developerWorks®

Java development 2.0: Cloud storage with Amazon's SimpleDB, Part 1

Get started with SimpleDB and the Amazon SDK

Andrew Glover
Author and developer
Beacon50

15 June 2010

Part of the Amazon Web Services family, Amazon's SimpleDB is a massively scalable and reliable key/value datastore that is exposed via a web interface and can be accessed using the Java language. Get started with Amazon's SimpleDB in this first of two articles exploring SimpleDB's unique approach to schemaless data storage, including a demonstration of one of the datastore's most unusual features: lexicographic searching.

View more content in this series

About this series

The Java development landscape has changed radically since Java technology first emerged. Thanks to mature open source frameworks and reliable for-rent deployment infrastructures, it's now possible to assemble, test, run, and maintain Java applications quickly and inexpensively. In this series, Andrew Glover explores the spectrum of technologies and tools that make this new Java development paradigm possible.

Throughout this series, I've shared with you a number of non-relational data stores, collectively dubbed NoSQL. In a recent article I showed you how a document-oriented datastore (CouchDB) differs vastly from schema-oriented relational databases. What's more, CouchDB's entire API is RESTful and supports a different means of querying: MapReduce functions defined in JavaScript. Obviously, this is a break from the traditional world of JDBC.

I have also recently written about Google's Bigtable, which isn't a relational *or* document-oriented data solution (and, incidentally, doesn't support JDBC in any way). Bigtable is what's known as a key/value store. That is, it's *schemaless* and allows you to store basically anything you'd like whether that be an instance of a parking ticket, a list of races, or even the runners in a race. Bigtable's lack of a schema offers a tremendous amount of flexibility, which supports rapid development.

Video demo: An introduction to Amazon SimpleDB

Follow along as Andrew Glover guides you through an introduction to SimpleDB, a massively scalable, highly available key/value datastore. Part of the Amazon Web Services suite, SimpleDB provides a simple web services interface to create and store multiple data sets, query the data, and return the results.

Bigtable isn't the only key/value datastore we have to choose from. Amazon has its own cloud-based key/value store dubbed Amazon SimpleDB. While Bigtable is exposed to Java developers through an abstraction facilitated by the Google App Engine, Amazon's SimpleDB is exposed via a web service interface. Thus, you can manipulate the SimpleDB datastore via the web and HTTP. Bindings on top of Amazon's Web Service infrastructure make it possible to leverage SimpleDB using the language of your choice, with options that include PHP, Ruby, C#, and the Java language.

This month, I'll introduce you to SimpleDB by way of Amazon's official SDK. I'll use another back-to-the-races example to demonstrate one of the more unusual facets of this powerful, cloud-based datastore: lexicographic searching.

Introducing SimpleDB

Develop skills on this topic

This content is part of a progressive knowledge path for advancing your skills. See Using NoSQL and analyzing big data

Under the hood, SimpleDB is a massively scalable, highly available datastore written in Erlang. Conceptually, it's like Amazon's S3. Whereas S3 has objects located in buckets, SimpleDB is logically defined as domains containing items. SimpleDB also permits items to contain attributes. Think of a *domain* much like you would a bucket in S3, or a table in the relational sense (or more appropriately, Bigtable's "kind" notion). But be careful not to project relational-ness into your concept of SimpleDB, because it's ultimately just as schemaless as Bigtable. Domains can have many items (which are similar to rows) and items can have many attributes (which are like columns in the relational world).

SimpleDB's 'eventual consistency'

The *CAP theorem* (see Resources) states that a distributed system can't be highly available, scalable, and guarantee consistency all at once; instead, a distributed system can only support two of these three qualities at any given time. Accordingly, SimpleDB guarantees a highly available and scalable datastore that doesn't support immediate consistency. What SimpleDB does support is *eventual consistency*, which isn't as bad as you might think.

In Amazon's case, eventual consistency means that things become consistent across all nodes (albeit within a region) within *seconds*. What you trade for that mere sliver of time where two concurrent processes may (just may) read two differing instances of the same data is massive reliability and at a very affordable price. (You only need to price out commercial entities offering similar reliability to see the difference.)

Attributes are really just name/value pairs (sounds like Bigtable, no?) and the "pair" aspect isn't limited to one value. That is, an attribute name can have a collection (or list) of associated values; a word item could, for example, have multiple-definition attribute values. What's more, all data within SimpleDB is represented as a `String`, which is distinctly different from Bigtable or even a standard RDBMS, which typically support myriad datatypes.

SimpleDB's single-datatype approach to attribute values could be a benefit or a limitation, depending on how you look at it. Either way, it does have implications for how queries are run (more about that shortly). SimpleDB also doesn't support the notion of cross-domain joins, so you can't query for items in multiple domains. You could, however, overcome this limitation by performing multiple SimpleDB queries and doing the join on your end.

Items don't have keys *per se* (like Bigtable does). The key or unique identifier for an item is the item's name. SimpleDB is smart enough to update an item when a duplicate creation request is issued, too, provided the attributes of that item have been altered.

Like other Amazon Web Services, SimpleDB exposes everything via HTTP, so there are myriad ways to interface with it. In the Java world, our options range from Amazon's own SDK (which we'll use in the examples that follow) to a popular project called Topica to even full-blown JPA implementations (which we'll explore in Part 2).

Racing in the clouds

So far in this series, I've used a race and a parking-ticket analogy to demonstrate the features of various Java 2.0 technologies. Using a familiar problem domain makes it easier to to appreciate the differences and commonalities between systems. So we'll stick with the finish-line analogy this time and see how runners and races are represented in Amazon's SimpleDB.

In SimpleDB, we could model a race as a domain. The race instance would be an item in SimpleDB and its name and date would be expressed as attributes (with values). It's important to note that the *name* in this case is an attribute and not the name of the item itself. The name you give to an item instance becomes its key. The key for this item could be the marathon's name. Alternately, rather than limit a race instance to one point in time (races are often annual events), we could give the item a unique name (like a timestamp), which would allow us to store multiple biannual races in SimpleDB.

Likewise, `runner` would be a domain. Individual runners would be items, and a runner's name and age would be attributes. Just like with a race, each `runner` item instance would need a unique name (distinguishing Pete Smith from Marty Howard, for instance). Unlike Bigtable, SimpleDB doesn't care what you name each item and in fact, it doesn't provide you with a key generator. Perhaps in this case, we could use a timestamp or just increment a counter for each runner, such as `runner_1`, `runner_2`, etc.

Because there is no schema, individual items are free to vary their attributes. Likewise, you can vary items in a domain if you want to. You'll want to limit this variability, however, as it tends to make data unorganized, and thus not easy to find or manage. Heed my words here: Willy-nilly, no-schema, non-organized data is a recipe for disaster!

Off and running with Amazon SDK

Amazon recently standardized a library containing code for working with all of its web services, including SimpleDB. This library, like most, abstracts the underlying communication required to access and use these services, enabling clients to work in a native way. For instance, Amazon's Java library for SimpleDB allows you to create domains and items, query for them, and of course update and remove them from storage — all the while blissfully unaware of these operations traveling over HTTP into the cloud.

Listing 1 shows an `AmazonSimpleDBClient` defined using plain-Jane Java code, along with a `Races` domain. (You'll need to create an account with Amazon if you want to duplicate this exercise on your workstation.)

Listing 1. Creating an instance of AmazonSimpleDBClient

```
AmazonSimpleDB sdb = new AmazonSimpleDBClient(new PropertiesCredentials(
                new File("etc/AwsCredentials.properties")));
String domain = "Races";
sdb.createDomain(new CreateDomainRequest(domain));
```

Note that Amazon SDK's pattern of `Request` objects will remain for all SimpleDB activities. In this case, creating a `CreateDomainRequest` creates a domain. I can add items via the client's `batchPutAttributes` method, which essentially takes a `List` of items like the ones shown in Listing 2:

Listing 2. Race_01

```
List<ReplaceableItem> data = new ArrayList<ReplaceableItem>();

data.add(new ReplaceableItem().withName("Race_01").withAttributes(
   new ReplaceableAttribute().withName("Name").withValue("Charlottesville Marathon"),
   new ReplaceableAttribute().withName("Distance").withValue("26.2")));
```

In Amazon's SDK, `Item`s are represented as `ReplaceableItem` types. You give each instance a name (that is, a key) and then you can add attributes (of `ReplaceableAttribute` type). In Listing 2, I've created a race, a marathon with the simple key, "`Race_01`". I add this instance to my `Races` domain by creating a `BatchPutAttributesRequset` and sending that along to the `AmazonSimpleDBClient`, as shown in Listing 3:

Listing 3. Creating an Item in SimpleDB

```
sdb.batchPutAttributes(new BatchPutAttributesRequest(domain, data));
```

Queries in SimpleDB

Now that I've got a race saved, I can, of course, search for it via SimpleDB's query language, which looks a lot like SQL. There's one catch, though. Do you remember when I said all item attribute values are stored as `String`s? This means data comparisons are done *lexicographically*, which has repercussions when it comes to searches.

If I run a query based on numbers, for example, SimpleDB will search based on characters, and not actual integer values. Right now, I've got a single race instance stored in SimpleDB, and I can search for it easily using SimpleDB's SQL-like statements, shown in Listing 4:

Listing 4. Searching for Race_01

```
String qry = "select * from `" + domain + "` where Name = 'Charlottesville Marathon'";
SelectRequest selectRequest = new SelectRequest(qry);
for (Item item : sdb.select(selectRequest).getItems()) {
 System.out.println("Race Name: " + item.getName());
}
```

The query in Listing 4 looks like normal SQL. In this case, I'm simply asking for all instances from `Race` where the `Name` is equal to "Charlottesville Marathon." Sending a `SelectRequest` to the `AmazonSimpleDBClient` yields a collection of `Item`s as a result. Thus, I am able to iterate over the items and print their names, and in this case I should only receive one item back.

But now let's see what happens when I add another race with a different distance attribute, shown in Listing 5:

Listing 5. A shorter race

```
List<ReplaceableItem> data2 = new ArrayList<ReplaceableItem>();

data2.add(new ReplaceableItem().withName("Race_02").withAttributes(
   new ReplaceableAttribute().withName("Name").withValue("Charlottesville 1/2 Marathon"),
   new ReplaceableAttribute().withName("Distance").withValue("13.1")));

sdb.batchPutAttributes(new BatchPutAttributesRequest(domain, data2));
```

With two races of difference distances, it makes sense to search based on distance, as shown in Listing 6:

Listing 6. Searching by distance

```
String disQry = "select * from `" + domain + "` where Distance > '13.1'";
SelectRequest selectRequest = new SelectRequest(disQry);
for (Item item : sdb.select(selectRequest).getItems()) {
 System.out.println("Race Name: " + item.getName());
}
```

Sure enough, the race that comes back is `Race_01`, which is correct: 26.2 is greater than 13.1, both mathematically *and* lexicographically. But watch what happens when I add a *really* long race to the mix:

Listing 7. Leesburg Ultra Marathon

```
List<ReplaceableItem> data3 = new ArrayList<ReplaceableItem>();

data3.add(new ReplaceableItem().withName("Race_03").withAttributes(
   new ReplaceableAttribute().withName("Name").withValue("Leesburg Ultra Marathon"),
   new ReplaceableAttribute().withName("Distance").withValue("103.1")));

sdb.batchPutAttributes(new BatchPutAttributesRequest(domain, data3));
```

In Listing 7, I've added a race with a distance of 103.1. When I rerun the query from Listing 6, guess what's there? Yes, it's true: 103.1, lexicographically speaking, is less than 13.1, not greater. That's why (if you're following along at home) you don't see the Leesburg Ultra Marathon listed!

Now let's see what happens if I run a different query looking for shorter races, as shown in Listing 8:

Listing 8. Let's see what shows up!

```
String disQry = "select * from `" + domain + "` where Distance < '13.1'";
SelectRequest selectRequest = new SelectRequest(disQry);
for (Item item : sdb.select(selectRequest).getItems()) {
 System.out.println("Race Name: " + item.getName());
}
```

To the unsuspecting eye, running the query in Listing 8 will yield a surprising result. Knowing that searches are performed lexicographically, however, it makes total sense — even though if you are looking for a short race, the (fictional) Leesburg Ultra Marathon isn't your bag!

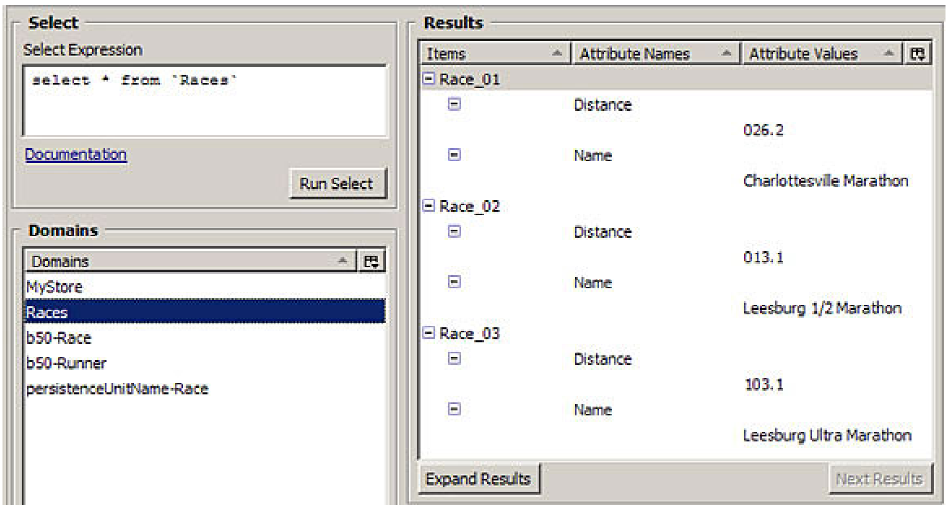
Lexicographic searching

Lexicographic searching can cause issues when looking for numbered data (including dates), but all is not lost. One way to fix the searching-on-distance issue is by padding the numbers used for distance attributes.

My longest race currently is 103.6 miles (although I've personally never come close to running this distance!), which reads lexicographically as three digits to the left of the decimal point. So I'll just pad the remaining races with leading zeros, giving all races the same number of characters. Doing this will make my distance-based searches work.

Figure 1 is a screenshot from SDB Tool, a Firefox plug-in for visually querying and updating Simple DB database domains (see Resources):

Figure 1. Padding the distance values



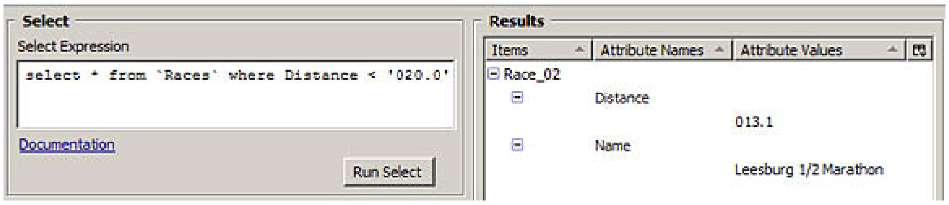
```
select * from `Races`
```

Domains
MyStore
Races
b50-Race
b50-Runner
persistenceUnitName-Race

Items	Attribute Names	Attribute Values
Race_01		
	Distance	
		026.2
	Name	
		Charlottesville Marathon
Race_02		
	Distance	
		013.1
	Name	
		Leesburg 1/2 Marathon
Race_03		
	Distance	
		103.1
	Name	
		Leesburg Ultra Marathon

As you can see, I've added a zero to both `Race_01` and `Race_02`'s distance values. While this might not make a lot of sense to the untrained eye, it'll make searching a lot easier. Thus, in Figure 2, you can see that I've issued a search for races with a distance less than 020.0 miles (or just plain 20 miles), and look what finally — and correctly — appears:

Figure 2. Padded searching solves the problem



```
select * from `Races` where Distance < '020.0'
```

```
Race_02
    Distance
        013.1
    Name
        Leesburg 1/2 Marathon
```

With a little foresight, it's not hard to overcome what might seem like a limiting factor of lexicographic searches. If padding isn't your bag, another option is to filter on the application side of things. That is, you could keep your integers as normal integers and filter things once you've got a collection of non-filtered items from Amazon — that is, issuing a `select *` on all items. This approach could be costly though, if you have a lot data.

Relationships in SimpleDB

Relationships aren't hard to set up in SimpleDB. Conceptually, you could easily create an attribute on a runner item called `race` and then place a race name (like `Race_01`) in it. Even better, there's nothing stopping you from holding a collection of race names in this value. The reverse is also true: You could easily hold a collection of runner names in a `race` domain (shown in Listing 9). Just remember: You can't actually join the two domains via Amazon's query language; you'll have to do that on your end.

Listing 9. Creating a Runners domain and two runners

```
sdb.createDomain(new CreateDomainRequest("Runners"));

List<ReplaceableItem> runners = new ArrayList<ReplaceableItem>();

runners.add(new ReplaceableItem().withName("Runner_01").withAttributes(
   new ReplaceableAttribute().withName("Name").withValue("Sally Smith")));

runners.add(new ReplaceableItem().withName("Runner_02").withAttributes(
 new ReplaceableAttribute().withName("Name").withValue("Richard Bean")));

sdb.batchPutAttributes(new BatchPutAttributesRequest("Runners", runners));
```

Once I've created a `Runners` domain and added runners to it, I can update an existing race and add my runners there, as in Listing 10:

Listing 10. Updating a race to hold two runners

```
races.add(new ReplaceableItem().withName("Race_01").withAttributes(
  new ReplaceableAttribute().withName("Name").withValue("Charlottesville Marathon"),
  new ReplaceableAttribute().withName("Distance").withValue("026.2"),
  new ReplaceableAttribute().withName("Runners").withValue("Runner_01"),
  new ReplaceableAttribute().withName("Runners").withValue("Runner_02")));
```

The bottom line is that relationships are possible, but you'll have to manage them outside of SimpleDB. If you wanted to get the full names of all runners in `Race_01`, for instance, you'd have to get the names in one query and then issue queries (two in this case, because `Race_01` only has two attribute values) against the `runner` domain to get the answers.

Cleanup operations

Cleaning up after yourself is important, so I'll conclude with a quick clean sweep using Amazon's SDK. Cleanup operations aren't much different from creating data and querying for it; you just create `Request` types and issue deletes.

Deleting `Race_01` is pretty easy, shown in Listing 11:

Listing 11. Issuing a delete in SimpleDB

```
sdb.deleteAttributes(new DeleteAttributesRequest(domain, "Race_01"));
```

If I used a `DeleteAttributesRequest` to delete an item, what you do you think I'd need to use to delete a domain? You guessed it: a `DeleteDomainRequest`!

Listing 12. Deleting a domain in SimpleDB

```
sdb.deleteDomain(new DeleteDomainRequest(domain));
```

This tour isn't over!

We're not done touring the clouds via Amazon's SimpleDB, but we are done with the Amazon SDK for now. Amazon's SDK is functional and can be useful to a point, but if you want to model things — like races and runners — you might want to leverage something like JPA. Next month in Part 2, we'll find out what happens when we combine JPA with SimpleDB. Until then, have fun with lexicographic searches!

Resources

Learn

- *Java development 2.0*: This developerWorks series explores technologies and tools that are redefining the Java development landscape, including CouchDB (November 2009) and Bigtable (May 2010).
- "*Cloud computing with Amazon Web Services, Part 5: Dataset processing in the cloud with SimpleDB*" (Prabhakar Chaganti, developerWorks, February 2009): Learn basic Amazon SimpleDB (SDB) concepts and explore some of the functions provided by boto, an open source Python library for interacting with SDB.
- "Eventually Consistent - Revisited" (Werner Vogels, All Things Distributed, December 2008): Amazon's CTO explains Eric Brewer's *CAP Theorem* and its impact on Amazon's Web Services infrastructure.
- Video demo: An introduction to Amazon SimpleDB (Andrew Glover, developerWorks): Learn how SimpleDB works, understand its advantages and disadvantages, and see how to create a record, query data, and delete data.
- "NoSQL Patterns" (Ricky Ho, Pragmatic Programming Techniques, November 2009): An overview and listing of NoSQL databases, followed by an in-depth look at the common architecture of NoSQL datastores.
- "Bigtable: A Distributed Storage System for Structured Data" (Fay Chang et al., Google, November 2006): Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Amazon SDK: Download Amazon's SDK and start leveraging the Amazon Web Services infrastructure. (You'll have to create an account with Amazon if you don't have one already.)
- SDB Tool: A Firefox GUI plugin that makes using Amazon SimpleDB easier.

Discuss

- Get involved in the My developerWorks community.

About the author

Andrew Glover

Andrew Glover is a developer, author, speaker, and entrepreneur with a passion for behavior-driven development, Continuous Integration, and Agile software development. He is the founder of the easyb Behavior-Driven Development (BDD) framework and is the co-author of three books: Continuous Integration, Groovy in Action, and Java Testing Patterns. You can keep up with Andrew by reading his blog and by following him on Twitter.