3
SimpleDB versus RDBMS

We have all used a **Relational Database Management System** (**RDBMS**) at some point in our careers. These relational databases are ubiquitous and are available from a wide range of companies such as Oracle, Microsoft, IBM, and so on. These databases have served us well for our application needs. However, there is a new breed of applications coming to the forefront in the current Internet-driven and socially networked economy. The new applications require large scaling to meet demand peaks that can quickly reach massive levels. This is a scenario that is hard to satisfy using a traditional relational database, as it is impossible to requisition and provision the hardware and software resources that will be needed to service the demand peaks. It is also non-trivial and difficult to scale a normal RDBMS to hundreds or thousands of nodes. The overwhelming complexity of doing this makes the RDBMS not viable for these kinds of applications. SimpleDB provides a great alternative to an RDBMS and can provide a solution to all these problems. However, in order to provide this solution, SimpleDB makes some choices and design decisions that you need to understand in order to make an informed choice about the data storage for your application domain.

In this chapter, we are going to discuss the differences between SimpleDB and a traditional RDBMS, as well as the pros and cons of using SimpleDB as the storage engine in your application.

No normalization

Normalization is a process of organizing data efficiently in a relational database by eliminating redundant data, while at the same time ensuring that the data dependencies make sense. SimpleDB data models do not conform to any of the normalization forms, and tend to be completely de-normalized. The lack of need for normalization in SimpleDB allows you a great deal of flexibility with your model, and enables you to use the power of multi-valued attributes in your data.

Let's look at a simple example of a database starting with a basic spreadsheet structure and then design it for an RDBMS and a SimpleDB. In this example, we will create a simple contact database, with contact information as raw data.

ID	First_Name	Last_Name	Phone_Num
101	John	Smith	555-845-7854
101	John	Smith	555-854-9885
101	John	Smith	555-695-7485
102	Bill	Jones	555-748-7854
102	Bill	Jones	555-874-8654

The obvious issue is the repetition of the name data. The table is inefficient and would require care to update to keep the name data in sync. To find a person by his or her phone number is easy.

```
SELECT * FROM Contact_Info WHERE Phone_Num = '555-854-9885'
```

So let's analyze the strengths and weaknesses of this database design.

SCORE—Raw data	Strength	Weakness
Efficient storage		No
Efficient search by phone number	Yes	
Efficient search by name		No
Easy to add another phone number	Yes	

The design is simple, but as the name data is repeated, it would require care to keep the data in sync. Searching for phone numbers by name would be ugly if the names got out of sync.

To improve the design, we can rationalize the data. One approach would be to create multiple phone number fields such as the following. While this is a simple solution, it does limit the phone numbers to three. Add e-mail and Twitter, and the table becomes wider and wider.

ID	First_Name	Last_Name	Phone_Num_1	Phone_Num_2	Phone_Num_3
101	John	Smith	555-845-7854	555-854-9885	555-695-7485
102	Bill	Jones	555-748-7854	555-874-8654	

Finding a person by a phone number is ugly.

```
SELECT * FROM Contact_Info WHERE Phone_Num_1 = '555-854-9885'
OR Phone_Num_2 = '555-854-9885'
OR Phone_Num_3 = '555-854-9885'
```

Now let's analyze the strengths and weaknesses of this database design.

SCORE—Rationalize data	Strength	Weakness
Efficient storage	Yes	
Efficient search by phone number		No
Efficient search by name	Yes	
Easy to add another phone number		No

The design is simple, but the phone numbers are limited to three, and searching by phone number involves three index searches.

Another approach would be to use a delimited list for the phone number as follows:

ID	First_Name	Last_Name	Phone_Nums
101	John	Smith	555-845-7854;555-854-9885;555-695-7485
102	Bill	Jones	555-748-7854;555-874-8654

This approach has the advantage of no data repetition and is easy to maintain, compact, and extendable, but the only way to find a record by the phone number is with a substring search.

```
SELECT * FROM Contact_Info WHERE Phone_Nums LIKE %555-854-9885%
```

This type of SQL forces a complete table scan. Do this with a small table and no one will notice, but try this on a large database with millions of records, and the performance of the database will suffer.

SCORE—Delimited data	Strength	Weakness
Efficient storage	Yes	
Efficient search by phone number		No
Efficient search by name	Yes	
Easy to add another phone number	Yes	

A delimited field is good for data that is of one type and will only be retrieved.

The **normalization** for relational databases results in splitting up your data into separate tables that are related to one another by keys. A **join** is an operation that allows you to retrieve the data back easily across the multiple tables.

Let's first normalize this data.

This is the `Person_Info` table:

ID	First_Name	Last_Name
101	John	Smith
102	Bill	Jones

And this is the `Phone_Info` table:

ID	Phone_Num
101	555-845-7854
101	555-854-9885
101	555-695-7485
102	555-748-7854
102	555-874-8654

Now a join of the `Person_Info` table with the `Phone_Info` can retrieve the list of phone numbers as well as the e-mail addresses. The table structure is clean and other than the ID primary key, no data is duplicated. Provided `Phone_Num` is indexed, retrieving a contact by the phone number is efficient.

```
SELECT First_Name, Last_Name, Phone_num, Person_Info.ID
FROM Person_Info JOIN Phone_Info
ON Person_Info.ID = Phone_Info.ID
WHERE Phone_Num = '555-854-9885'
```

So if we analyze the strengths and weaknesses of this database design, we get:

SCORE—Relational data	Strength	Weakness
Efficient storage	Yes	
Efficient search by phone number	Yes	
Efficient search by name	Yes	
Easy to add another phone number	Yes	

While this is an efficient relational model, there is no `join` command in SimpleDB. Using two tables would force two selects to retrieve the complete contact information. Let's look at how this would be done using the SimpleDB principles.

No joins

SimpleDB does not support the concept of joins. Instead, SimpleDB provides you with the ability to store multiple values for an attribute, thus avoiding the necessity to perform a join to retrieve all the values.

ID			
101	First_Name=John	Last_Name=Smith	Phone_Num = 555-845-7854 Phone_Num = 555-854-9885 Phone_Num = 555-695-7485
102	First_Name=Bill	Last_Name=Jones	Phone_Num = 555-748-7854 Phone_Num = 555-874-8654

In the SimpleDB table, each record is stored as an item with attribute/value pairs. The difference here is that the `Phone_Num` field has multiple values. Unlike a delimited list field, SimpleDB indexes all values enabling an efficient search each value.

```
SELECT * FROM Contact_Info WHERE Phone_Num = '555-854-9885'
```

This `SELECT` is very quick and efficient. It is even possible to use `Phone_Num` multiple times such as follows:

```
SELECT * FROM Contact_Info WHERE Phone_Num = '555-854-9885'
OR Phone_Num = '555-748-7854'
```

Let's analyze the strengths and weaknesses of this approach:

SCORE—SimpleDB data	**Strength**	**Weakness**
Efficient storage	Yes	
Efficient search by phone number	Yes	
Efficient search by name	Yes	
Easy to add another phone number	Yes	

No schemas

There are no schemas anywhere in sight of SimpleDB. You don't have to create schemas, change schemas, migrate schemas to a new version, or maintain schemas. This is yet another thing that is difficult for some people from a traditional relational database world to grasp, but this flexibility is one of the keys to the power of scaling offered by SimpleDB. You can store any attribute-value data you like in any way you want. If the requirements for your application should suddenly change and you need to start storing data on a customer's Twitter handle for instance, all you need to do is store the data without worrying about any schema changes!

Let's add an e-mail address to the database in the previous example. In the relational database, it is necessary to either add e-mail to the phone table with a type of contact field or add another field. Let's add another table named `Email_Info`.

`Person_Info` table:

ID	First_Name	Last_Name
101	John	Smith
102	Bill	Jones

`Phone_Info` table:

ID	Phone_Num
101	555-845-7854
101	555-854-9885
101	555-695-7485
102	555-748-7854
102	555-874-8654

`Email_Info` table:

ID	Email_Addr
101	`john@abc.ccc`
102	`bill@def.ccc`

Using a traditional relational database approach, we join the three tables to extract the requested data in one call.

```
SELECT First_Name, Last_Name, Phone_num, Person_Info.ID, Email_Addr
FROM Person_Info JOIN Phone_Info JOIN Email_Info
ON Person_Info.ID = Phone_Info.ID
AND Person_Info.ID = Email_Info.ID
WHERE Phone_Num = '555-854-9885'
```

Now let's analyze the strengths and weaknesses of this approach:

SCORE—Relational data	**Strength**	**Weakness**
Efficient storage	Yes	
Efficient search by phone number, email	Yes	
Efficient search by name	Yes	
Easy to add another phone number	Yes	
Expandable	Yes	New table defined Two joins required

We ignored the issue of join versus **left outer join**, which is really what should be used here unless all contacts have a phone number and e-mail address. The example is just to illustrate that the `Contact_Info` schema must be modified.

`Contact_Info` domain:

ID			
101	First_Name = John	Last_Name = Smith	Phone_Num = 555-845-7854 Phone_Num = 555-854-9885 Phone_Num = 555-695-7485 Email_Addr = john@abc.ccc
102	First_Name = Bill	Last_Name = Jones	Phone_Num = 555-748-7854 Phone_Num = 555-874-8654 Email_Addr = john@def.ccc

The obvious question is why is `Email_Addr` not in its own column? In SimpleDB, there is no concept of a column in a table. The spreadsheet view of the SimpleDB data was done for ease of readability, not because it reflects the data structure. The only structure in SimpleDB consists of the item name and attribute/value pairs. The proper representation of the SimpleDB data is:

ID	Attribute/Value pairs
101	First_Name = John Last_Name = Smith Phone_Num = 555-845-7854 Phone_Num = 555-854-9885 Phone_Num = 555-695-7485 Email_Addr = john@abc.ccc
102	First_Name = Bill Last_Name = Jones Phone_Num = 555-748-7854 Phone_Num = 555-874-8654 Email_Addr = john@def.ccc

Use the following query to fetch a contact item by the e-mail address:

```
SELECT * FROM Contact_Info WHERE Email_Addr = 'john@def.ccc'
```

Let's analyze the strengths and weaknesses of this approach:

SCORE—SimpleDB data	Strength	Weakness
Efficient storage	Yes	
Efficient search by phone number, email	Yes	
Efficient search by name	Yes	
Easy to add another phone number	Yes	
Expandable	Yes	

Simpler SQL

Structured Query Language (**SQL**) is a standard language that is widely used for accessing and manipulating the data stored in a relational database. SQL has evolved over the years into a highly complex language that can do a vast variety of things to your database. SimpleDB does not support the complete SQL language, but instead it lets you perform your data retrieval using a much smaller and simpler subset of an SQL-like query language. This simplifies the whole process of querying your data. A big difference between the simpler SQL supported by SimpleDB and SQL is the support for multi-valued SimpleDB attributes, which makes it super simple to query your data and get back multiple values for an attribute.

The syntax of the SimpleDB SQL is summarized in this syntax:

```
select output_list
from domain_name
[where expression]
[sort_instructions]
[limit limit]
```

We will go into detail on SimpleDB SQL in *Chapter 6*, *Querying*.

Only strings

SimpleDB uses a very simple data model, and all data is stored as an UTF-8 string. This simplified textual data makes it easy for SimpleDB to automatically index your data and give you the ability to retrieve the data very quickly. If you need to store and retrieve other kinds of data types such as numbers and dates, you must encode these data types into strings whose lexicographical ordering will be the same as your intended ordering of the data. As SimpleDB does not have the concept of schemas that enforce type correctness for your domains, it is the developer's responsibility to ensure the correct encoding of data before storage into SimpleDB.

Working only in strings impacts two aspects of using the database: queries and sorts.

Consider the following `Sample_Qty` table:

ID	
101	Quantity = 1.0
102	Quantity = 1.00
103	Quantity = 10
104	Quantity = 25
105	Quantity = 100

Now try and execute the following SQL statement:

```
SELECT * FROM Sample_Qty WHERE Quantity= '1'
```

This SQL statement will retrieve nothing—not even items 101 and 102.

Selecting all records sorted by `Quantity` will return the order 101, 102, 103, 105, 104.

Dates present an easier problem, as they can be stored in ISO 8601 format to enable sorting as well as predictable searching. We will cover this in detail in *Chapter 5*, *Data Types*.

Eventual consistency

Simple DB can be thought of as a Write-Seldom-Read-Many model. Updates are done to a central database, but reads can be done from many read-only database slave servers.

SimpleDB keeps multiple copies of each domain. Whenever data is written or updated within a domain, first a success status code is returned to your application, and then all the different copies of the data are updated. The propagation of these changes to all of the nodes at all the storage locations might take some time, but eventually the data will become consistent across all the nodes.

SimpleDB provides this assurance only of eventual consistency for your data. This means that the data you retrieve from SimpleDB at any particular time may be slightly out of date. The main reason for this is that SimpleDB service is implemented as a distributed system, and all of the information is stored across multiple physical servers and potentially across multiple data centers in a completely redundant manner. This ensures the large-scale ready accessibility and safety of your data, but comes at the cost of a slight delay before any addition, alteration, or deletion operations you perform on the data being propagated throughout the entire distributed SimpleDB system. Your data will eventually be globally consistent, but until it is consistent, the possibility of retrieving slightly outdated information from SimpleDB exists.

Amazon has stated in the past that states of global consistency across all the nodes will usually be achieved "within seconds"; however, please be aware that this timeframe will depend to a great degree on the processing and the network load on SimpleDB at the time that you make a change to your data. An intermediate caching layer can quickly solve this consistency issue if data consistency is highly important and essential to your application. The principle of eventual consistency is the hardest to grasp, and it is the biggest difference between a RDBMS and SimpleDB. In order to scale massively, this is a trade-off that needs to be made at design time for your application. If you consider how often you will require immediate consistency within your web applications, you might find that this trade-off is well worth the improved scalability of your application.

Flash: February 24, 2010 — consistent read added

While eventual consistency is still the normal mode for SimpleDB, Amazon announced several extensions for consistent read. When using a `GetAttributes` or `SELECT`, the `ConsistentRead = true` can be selected, forcing a read of the most current value. This tells SimpleDB to read the items from the master database rather than from one of the slaves, guaranteeing the latest updates or deletes. This does not mean you can use this on all reads and still get the extreme scaling. In *Chapter 8*, *Tuning and Usage Costs*, we will look at the cost of using consistent reads.

A conditional PUT or DELETE was also announced, which will execute a database PUT or DELETE only if the consistent read of a specific attribute has a specific value or does not exist. This is useful if concurrent controls or counters primitives. In later chapters, we will look at the implications of these new features.

Scalability

Relational databases are designed around the entities and the relationships between the entities, and need a large investment in hardware and servers in order to provide high scaling. SimpleDB provides a great alternative that is designed around partitioning your data into independent chunks that are stored in a distributed manner and can scale up massively. SimpleDB provides the automatic partitioning and replication of your data, while at the same time guaranteeing fast access and reliability for your data. You can let Amazon scale their platform as needed using their extensive resources, while you enjoy the ability to easily scale up in response to increased demand!

The best feature of SimpleDB scalability is that you only pay for usage, not for the large cluster needed in anticipation of large usage.

Low maintenance

Maintaining a relational database and keeping it humming with indexing takes effort, know-how, and technical and administrative resources. Applications are not static but dynamic things, and change constantly along with additions of new features. All of these updates can result in changes and modifications to the database schema along with increased maintenance and tuning costs. SimpleDB is hosted and maintained for you by Amazon. Your task is as simple as storing your data and retrieving it when needed. The simplicity of structured data and lack of schemas helps your application be more flexible and adaptable to change, which is always around the corner. SimpleDB ensures that your queries are optimized and retrieval times are fast by indexing all your data automatically.

Advantages of the SimpleDB model

SimpleDB's alternative approach for storing data can be advantageous for meeting your application needs when compared to a traditional relational database. Here's the list of advantages:

- Reduced maintenance as compared to a relational database
- Automated indexing of your data for fast performance
- Flexibility to modify or change your stored data without the need to worry about schemas
- Failover for your data automatically being provided by Amazon
- Replication for your data across multiple nodes also handled for you by Amazon
- Ability to easily scale up in response to increased demand without worrying about running out of hardware or processing capacity
- Simplified data storage and querying using a simple API
- The lack of object-to-relational mapping that is common for an RDBMS allows your structured data to map more directly to your underlying application code and reduce the application development time

Disadvantages of the SimpleDB model

SimpleDB's alternative approach also has some disadvantages compared to a relational database for certain applications.

- Those using applications that always need to ensure immediate consistency of data will find that SimpleDB's eventual data consistency model may not suit their needs. The consistent read announcement does change this, but the eventual consistency model is still the basis of the extreme scalability.
- Using SimpleDB as the data storage engine in your applications needs the development team to get used to different concepts over a simple, traditional RDBMS.
- Because relationships are not explicitly defined at the schema level as in a relational database, you might need to enforce some data constraints within your application code.
- If your application needs to store data other than strings, such as numbers and dates, additional effort will be required on your part to encode the strings before storing them in the SimpleDB format.
- The ability to have multiple attributes for an item is a completely different way of storing data and has a learning curve attached to it for new users who are exposed to SimpleDB.

Summary

In this chapter, we discussed the differences between SimpleDB and the traditional relational database systems in detail. In the next chapter, we are going to review the data model used by SimpleDB.