2.1.5 Storing your application's structured data in a cloud database

Your car's navigation system provides constant updates about your current location and destination during your journey. It guides you through the route you need to take. This data, although critical for the trip, isn't useful afterward. The navigation system is to the car what a cloud database is to an application running in the cloud: it's transactional data created and used during the running of that application. When we think of transactional data stored in databases, we usually think of relational databases.

What is a Relational Database Management System (RDBMS)? Why do we frequently hear that they don't work in the cloud? An RDBMS is a database management system in which you store data in the form of tables; the relationship among the data is also stored in the form of tables. You can see this in the simple relation in figure 2.4.

RDBMS A database management system (DBMS) based on the relational model. *Relational* describes a broad class of database systems that at a minimum present the data to the user as relations (a presentation in tabular form—that is, as a collection of tables with each table consisting of a set of rows and columns—can satisfy this property) and provide relational operators to manipulate the data in tabular form. All modern commercial relational databases employ SQL as their query language, leading to a shorthand for RDBMSs as *SQL databases*.

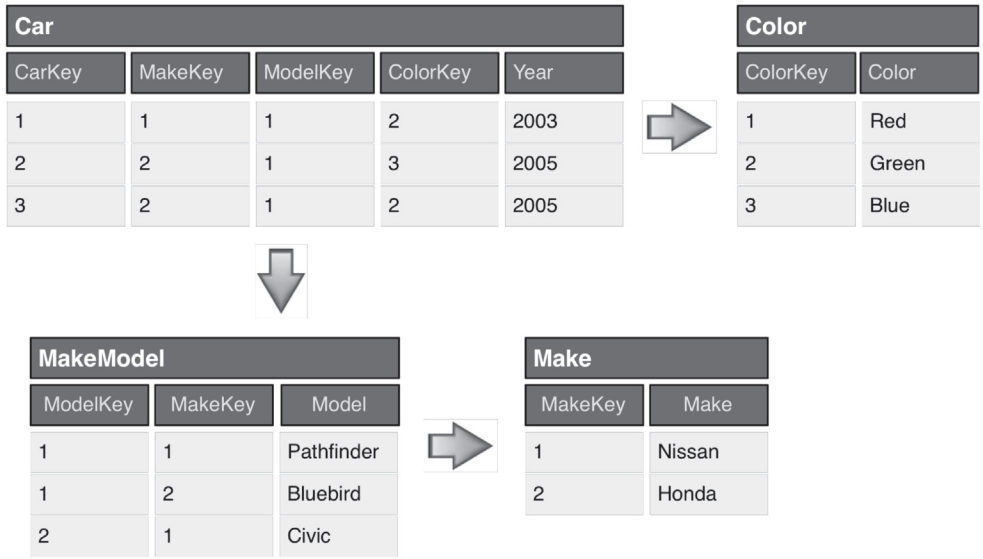

Figure 2.4 A simple example of how a relational database works. Four tables map out relationships among the data. Because a separate table lists the car manufacturers and colors, there is no need to separately list a red Nissan and a blue Nissan. But to fully understand what the car with CarKey 1 is, you must do a join of the Car, Color, MakeModel, and Make tables.

The challenge for an RDBMS in the cloud is scaling. Applications having a fixed number of users and workload requiring an RDBMS won't have any problems. Most cloud providers have an RDBMS offering for these cases. But when applications are launched in environments that have massive workloads, such as web services, their scalability requirements can change quickly and grow large. The first scenario can be difficult to manage if you have a relational database sitting on a single in-house server. For example, if your load triples overnight, how quickly can you upgrade your hardware? The second scenario can be too difficult to manage with a relational database, because it becomes a bottleneck choking the application's ability to scale. We'll cover solutions to this in depth in chapter 5.

As you've already learned, one of the core benefits of the cloud is the ability to quickly (or automatically, as we'll show) add more servers to an application as its load increases, thereby scaling it to heavier workloads. But it's hard to expand an RDBMS this way. You have to either replicate data across the new servers or partition between them. In either case, adding a machine requires data to be copied or moved to the new server. This data shipping is a time-consuming and expensive process, so databases are unable to be dynamically and efficiently provisioned on demand.

A big challenge with RDBMS partitioning or replicating is maintaining *referential integrity*. Referential integrity requires that every value of one attribute (column) of a relation (table) exist as a value of another attribute in a different (or the same) relation

(table). A little less formally, any field in a table that's declared a foreign key can contain only values from a parent table's primary key or a candidate key. In practice, this means that deleting a record that contains a value referred to by a foreign key in another table break's referential integrity. When you partition or replicate a database, it becomes nearly impossible to guarantee maintenance of referential integrity across all databases. This extremely useful property of RDBMS—its ability to construct a relation out of lots of small index tables that are referred to by values in records—becomes unworkable when these databases have to scale to deal with huge workloads, but cloud applications are otherwise ideally suited for this purpose.

THE NOSQL MOVEMENT

Since 1998, there has been a small but rapidly growing movement away from SQL databases. Instead, participants in this movement promote a class of nonrelational data stores that break some of the fundamental guarantees of SQL in favor of being able to reach massive scale. This is obviously important for *some* cloud applications. These non-SQL data stores may not require fixed table schemas and usually avoid join operations. They're described as scaling *horizontally*. Some categorize them as *structured storage*.

A new non-SQL type of database, generically a key-value database, does scale well. Consequently, it's started to be used in the cloud. Key-value databases are *item-oriented*, meaning all relevant data relating to an item are stored in that item. A table can

NoSQL architecture

Relational databases have a limitation on handling big data volumes and typical modern workloads. Today's scale is unprecedented and can't be handled with relational SQL databases. Examples of enormous scale are synonymous with the most popular sites: Digg's 3 TB for green badges, Facebook's 50 TB for inbox search, and eBay's 2 PB overall data.

NoSQL systems often provide weak consistency guarantees, such as *eventual* consistency and transactions restricted to single data items; in most cases, you can impose full ACID (atomicity, consistency, isolation, durability) guarantees by adding a supplementary middleware layer.

Several NoSQL systems employ a distributed architecture, with the data being held in a redundant manner on several servers, often using a distributed hash table. In this way, the system can be scaled up easily by adding more servers, and failure of a server can be tolerated.

Some NoSQL advocates promote simple interfaces, such as associative arrays or key-value pairs. Other systems, such as native XML databases, promote support of the XQuery standard.

Clearly, we're in the early days of cloud evolution, with a lot of development yet to come.

Car	
Key	Attributes
1	Make: Nissan Model: Pathfinder Color: Green Year: 2003
2	Make: Nissan Model: Pathfinder Color: Blue Year: 2005 Trans: Automatic

Figure 2.5 The same data as in figure 2.4, shown for a key-value type of database. Because all data for an item (row) is contained in that item, this type of database is trivial to scale because a data store can be split (by copying some of the items) or replicated (by copying all the items to an additional data store), and referential integrity is maintained.

contain vastly different items. For example, a table may contain car makes, car models, and car color items. This means data are commonly duplicated between items in a table (another item also contains Color: Green). You can see this in figure 2.5. In an RDBMS, this is anathema; here, this is accepted practice because disk space is relatively cheap. But this model allows a single item to contain all relevant data, which improves scalability by eliminating the need to join data from multiple tables. With a relational database, such data needs to be joined to be able to regroup relevant attributes. This is the key issue for scaling—if a join is needed that depends on shared tables, then replicating the data is hard and blocks easy scaling.

When companies set out to create a public computing cloud (such as Amazon) or build massively parallel, redundant, and economical data-driven applications (such as Google), relational databases became untenable. Both companies needed a way of managing data that was almost infinitely scalable, inherently reliable, and cost effective. Consequently, both came up with nonrelational database systems based on this key-value concept that can handle massive scale. Amazon calls its cloud database offering SimpleDB, and Google calls its BigTable. (Both were developed long before either company launched a cloud. They created these structures to solve their own problems. When they launched a cloud, the same structures became part of their cloud offerings.)

Google's BigTable solution was to develop a relatively simple storage management system that could provide fast access to petabytes of data, potentially redundantly distributed across thousands of machines. Physically, BigTable resembles a B-tree index-organized table in which branch and leaf nodes are distributed across multiple machines. Like a B-tree, nodes split as they grow, and—because nodes are distributed—this allows for high scalability across large numbers of machines. Data elements in BigTable are identified by a primary key, column name, and, optionally, a timestamp. Lookups via primary key are predictable and relatively fast. BigTable provides the data storage mechanism for Google App Engine. You'll learn about this PaaS cloud-based application environment in detail later in this chapter.

Google charges $180 per terabyte per month for BigTable storage. Here are some examples of BigTable usage (in Python):

This code declares a data store class:

```
class Patient(db.Modal);
    firstName = db.UserProperty()
    lastName = db.UserProperty()
    dateOfBirth = db.DateTimeProperty()
    sex = db.UserProperty()
```

This code creates and stores an object:

```
patient = Patient()

patient.firstName = "George"
patient.lastName = "James"
dateOfBirth = "2008-01-01"
sex = "M"

patient.put()
```

This code queries a class:

```
patients = Patient.all()

for patient in patients:
    self.response.out.write('Name %s %s. ',
            patient.firstName, patient.lastName)
```

And this code selects the 100 youngest male patients:

```
allPatients = Patient.all()
allPatients.filter('sex=', 'Male')
allPatients.order('dateOfBirth')
patients = allPatients.fetch(100)
```

Amazon's SimpleDB is conceptually similar to BigTable and forms a key part of the Amazon Web Services (AWS) cloud computing environment. (Microsoft's SQL Server Data Services [SSDS] provides a similar capability in their Azure cloud.) Like BigTable, this is a key-value type of database. The basic organizing entity is a *domain*. Domains are collections of items that are described by attribute-value pairs. You can see an abbreviated list of the SimpleDB API calls with their functional description in table 2.4.

Table 2.4 Amazon's SimpleDB API summary

API call	API functional description
`CreateDomain`	Creates a domain that contains your dataset.
`DeleteDomain`	Deletes a domain.
`ListDomains`	Lists all domains.
`DomainMetadata`	Retrieves information about creation time for the domain, storage information both as counts of item names and attributes, and total size in bytes.
`PutAttributes`	Adds or updates an item and its attributes, or adds attribute-value pairs to items that exist already. Items are automatically indexed as they're received.
`BatchPutAttributes`	For greater overall throughput of bulk writes, performs up to 25 `PutAttribute` operations in a single call.

Table 2.4 Amazon's SimpleDB API summary (*continued*)

API call	API functional description
`DeleteAttributes`	Deletes an item, an attribute, or an attribute value.
`GetAttributes`	Retrieves an item and all or a subset of its attributes and values.
`Select`	Queries the data set in the familiar "Select target from *domain_name* where *query_expression*" syntax. Supported value tests are `=`, `!=`, `<`, `>`, `<=`, `>=`, `like`, `not like`, `between`, `is null`, `isn't null`, and `every()`. Example: `select * from mydomain where every(keyword) = "Book"`. Orders results using the `SORT` operator, and counts items that meet the condition(s) specified by the predicate(s) in a query using the `Count` operator.

Converting an existing application to use one of these cloud-based databases is somewhere between difficult and not worth the trouble; but for applications already using the Object-Relational Mapping (ORM)-based frameworks, these cloud databases can easily provide core data-management functionality. They can do it with compelling scalability and the same economic benefits of cloud computing in general. But as table 2.5 illustrates, there are definite drawbacks to these new types of cloud databases that you must take into account when contemplating a shift to the cloud.

Table 2.5 Cloud database drawbacks

Database use	Challenges faced with a cloud database
Transactional support and referential integrity	Applications using cloud databases are largely responsible for maintaining the integrity of transactions and relationships between tables.
Complex data access	Cloud databases (and ORM in general) excel at single-row transactions: get a row, save a row, and so on. But most nontrivial applications have to perform joins and other operations that cloud databases can't.
Business Intelligence	Application data has value not only in terms of powering applications but also as information that drives business intelligence. The dilemma of the pre-relational database, in which valuable business data was locked inside impenetrable application data stores, isn't something to which business will willingly return.

Cloud databases could displace the relational database for a significant segment of next-generation, cloud-enabled applications. But business is unlikely to be enthusiastic about an architecture that prevents application data from being used for business intelligence and decision-support purposes, which fundamentally require a relational database. An architecture that delivered the scalability and other advantages of cloud databases without sacrificing information management would fit the bill. We can expect a lot of innovation and advancements in these database models over the next few years.