both keys: ItemID + CustomerID. Without the parentheses, only the first column partitions the storage, with the extra parentheses both keys define the storage location—and both are required to retrieve the data. Because Cassandra retrieves all rows based on the partition key, the difference affects the queries.

With a simple compound key (ItemID, CustomerID), a query would retrieve data by specifying ONLY the ItemID value, which would return all of the comments made by each customer. With a composite key ((ItemID, CustomerID)), a query requires BOTH the ItemID and CustomerID values to return exactly one row. The Pet Store application has to use the simple compound key, because when it displays an item, it knows only the value of the ItemID, not all of the Custom-

Compound key: ItemID, CustomerID

ItemID	CustomerID	Data
588e633f...	7f81c5d6...	Not big enough...
	804a2cdb...	Easy to assemble...
7ee762a1...	04201f56...	Smells bad...
	3e137d55...	Yummy...
	538adbba...	Too big...

Composite key: ItemID + CustomerID

ItemID	CustomerID	Data
588e633f...	7f81c5d6...	Not big enough...
588e633f...	804a2cdb...	Easy to assemble...
7ee762a1...	04201f56...	Smells bad...
7ee762a1...	3e137d55...	Yummy...
7ee762a1...	538adbba...	Too big...

Figure 13.14

Compound v. Composite key. The compound key partitions by the first column ONLY. A query specifies just the value for ItemID and returns comments by all customers for that item. A composite key partitions by both columns. A query must list both the ItemID and CustomerID values to retrieve exactly one row. The problem is that there is no easy way to get the list of all CustomerID values in the case of the composite key.

erIDs who have entered comments. And realistically, there is no way to obtain the list of CustomerIDs—without testing every possible value, which would be horribly slow.

Figure 13.14 illustrates the difference using some of the sample data. The compound key uses only the first column (ItemID) to partition (store and retrieve) the data. So a query needs to know only the value of the ItemID and it will return comments from all customers in that “row.” The composite key uses an extra set of parentheses to partition by both the ItemID + CustomerID columns. A query needs values for both ID columns to retrieve exactly one row of data. The composite key approach is faster—if the application always has values for both ID columns. In the Pet Store example, the usage description says that the application knows only the ItemID, so the design needs to use a compound key based only the ItemID column.

Some of these points might seem a little confusing at the moment. Do not panic. They are easier to understand once you see the limitations of queries as explained in the next section. So, read the section on queries and then come back and re-read the design guidelines. Remember that data design and storage depend on the queries that need to be answered, so the design (and learning) process is iterative.

Initial Queries

The real differences with a non-relational DBMS arise when looking at queries—which partly explains the misnomer: NoSQL. Interestingly, Cassandra now has an interactive query language named **CQL** (Cassandra Query Language). CQL

<pre>SELECT Count(*) FROM Customer;</pre>
<pre>count ----- 99</pre>
<pre>SELECT * FROM Customer WHERE CustomerID=71c1da88-88af-4217-aa41-332ea3d33ae9;</pre>
<pre>customerid email firstname lastname... -----+-----+-----+-----+ 71c1da88... {BCummings@gmail.com, bignotes@gmail.com} Brent Cummings </pre>

Figure 13.15

Two basic CQL queries. The basic CQL syntax is similar to SQL but much more limited. Count is the only aggregate function supported. The SELECT clause lists columns to retrieve and the WHERE clause can be used to specify primary key entries.

borrowing some of the basic structure of SQL, which makes it a little easier to write the syntax, but ultimately, the queries have almost nothing in common with even simple SQL queries. But it is not just a limitation of the query system. The restrictions arise because of the way the DBMS stores the data, which is done to improve performance for specific queries. This section shows some basic queries using the sample Pet Store Web data. A different approach is used here compared to learning SQL: Several of the initial queries will not work—specifically to demonstrate the limits. If at all possible, you should install Cassandra and the sample database and run the queries to follow along. Start the CQL processor by opening a terminal window and typing: `cqlsh` (for CQL shell). Remember to enter the command to use the keyspaces: `use PetStoreWeb`; Note that the use of uuids makes it a challenge to type some of the queries.

Figure 13.15 shows two basic CQL queries using the SELECT command. The first uses the Count function to return the number of rows in the table. Count is the only aggregation function supported by CQL, but it can be useful to identify large tables. The second query looks similar to a simple SQL query. The SELECT clause can use `*` for all columns or the names of individual columns can be entered. The WHERE clause is even more restrictive. Initially, the only conditions you can enter in the WHERE clause are conditions on the primary key (CustomerID in the example). Remember that rows are stored as key-value pairs and initially data can be retrieved only through the primary key.

Figure 13.16 shows some basic queries to experiment with variations of the WHERE condition in the SELECT command. The bottom line is that the WHERE clause can contain conditions that only use the primary key and an equals sign. It does not even support conjunctions (And, Or). However, it does support the IN () condition which takes multiple key values and finds matching rows based on equality—which is equivalent to several OR conditions. A token () function exists which does support inequality conditions. However, the token function converts the values to their hashed-storage values and then makes the comparison. The default hashing function essentially randomizes the values, so the results are usually

```

SELECT * FROM Customer WHERE
CustomerID= 71c1da88-88af-4217-aa41-332ea3d33ae9 OR
CustomerID= 378feb73-34cd-451f-90a9-a739a94c30f4;

>>> Error: Expected EOF at OR...

SELECT * FROM Customer WHERE CustomerID IN
(71c1da88-88af-4217-aa41-332ea3d33ae9,
 378feb73-34cd-451f-90a9-a739a94c30f4);

>>> Retrieves two rows.

SELECT * FROM Customer
WHERE CustomerID > 71c1da88-88af-4217-aa41-332ea3d33ae9;

>>> Error: Must use EQ or IN

SELECT CustomerID, LastName FROM Customer
WHERE token(customerid) > token(00000000-0000-0000-0000-000000000000);

>>> Retrieves random rows where the hash value is greater than the hash of 0...

```

Figure 13.16

Experiments with CQL SELECT. Initially, a table can be searched only by individual values of the primary key. Conjunctions (Or, And) and inequalities (<, >) are not allowed. The IN (...) condition is used to find multiple values in one command. The token () function does support inequality values but the comparison is made based on the hashed value of the key which is probably random.

meaningless. However, Cassandra does support a ByteOrdered partitioner, which arranges tokens in the same order as the keys. If this partitioner is specified as the storage mechanism when the table is created, the token function might be useful.

The purpose of the examples is to demonstrate the constraints of the query system. Although the SELECT command might look a little like simple SQL, it is far more limited. Remember that the data storage places strong limits on what can be done to retrieve data. At the moment, the SELECT command can retrieve only data based on specified values of the primary key.

Indexes

Obviously, retrieving data based only on primary keys is too restrictive. Look again at the usage goals for the Web site. At a minimum, it requires finding a Customer based on Username, and retrieving Merchandise based on the Category value. Neither of these columns is in the primary key. In fact, Category could never be a primary key column because it is not unique. So how can Cassandra retrieve data using those conditions? The answer is to create indexes. An index is basically just another set of key-value pairs.

```

SELECT * FROM Merchandise
WHERE Category = 'Cat';

>>> Error: No indexed columns present...

CREATE INDEX idxMerchandiseCategory
ON Merchandise (Category);

SELECT Category, Description, ListPrice
FROM Merchandise
WHERE Category = 'Cat';

```

category	description	listprice
Cat	Cat Bed-Small	25
Cat	Cat Litter-10 pound	8
Cat	Cat Food-Dry-10 pound	10
Cat	Cat Food-Dry-5-pound	7
Cat	Cat Toy	3
Cat	Cat Food-Dry-25 pound	18
Cat	Cat Food-Can-Regular	0.5
Cat	Brush-Soft	8
Cat	Cat Food-Can-Premium	1
Cat	Cat Bed-Medium	35
Cat	Flea Collar-Cat	6
Cat	Collar-Cat	8
Cat	Litter Box-Covered	15
Cat	Litter Box	8

Figure 13.17

Creating an index to search by non-key columns. The CREATE INDEX command builds an index that can be used to add new conditions to a SELECT statement. The application requires searching by Category.

Figure 13.17 shows how to create an index on Merchandise Category so that the application can retrieve all items that match a specified category. The CREATE INDEX syntax is similar to SQL:

```
CREATE INDEX indexName ON table (column);
```

Technically, the index name is optional, but it should always be used because then the DROP INDEX command can be used to remove it later. Note that primary keys cannot be indexed—but it would not make any sense to do that. After the index has been created, the specified column can be used in the WHERE clause of a SELECT query—but only with an equals sign. In production databases, the CREATE INDEX command should be issued when the tables are CREATED and before data is loaded.

Figure 13.18 shows an interesting effect of using an index. Remember that only the Category column has been indexed. Yet, now the SELECT statement supports additional conditions in the WHERE clause—as long as the condition applies to a non-key column and the **ALLOW FILTERING** clause is added to the query. CQL will provide a warning if the ALLOW FILTERING clause is missing. Inequality searches are potentially expensive and slow, so be certain that they are

```

SELECT Category, Description, ListPrice
FROM Merchandise
WHERE Category = 'Cat'
AND ListPrice > 10
LIMIT 10
ALLOW FILTERING;

```

category	description	listprice
Cat	Cat Bed-Small	25
Cat	Cat Food-Dry-25 pound	18
Cat	Cat Bed-Medium	35
Cat	Litter Box-Covered	15

Figure 13.18

Secondary indexes enable additional conditions. Conditions on other (non-indexed) columns can be added as long as the `ALLOW FILTERING` phrase is added at the end. The `LIMIT n` command can be used in any `SELECT` query and defaults to 10,000 rows if not specified.

necessary before using them. It is often useful to include the `LIMIT` statement to restrict the number of rows returned. In fact, Cassandra has a default value of 10,000 for the number of rows returned for any query. Queries that might return more rows need to use the `LIMIT` statement to increase that value. But, before blindly inserting a large number, ask yourself why you need a query to return so many rows. No one is going to read that many, and a large value would slow down almost any Web site. The statement would be useful when it is necessary to extract large chunks of data to transfer to other systems, but small values would be used in production applications.

Note that the additional clause (`AND ListPrice > 10`) can be used only if the `Cat` condition is used. Try running the `SELECT` query with just the `ListPrice` condition and it will generate an error (no index). If a new index is created for `ListPrice` the inequality condition by itself (`ListPrice > 10`) still will not work—because indexed columns can be searched only using equality conditions. The basic `SELECT` search rule is that a `WHERE` clause can search for only primary keys

Index Issues

Technically, indexes are supposed to be built on existing data as soon as the `CREATE INDEX` command is issued. However, some queries in testing returned no matching values after the index was created (Cassandra 1.2). In production situations, it is best to create all indexes before loading data. For the examples in the book, it might be necessary to create the index, remove the data, and reload the data:

```

CREATE INDEX ON Merchandise(...);
TRUNCATE Merchandise;
COPY petstoreweb.merchandise(itemid, description,
qoh, listprice, category) FROM 'Merchandise.csv';

```

Production databases also require periodic use of the `nodetool` command to repair the database or force updates with the UNIX command line:

```
nodetool repair
```


and indexed columns using equality conditions. It is possible to add additional filtering conditions but only on the other non-key columns which are stored in the same row.

Once more look at the usage plan for the Pet Store Web application. What searches are required in the application that will require indexes? The second issue is the search by Username. When a person logs in, only the Username and Password are provided. The Username has to be unique, so the application needs to retrieve the Customer row with that Username and then verify that the password values match. Because Username is not a primary key, it needs to be indexed:

```
CREATE INDEX idxCustomerUsername ON Customer (Username);
```

The index can be tested by searching for a known Username (BCummings):

```
SELECT CustomerID, Username, Password
FROM Customer
WHERE Username='BCummings';
```

Querying Tables with Compound Keys

The Pet Store Web application needs one more SELECT statement—to retrieve the comments for a given Item. This data is in the ItemComments table which has a compound primary key (ItemID, CustomerID). What command is needed to retrieve this data? Does it need a secondary index? The answer to the second question is “no,” which makes the SELECT query straightforward.

Figure 13.19 shows the Pet Store Web query for retrieving the first 10 comments for a specific ItemID. From the usage diagram, when a user clicks on an item, a page is generated that displays the basic item information (a different query), and then displays some of the comments for that item. The ItemID value is available to the application from the Web click. The designers decided to limit the comments to no more than 10 per page to improve the page performance. The query is relatively simple, which is good. This result is exactly what is needed

Figure 13.19

Queries on compound primary keys. Only the first column in a compound key controls storage so only the ItemID is needed for a search condition. No indexes are necessary and the query will return all rows with the specified key value. A lower LIMIT value is useful for Web pages.

```
SELECT CommentDate, ScreenName, Title, Comment, Rating
FROM ItemComments
WHERE ItemID=7ee762a1-3a27-42a0-a51e-e7988250ecd5
LIMIT 10;
```

commentdate	screenname	title	comment	rating
2014-11-14...	Gazer33	Smells...	The smell is horrible...	4
2014-11-01...	Caged19	Yummy...	My human/slave feeds...	5
2014-15-21...	Cathouse	Too big...	OK I only have one cat...	3
2014-03-07...	RedStar	Not...	Not sure it matters...	3

<pre>SELECT ItemID, CommentDate FROM ItemComments WHERE CustomerID=9f9f66c2-a949-4f60-b21b-1ec95158583c ALLOW FILTERING;</pre>	
itemid	commentdate
-----+-----	
563907d0-16bf-4b17-b516-3f42b7c787b7	2013-02-10...
7cbc9858-3cf6-41e7-aba3-db09cc27ebbb	2013-02-03...

Figure 13.20

Query a compound primary key on the second column. The second (and later) columns in a compound key effectively already have an index and can be retrieved directly with a WHERE statement as long as the ALLOW FILTERING command is used.

for the application, which is why the compound key was chosen in the database design. The key (ItemID, CustomerID) supports a many-to-many relationship that returns all of the customer comments for a given item.

From an application perspective, it might be nice to retrieve the Customer comments sorted by date, but CQL does not support any sorting by clustered columns (as of version 1.2). Instead, the application could read the rows into an array and then sort the data in the code.

What about querying for comments made by a specific customer? As shown in Figure 13.20, because CustomerID is part of the primary key, yes the query will work—but it requires using the ALLOW FILTERING command. What about finding the Item information? CQL does not support any type of JOIN command so the Item data cannot be retrieved with a single query. Instead, the application would have to examine the initial results row-by-row, and then create a new query to retrieve the matching data from the Item table using a single ItemID value at a time.

The point of the example is that the database design was specifically chosen to make the first query easy—not just easy to write but easy and fast to execute. In fact, go back and look at the data storage again in conjunction with the queries used to retrieve the data. The application needed only two secondary indexes to use simple queries to retrieve all of the data. The data was stored in three tables in a distributed system that supports fast write and retrieval. All without using JOIN statements and extra lookups. But, the data tables had to be designed specifically to match the query needs for the application.

INSERT and UPDATE

Cassandra CQL also supports **INSERT** and **UPDATE** commands to add new rows or change the data in an existing row. The syntax for both resembles SQL, as can be seen from two simple examples:

```
INSERT INTO Customer(CustomerID, FirstName, LastName, ScreenName,
Username, Password, Email)
VALUES (469aac21-5600-47c3-882f-f7a1ca269ede, 'Jones', 'Jackie', 'JJJ',
'JJones329', 'password', {'JJones329@gmail.com'});
```

```
UPDATE Customer
SET Password='password2', ScreenName='JJ3'
WHERE CustomerID=469aac21-5600-47c3-882f-f7a1ca269ede;
```

Note the importance of listing the column names in the INSERT statement. The columns in the primary key are the only required columns, all of the others are optional, so the column names need to be listed to ensure the values are matched correctly to the columns. Observe the braces used in the syntax for the e-mail column because it is defined as a set. Lists use square brackets (['a', 'b']) instead. And mappings require braces and colons such as { 'cost' : '3200', 'name' : 'test15' }.

The UPDATE command changes the column values to the new items. Multiple columns can be set at one time, but the WHERE clause must specify exactly one row. So the UPDATE (and INSERT) command lack the power of the SQL versions. Still, it is convenient to use similar syntax.

A far more interesting twist is what happens if an INSERT command is issued with an ID value that already exists. For instance, assume the two commands have been issued as shown in the short example. Then enter a new command:

```
INSERT INTO Customer(CustomerID, Username, Password)
VALUES (469aac21-5600-47c3-882f-f7a1ca269ede, 'Jones329', 'password');
```

Note that the CustomerID value exactly matches the one used above. What will be the result of this command? An error message—because of duplication of the IDs? A duplicate row? Try entering the three commands in the order shown, and then issue a SELECT command to examine the values for the specified CustomerID. The answer is that the query processor knows that the CustomerID value already exists, so it effectively converts the INSERT statement into an UPDATE statement. The result is that the Username and Password columns are reset to the values in the last command for the specified CustomerID. Technically, this result means the UPDATE command is not really needed; but what it really means is that you must be careful with any INSERT commands to ensure that the ID values are new (and unique). It is the reason that uuid and timeuuid are important data types—because they ensure that INSERT commands will never overlap an existing ID value.

Cloud Databases

How does cloud computing benefit key-value pair databases? Cloud providers offer a variety of database tools including traditional relational and newer non-relational systems. Some of these DBMSs are available directly from cloud-computing companies, such as DynamoDB from Amazon and App Engine Datastore (bigtable) from Google among others. On the other hand, Cassandra also has an installation script for creating your own cloud using Amazon's EC2 computers. EC2 systems are virtual machines that can be configured quickly and essentially rented by the hour.

A major goal of cloud computing is provide a way to quickly scale an application to handle greater loads—without the need for high upfront fixed costs. Most clouds accomplish this task through distributed virtual servers. With public clouds, a company runs large data centers and installs thousands of servers connected to large-capacity networks. Other companies (you) then configure a virtual machine server and pay hourly (or monthly) rates for using the virtual machine.

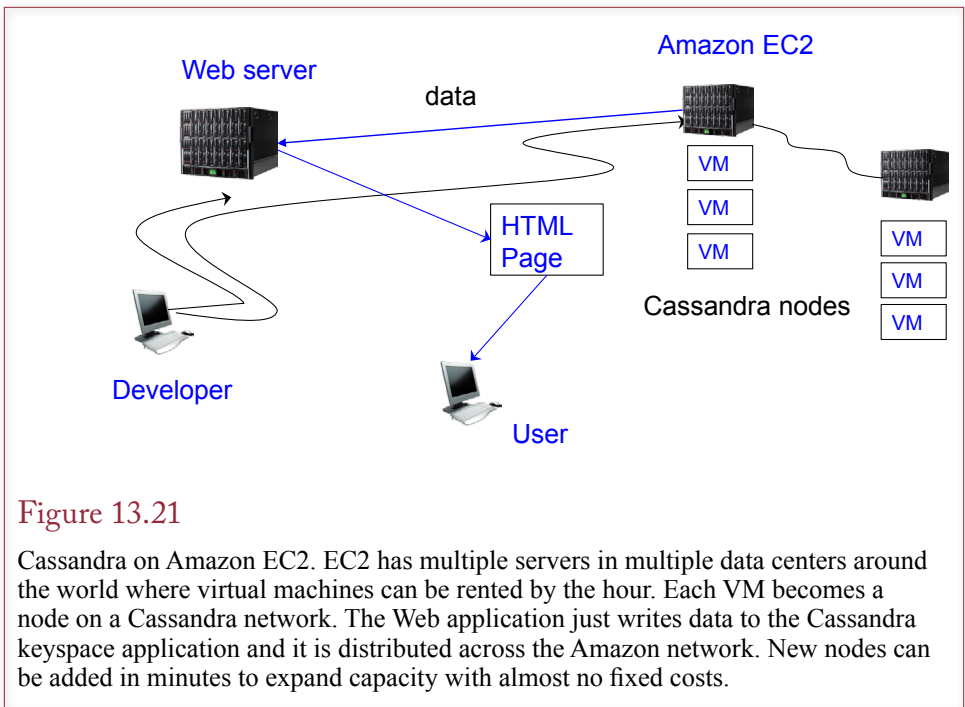


Figure 13.21

Cassandra on Amazon EC2. EC2 has multiple servers in multiple data centers around the world where virtual machines can be rented by the hour. Each VM becomes a node on a Cassandra network. The Web application just writes data to the Cassandra keyspace application and it is distributed across the Amazon network. New nodes can be added in minutes to expand capacity with almost no fixed costs.

If security is critical and cannot be handled through encryption, a company might choose to build its own data centers, but the distributed concepts and virtual machine configurations are largely the same. In both cases, you will be responsible for configuring and running the server and its applications.

DataStax provides a special copy of Cassandra and instructions for installing nodes on an Amazon EC2 cluster. As Figure 13.21 shows, with these tools, it is straightforward to build as many nodes as necessary on Amazon's system. Because Cassandra is designed to be distributed, it runs well on the distributed servers. Detailed configuration options provide control over replication within and across data centers to meet a variety of different Web needs. And more nodes can quickly be added as the number of users increases. Once the base system is configured, it is directly accessible to a Web application running on any server.

In the case of huge applications with millions of global users, the Web application can be written to connect to the closest geographical data center. Cassandra eventually replicates the data so it is available everywhere, even if a few nodes are inaccessible. Yet, most of the data is provided locally to the user, reducing the need to transmit data immediately around the world; which improves performance of the applications.

If you do not want to install and run your own copies of the DBMS, many tools (such as DynamoDB and App Engine Datastore) are available for hourly or monthly lease charges. In these cases, you simply define the tables and columns needed and tell the application to use the cloud databases instead of your own copy. Data distribution and backups are handled by the cloud provider.

The difference between the options largely comes down to cost. Running your own data centers involves a substantial upfront fixed cost, along with expertise and people to manage the centers on a day-to-day basis. Using virtual servers and

configuring your own databases through companies such as Amazon and Rack-space eliminates the fixed cost of installing the physical servers and networks. But, the monthly operating costs are higher than if you ran the same capacity on your own machines. The third option of using a prebuilt database cloud (DynamoDB or Google bigtable) has even higher monthly costs, but requires less expertise to configure and manage.

Many times it is difficult to predict the exact level of capacity needed for each month. Cloud-based systems have slightly higher marginal costs, but remove the need to guess ahead of time on the necessary capacity, because they are easy to expand or contract when needed. They also provide professional-level management and bandwidth to even tiny firms. Small firms can get the same high-level of distributed systems without having to pay huge upfront costs, just by purchasing capacity on the public cloud systems. If customer demand increases, presumably the revenues will also increase to cover the higher costs.

Summary

Web-based applications are important to many organizations. Some Web applications are like any other business application, but a few are radically different. Applications built to be used by millions of customers to store data online, such as the social-based sites, have different data needs than a standard business application. Performance and continuous accessibility are critical to these applications. Transactions and perfect data consistency are less important. Cost is another critical factor, particularly when the social features are offered at low or minimal cost.

Highly-distributed databases are an important tool to address these new applications. These systems can use hundreds or thousands of servers in data centers around the world to provide the data backbone for the application. To hold down the licensing costs, several new non-relational database systems were created to provide these features, by using open-source servers and software. Cassandra is a leading tool in many organizations, including Facebook. These new systems focus on storing data in key-value pairs; and replicate data automatically across multiple server nodes.

Database design is important to non-relational systems, but the rules are more flexible. It is useful to start with normalized tables, and then adjust the design to optimize performance for the specific application. The key point is that the design must match the individual needs of the application, so it is important to lay out the usage (use cases in OO terms) and identify all data retrieval needs.

Tables contain primary keys but the keys typically use uuid values which are safer to generate in a distributed system. Initially, rows can be retrieved from a table only through the primary key values; and those values must only use equality conditions. Queries on additional columns can be supported by adding a secondary index on that column, but those queries can also only use equality conditions. Adding too many indexes will slow down processing of new data, so the design has to be conservative. Table JOINS are not supported, so data is often duplicated by including base information in transaction rows. For instance, a Sale table would likely include Customer name and shipping address so that information can be retrieved automatically with each Sale instead of requiring an additional lookup.

Many-to-many relationships are indicated by specifying both columns in the primary key; just as in the relational model. However, the difference between compound and composite keys is critical. Compound keys are written as (ItemID,

CustomerID); while composite keys include an extra set of parentheses: ((ItemID, CustomerID)). Compound keys store data using only the first column, while composite keys use both columns to partition the data. Consequently, data stored with compound keys require only a value for the first column (ItemID) to retrieve a row; while composite keys require both values. A compound key actually returns multiple “rows” but a composite key returns exactly one row that matches all of the ID values. Typical applications will likely use compound keys instead of composite keys.

Queries in CQL use a simplified SELECT command. But the command is even more limited than it appears. JOINS are not supported and WHERE conditions are almost always based only on equality constraints. Additional constraints often require the ALLOW FILTERING clause to indicate they might be slower queries. Data results are always limited to a specified number of rows. The default is 10,000 rows, but the value can be changed with the LIMIT clause.

The DBMSs, including Cassandra, are constantly being revised and updated, so limitations are likely to change; but the main limitations of the query system are due to the data storage model and the emphasis on performance. If an application needs more complex queries, it would probably be easier to move it to a relational DBMS or perhaps a data warehouse. The purpose of key-value pair non-relational DBMSs is to provide a fast, reliable way to store and retrieve individual pieces of data to millions of users.



A Developer's View

As more applications move to the Web, performance and continuous availability can become critical. Distributed databases can be significantly more responsive in this environment, but installing thousands of servers and copies of the DBMS software can be expensive. Open-source non-relational systems, such as Cassandra, are designed to handle these issues. Data is stored and retrieved as key-value pairs, which is fast for retrieving specific pieces of data. The tools do not support JOINS, referential integrity, or complex queries; so they are less useful for complex, interrelated business data. But there are times when you need a different tool to handle performance issues, and a DBMS like Cassandra can solve problems that are difficult and expensive to handle with a relational DBMS. However, the performance gains also arise through careful database design adjusted specifically for each application.

Key Terms

ad hoc queries	list
ALLOW FILTERING	map
Cassandra Query Language (CQL)	non-relational database
column family	NoSQL
columns	peer-to-peer
composite primary key	primary key
compound primary key	set
gossip	solid state drives (SSDs)
INSERT	sparse table
keyspace	tunable consistency
key-value pair	universally unique identifier (uuid)
LIMIT	UPDATE

Review Questions

1. How is a table in a key-value pair database different from one in a relational database?
-  2. How do highly-distributed databases create problems with data consistency?
3. What are the benefits and drawbacks to changing table columns over time?
4. What consistency problems can arise in a key-value pair database like Cassandra?
5. A database contains tables for Employee, Factory, and Assembly, where the Assembly table records which parts were installed by each employee at a specified time in each factory. Why would some of the employee data be stored in the Assembly rows?
-  6. What programming language is Cassandra written in and why does it matter?
7. What are the benefits and drawbacks to using uuids as primary key values?
8. Briefly explain what queries are supported by a table with a compound key of two columns, without adding indexes.
9. Why does Cassandra require indexes for some queries?
10. How is a composite key different from a compound key?

Exercises



1. Explain why you should avoid storing totals (such as inventory quantity on hand) in a key-value pair database and briefly describe an alternative to avoid totals.
2. An online site wants to hold medical health records for patients in a specific program. The records include basic patient data (name, birthdate, gender), and medical test results at various points in time that include levels for standard items such as Glucose, Potassium, and blood pressure. Design the tables you would use to store the data in Cassandra. Highlight the main queries.
3. Define the key-value-pair tables that would be needed for a Web site that sells custom shirts. Customers choose colors and sizes, and can enter text to be printed on the back, front, or sleeves.
4. Using Exercise 2 in Chapter 3 as a guide, define the key-value-pair tables needed for a Web site that lets individuals track their weight-lifting progress. Each session has multiple exercises (equipment), with different sets of weight levels, and a need to record the number of repetitions. For instance, a bench press might involve set1: 135 pounds, 10 reps, 185 pounds, 10 reps; and so on.
5. Looking at the previous exercise on weight lifting, explain how to change the design to support an application that shows the maximum weight lifted in an exercise over time (session). For example, the highest weight lifted in the bench press over the last year.
6. A Web site is built to access a database of music/songs (not classical music which has unique data elements). Users have the ability to rate each individual song. Define the key-value-pair tables needed for this site. Identify common queries that will be used and any indexes needed to support those queries.
7. Define the tables needed by Cassandra to build a Web site for a basketball league that records points scored by each team, the name of the referee, and the names of the players on each team. Similar to Chapter 3, Exercise 3.
8. Research Cassandra and briefly explain the difference between the timestamp and timeuuid data types.
9. Find a programming tool (not an online Web service) and generate 5 uuid values.
10. Find a different key-value-pair (NoSQL) DBMS and briefly compare it to Cassandra.



Sally's Pet Store


Using the Pet Store Web sample database for Cassandra, write the queries to answer the following questions.

11. Get the Description, ListPrice, and QOH for Item 5e9c2e10-3db1-4189-8c0d-0c700d421f17.

12. List all items in the Fish category? What indexes are needed?
-  13. List all items in the Dog category with a list price above \$20 and a QOH greater than 50. What indexes are needed?
14. Write a query that enables the system to e-mail the username/password to users who cannot remember what they entered, but do know their e-mail address and name. What indexes are needed?
15. List all of the comments with a rating of less than 3. What indexes are needed?
16. If the application needs to display only the first 10 (earliest) comments for a specific item (on the Web page), how should the design be modified and what is the new query?
17. How many comments have been made by the user with the screen name of Caged19? What indexes are needed?
-  18. Run the three UPDATE and INSERT queries in the text and issue a SELECT statement to see the ending values for the new CustomerID;
19. Change the list price on item ed7bb389-152c-4bdb-8546-4cd070fb4ae9 to \$10.
20. Add a new comment to the ItemComments table.



Rolling Thunder Bicycles

21. Rolling Thunder managers want to create a Web site to let owners upload photos of their custom bikes and let other users submit comments or questions; which can then be answered by other users (basically a discussion list). Define the tables needed for this application.
-  22. Assuming standard Web conventions, such as login by Username, what initial indexes will be needed for the discussion Web site in the previous question?
23. If Rolling Thunder managers decide to build the entire main ordering system as a Web site, would it be better to use a traditional relational DBMS or a key-value pair system like Cassandra? Explain your answer.



Corner Med

24. The managers of Corner Med want a Web site that handles communication between physicians and patients. They do not want to put all the patient visit data online, but want a secure site that enables patients to ask questions that are answered by physicians. What Cassandra tables would be needed for this site?
25. Assuming the solution to the previous question includes at least a Person and Discussion table, write the query to list all of the questions posted from a specific patient. What indexes would be needed?

Web Site References

http://cassandra.apache.org	Open-soure location of code.
http://www.datastax.com/docs	DataStax documentation site.
http://nosql-database.org/	List of NoSQL database projects.
http://planetcassandra.org/	Cassandra background (DataStax)
http://aws.amazon.com/nosql/	Amazon NoSQL options.
https://developers.google.com/appengine/	Google App Engine.

Additional Reading

<http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>. Interesting, readable comments on distributed transactions, scalability, and consistency as a reason for non-relational databases. (Brewer's CAP Theorem and references to other articles.)