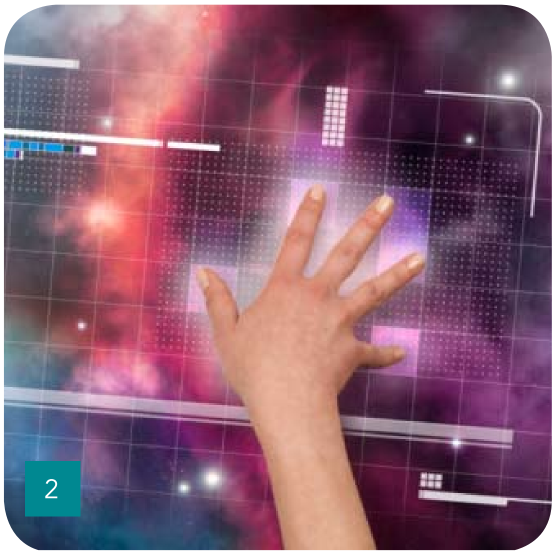
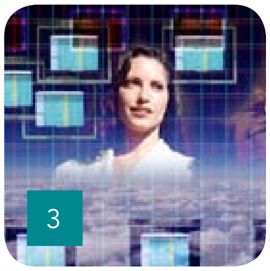
Diving Into Cloud Development: Working with Data

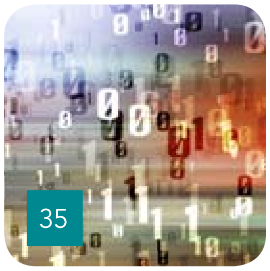
an internet.com® Developer eBook

Contents...

Diving Into Cloud Development: Working with Data







This content was adapted from Internet.com's DevX website and IBM developWorks. developerWorks content is used with permission. Contributors: Herman Mehling and Andrew Glover.

Introduction

Internet applications continue to evolve and with the advent of Cloud Computing, the changes continue. One area of Cloud Computing that developers are finding of extreme value is the ability to store data. The Cloud allows for broad access as well as for levels of scalability that might not be available locally.

Of course, a few of the paradigms related to working with data need to evolve as migration to the cloud occurs. One such paradigm is the concept of NoSQL. In this eBook you'll not only learn what NoSQL is, but also why it is important and how it can be used. You discover how to make adapting to NoSQL domain modeling and cloud-based storage a little easier. Not only will you learn about NoSQL, but you will also discover Bigtable, CouchDB and SimpleDB and how to use each of them.

This eBook then does a deeper dive into showing you how to use Amazon's SimpleDB with Java to get massively scalable storage storage. You'll get to put schemaless data storage into action! ■

The Rise of NoSQL Databases

By Herman Mehling

If you think databases are all of the relational SQL kind (as in MySQL, MS SQL and PostreSQL), think again. Better still: think NoSQL, as in non-relational, distributed databases that do not require fixed-table schemas, typically scale horizontally, and provide superior data replication to SQL databases.

While not exactly new – the NoSQL concept has been around for 10 years or so – NoSQL has been attracting a lot of attention in recent years, primarily due to big-name production implementations. Two of the best known implementations are Amazon's Dynamo and Google's BigTable.

However, there are also many publicly available open source variants, such as Cassandra, CouchDB, Hbase, MongoDB, Redis and Riak CounchDB.

In the past year, the NoSQL phenomenon has blossomed into something of a movement, popularized by events in this country and Europe, and by leading companies in diverse industries adopting NoSQL technologies.

Driving interest in NoSQL databases is the limitations of RDBMSes, which were originally built for single users on single machines doing single operations. RDBMSes weren't designed for today's computing world of thousands, even millions, of users simultaneously accessing a database full of images, and digital and audio/video data.

NoSQL is very much a user-led phenomenon highlighted by the likes of Google, Amazon, Facebook, LinkedIn and Twitter, which created their own distributed data management technologies, says Mathew Aslett, an analyst at the 451 Group.

Aslett says Google and the other companies above chose NoSQL to deliver the performance and scalability benefits that traditional database products cannot match.

Companies and developers choose NoSQL because it gives them solutions unavailable with SQL, says Damien Katz, co-founder and CEO, Couchio, developer of CouchDB.

"NoSQL technologies give people a choice of tools that SQL cannot," says Justin Sheehy, CTO, Basho Technologies. Basho is the creator of Riak, a distributed data store that aims to combines high availability and powerful partitioning.

Riak's high availability means that applications built using Riak remain both read and write available under almost any conditions, says Sheehy.

Basho's customers include Comcast and Electronic Arts. The latter uses Basho infrastructure to support 7 million daily users of Warhammer Online on Facebook, saving each player's status every half-minute.

While high availability is the main benefit of Riak, replication is the main benefit of CouchDB, a distributed, fault-tolerant and schema-free document-oriented database.

Unlike SQL databases, which are designed to store and report on highly structured, interrelated data, CouchDB (still in Beta) stores and reports on large amounts of semi-structured, document-oriented data.

"CouchDB greatly simplifies the development of document-oriented applications, which make up the bulk of collaborative web applications," says Katz.

"One of our customers is the BBC, which is using the CouchDB to handle 150 million requests per day in two clusters of machines," says Katz. The BBC, like other customers, selected CouchDB because of its superior replication and easy upgrade capabilities, says Katz.

He explains that NoSQL databases handle upgrades much better than SQL ones.

"In a SQL database, updates involve updating the schema and the stored data," says Katz. "This often causes problems as new needs arise that weren't anticipated in the initial database designs. With CouchDB, no schema is enforced, so new document types with new meaning can be safely added alongside the old." ■

Java Development 2.0: NoSQL

By Andrew Glover

Relational databases have ruled data storage for more than 30 years, but the growing popularity of schemaless (or NoSQL) databases suggest that change is underway. While the RDBMS provides a rock-solid foundation for storing data in traditional client-server architectures, it doesn't easily (or cheaply) scale to multiple nodes. In the era of highly scalable Web apps like Facebook and Twitter, that's a very unfortunate weakness to have.

Whereas earlier alternatives to the relational database (remember object-oriented databases?) failed to solve a truly urgent problem, NoSQL databases like Google's Bigtable and Amazon's SimpleDB arose as a direct response to the Web's demand for high scalability. In essence, NoSQL could be the killer app for a killer problem — one that Web application developers are likely to encounter more, not less, as Web 2.0 evolves.

I'll get you started with schemaless data modeling, which is the primary hurdle of NoSQL for many developers trained in the relational mindset. As you'll learn, starting with a domain model (rather than a relational one) is key to easing your way in. If you're using Bigtable, as my example does, you can also enlist the help of Gaelyk: a lightweight framework extension to Google App Engine.

NoSQL: A new mindset?

When developers talk about non-relational or NoSQL databases, the first thing often said is that they require a change in mindset. In my opinion, that actually depends upon your initial approach to data modeling. If you are accustomed to designing applications by modeling the database structure first (that is, you figure out tables and their associated relationships first), then data modeling with a schemaless datastore like Bigtable will require rethinking how you do things. If, however, you design your applications starting with the domain model, then Bigtable's schemaless structure will feel more natural.

Non-relational datastores don't have join tables or primary keys, or even the notion of foreign keys (although keys of both types are present in a looser form). So you'll probably end up frustrated if you try to use relational modeling as a foundation for data modeling in a NoSQL database. Starting from a domain model simplifies things; in fact, I've found that the flexibility of the schemaless structure living under the domain model is refreshing.

The relative complexity of moving from a relational to a schemaless data model depends on your approach: namely whether you start from a relational or a domain-based design. When you migrate to a datastore like CouchDB or Bigtable, you do lose the slickness of an established persistence platform like Hibernate (for now, at least). On the other hand, there's the green-pasture effect of being able to build it for yourself. And

Built to Scale

Along with the new problems of the highly scalable Web app come new solutions. Facebook doesn't rely on a relational database for its storage needs; instead it uses a key/value store — essentially a high performance HashMap. The in-house solution, dubbed Cassandra, is also used by Twitter and Digg and was recently donated to the Apache Software Foundation. Google is another Web entity whose explosive growth required it to seek non-relational data storage — Bigtable is the result.

in the process, you'll learn in-depth about schemaless datastores.

Entities and relationships

A schemaless datastore gives you the flexibility to design a domain model with objects first (something newer frameworks like Grails automatically facilitate). Your work going forward then becomes mapping your domain to the underlying datastore, which in the case of Google App Engine couldn't be easier.

In the article "Java development 2.0: Gaelyk for Google App Engine," I introduced Gaelyk, a Groovy-based framework that facilitates working with Google's underlying datastore. A big part of that article focused on leveraging Google's Entity object. The following example (from that article) shows how object entities work in Gaelyk.

Listing 1. Object persistence with Entity

```
def ticket = new Entity("ticket")
ticket.officer = params.officer
ticket.license = params.plate
ticket.issuseDate = offensedate
ticket.location = params.location
ticket.notes = params.notes
ticket.offense = params.offense
```

This approach to object persistence works, but it's easy to see how it could become tedious if you used ticket entities a lot — for example, if you were creating (or finding) them in various servlets. Having a common servlet (or Groovlet) handle the tasks for you would remove some of the burden. A more natural option — as I'll demonstrate — would be to model a Ticket object.

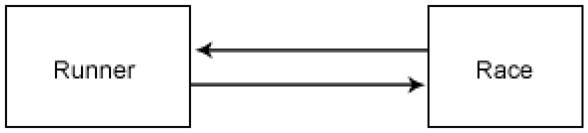
Back to the races

Rather than redo the tickets example from the introduction to Gaelyk, I'm going to keep things fresh and use a running theme in this article and build an application to demonstrate the techniques I discuss.
As the many-to-many diagram in Figure 1 shows, a Race has many Runners and a Runner can belong to many Races.

Design by object

The pattern of favoring the object model over the design of the database shows up in modern Web application frameworks like Grails and Ruby on Rails, which stress an object model's design and handle the underlying database schema creation for you.

Figure 1. Race and runners



If I were to use a relational table structure to design this relationship, I'd need at least three tables: the third being a join table linking a many-to-many relationship. I'm glad I'm not bound to the relational data model. Instead, I'll use Gaelyk (and Groovy code) to map this many-to-many relationship to Google's Bigtable abstraction for Google App Engine. The fact that Gaelyk allows an Entity to be treated like a Map makes the process quite simple.

One of the beauties of a schemaless datastore is that I don't have to know everything up front; that is, I can accommodate change much more easily than I could with a relational database schema. (Note that I'm not implying that you can't change a schema; I'm just saying that change is more easily accommodated without one.) I'm not going to define properties on my domain objects — I defer that to Groovy's dynamic nature (which allows me, in essence, to make my domain objects proxies to Google's Entity objects). Instead, I'll spend my time figuring out how I want to find objects and handle relationships. That's something NoSQL and the various frameworks leveraging schemaless datastores don't yet have built in.

Scaling with Shards

Sharding is a form of partitioning that replicates a table structure across nodes but logically divides data between them. For instance, one node could have all data related to accounts residing in the U.S. and another for all accounts residing in Europe. The challenge of shards occurs when nodes have relationships — that is, cross-shard joins. It's a hard problem to solve and in many cases goes unsupported.

The Model base class

I'll start by creating a base class that holds an instance of an Entity object. Then, I'll allow subclasses to have dynamic properties that will be added to the corresponding Entity instance via Groovy's handy setProperty method. setProperty is invoked for any property setter that doesn't actually exist in an object. (If this sounds strange, don't worry, it'll make sense once you see it in action.)

Listing 2 shows my first stab at a Model instance for my example application:

Listing 2. A simple base Model class

```
package com.b50.nosql

import com.google.appengine.api.datastore.DatastoreServiceFactory
import com.google.appengine.api.datastore.Entity
```

```
abstract class Model {

 def entity
 static def datastore = DatastoreServiceFactory.datastoreService

 public Model(){
  super()
 }

 public Model(params){
  this.@entity = new Entity(this.getClass().simpleName)
  params.each{ key, val ->
   this.setProperty key, val
  }
 }

 def getProperty(String name) {
  if(name.equals("id")){
   return entity.key.id
  }else{
   return entity."${name}"
  }
 }

 void setProperty(String name, value) {
  entity."${name}" = value
 }

 def save(){
  this.entity.save()
 }
}
```

Note how the abstract class defines a constructor that takes a Map of properties — I can always add more constructors later, and I will shortly. This setup is quite handy for Web frameworks, which often act off of parameters being submitted from a form. Gaelyk and Grails nicely wrap such parameters into an object called params. The constructor iterates over this Map and invokes the setProperty method for each key/value pair.

Looking at the setProperty method reveals that the key is set to the property name of the underlying entity, while the corresponding value is the entity's value.

Groovy tricks

As I previously mentioned, Groovy's dynamic nature allows me to capture method calls to properties that don't exist via the get and set Property methods. Thus, subclasses of Model in Listing 2 don't have to define properties of their own — they simply delegate any calls to a property to the underlying entity object.

The code in Listing 2 does a few other things unique to Groovy that are worth pointing out. First, I can bypass the accessor method of a property by prepending a @ to a property. I have to do this for the entity object reference in the constructor, otherwise I'd invoke the setProperty method. Invoking setProperty at this juncture would obviously break the pattern, as the entity variable in the setProperty method would be null.

Second, the call this.getClass().simpleName in the constructor sets the "kind" of entity — the simpleName property will yield a subclass's name without a package prefix (note that simpleName is really a call to getSimpleName, but that Groovy permits me to attempt to access a property without the corresponding JavaBeans-esque method call.)

Finally, if a call is made to the id property (that is, the object's key), the getProperty method is smart enough to ask the underlying key for its id. In Google App Engine, key properties of entities are automatically generated.

The Race subclass

Defining the Race subclass is as easy as it looks in Listing 3:

Listing 3. A Race subclass

```
package com.b50.nosql

class Race extends Model {
 public Race(params){
  super(params)
 }
}
```

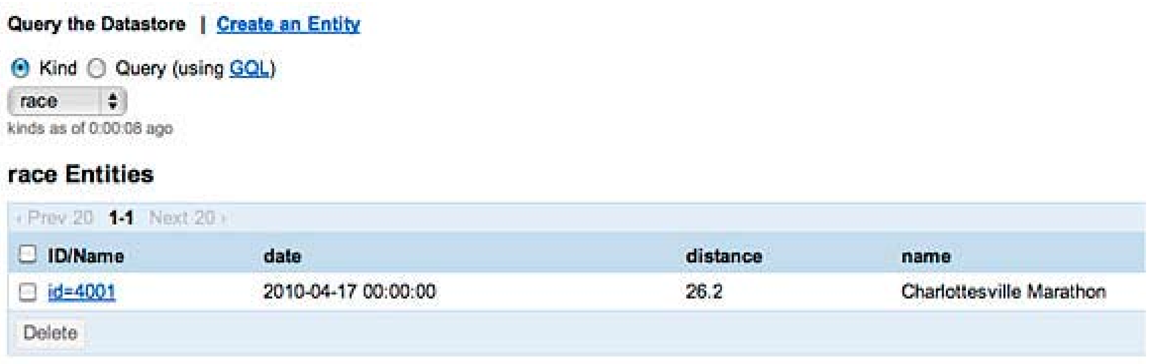
When a subclass is instantiated with a list of parameters (that is, a Map containing key/value pairs), a corresponding entity is created in memory. To persist it, I just need to invoke the save method.

Listing 4. Creating a Race instance and saving it to GAE's datastore

```
import com.b50.nosql.Runner

def iparams = [:]

def formatter = new SimpleDateFormat("MM/dd/yyyy")
def rdate = formatter.parse("04/17/2010")

iparams["name"] = "Charlottesville Marathon"
iparams["date"] = rdate
iparams["distance"] = 26.2 as double

def race = new Race(iparams)
race.save()
```

In Listing 4, which is a Groovlet, a Map (dubbed iparams) is created with three properties — a name, date, and distance for a race. (Note that in Groovy, an empty Map is created via [:].) A new instance of Race is created and consequently saved to the underlying datastore via the save method.
I can check the underlying datastore via the Google App Engine console to see that my data is actually there, as shown in Figure 2:

Figure 2. Viewing the newly created Race



race Entities

ID/Name	date	distance	name
id=4001	2010-04-17 00:00:00	26.2	Charlottesville Marathon

Finder methods yield persisted Entities

Now that I've got an Entity saved, it's helpful to have the ability to retrieve it; subsequently, I can add a "finder" method. In this case, I'll make it a class method (static) and I'll allow Races to be found by name (that is, I'll search based on the name property). I can always add other finders by other properties later.

I'm also going to adopt a convention for my finders specifying that any finder without the word all in its name is intended to find one instance. Finders with the word all (as in findAllByName) can return a Collection, or List, of instances. Listing 5 shows the findByName finder:

Listing 5. A simple finder searching based on an Entity’s name

```
static def findByName(name){
 def query = new Query(Race.class.simpleName)
 query.addFilter(“name”, Query.FilterOperator.EQUAL, name)
 def preparedQuery = this.datastore.prepare(query)
 if(preparedQuery.countEntities() > 1){
  return new Race(preparedQuery.asList(withLimit(1))[0])
 }else{
  return new Race(preparedQuery.asSingleEntity())
 }
}
```

This simple finder uses Google App Engine’s Query and PreparedQuery types to find an entity of kind “Race,” whose name equals (exactly) what is passed in. If more than one Race meets this criteria, the finder will return the first one out of a list, as instructed by the pagination limit of 1 (withLimit(1)).
The corresponding findAllByName would be similar but with an added parameter of how many do you want?, as shown in Listing 6:

Listing 6. Find all by name

```
static def findAllByName(name, pagination=10){
 def query = new Query(Race.class.getSimpleName())
 query.addFilter(“name”, Query.FilterOperator.EQUAL, name)
 def preparedQuery = this.datastore.prepare(query)
 def entities = preparedQuery.asList(withLimit(pagination as int))
 return entities.collect { new Race(it as Entity) }
}
```

Like the previously defined finder, findAllByName finds Race instances by name, but it returns all Races. Groovy’s collect method is rather slick, by the way: it allows me to drop in a corresponding loop that creates Race instances. Note how Groovy also permits default values for method parameters; thus, if I don’t pass in a second value, pagination will have the value 10.

Listing 7. Finders in action

```
def nrace = Race.findByName(“Charlottesville Marathon”)
assert nrace.distance == 26.2

def races = Race.findAllByName(“Charlottesville Marathon”)
assert races.class == ArrayList.class
```

The finders in Listing 7 work as you'd expect: findByName returns one instance while findAllByName returns a Collection (assuming there's more than one "Charlottesville Marathon").

Runner objects aren't much different

Now that I'm comfortable creating and finding instances of Race, I'm ready to create a speedy Runner object. The process is just as easy as creating my initial Race instance was; I just extend Model, as shown in Listing 8:

Listing 8. A Runner is too easy

```
package com.b50.nosql

class Runner extends Model{
 public Runner(params){
  super(params)
 }
}
```

Looking at Listing 8, I get the feeling that I'm almost to the finish line. I've still got to create the links between runners and races. And of course, I'll be modeling it as a many-to-many relationship because I hope my runners will run more than one race.

Domain modeling without the schema

Google App Engine's abstraction on top of Bigtable isn't an object-oriented one; that is, I can't save relationships as is, but I can share keys. Consequently, in order to model the relationship between Races and Runners, I'll store a list of Runner keys inside each instance of Race, and vice versa.

I'll have to add a bit of logic around my key-sharing mechanism, however, because I want the resulting API to be natural — I don't want to ask a Race for a list of Runner keys, I want a list of Runners. Luckily, this isn't hard.

In Listing 9, I've added two methods to the Race instance. When a Runner instance is passed to the addRunner method, its corresponding id is added to a Collection of ids residing in the runners property of the underlying entity. If there is an existing collection of runners, the new Runner instance key is added to it; otherwise, a new Collection is created and the Runner's key (the id property on the entity) is added to it.

Listing 9. Adding and retrieving runners

```
def addRunner(runner){
 if(this.@entity.runners){
  this.@entity.runners << runner.id
 }else{
  this.@entity.runners = [runner.id]
 }
}

def getRunners(){
 return this.@entity.runners.collect {
  new Runner( this.getEntity(Runner.class.simpleName, it) )
 }
}
```

When the getRunners method in Listing 9 is invoked, a collection of Runner instances is created from the underlying collection of ids. Thus, a new method (getEntity) is defined in the Model class, as shown in Listing 10:

Listing 10. Creating an entity from an id

```
def getEntity(entityType, id){
 def key = KeyFactory.createKey(entityType, id)
 return this.@datastore.get(key)
}
```

The getEntity method uses Google's KeyFactory class to create the underlying key that can be used to find an individual entity within the datastore.
Lastly, a new constructor is defined that accepts an entity type, as shown in Listing 11:

Listing 11. A newly added constructor

```
public Model(Entity entity){
 this.@entity = entity
}
```

As you can see from Listings 9, 10, and 11, and Figure 1's object model, I can add a Runner to any Race, and I can also get a list of Runner instances from any Race. In Listing 12, I create a similar linkage on the Runner's side of the equation. Listing 12 shows the Runner class's new methods.

Listing 12. Runners and their races

```
def addRace(race){
 if(this.@entity.races){
  this.@entity.races << race.id
 }else{
  this.@entity.races = [race.id]
 }
}

def getRaces(){
 return this.@entity.races.collect {
  new Race( this.getEntity(Race.class.simpleName, it) )
 }
}
```

In this way, I've managed to model two domain objects with a schemaless datastore.

Finishing the race with some runners

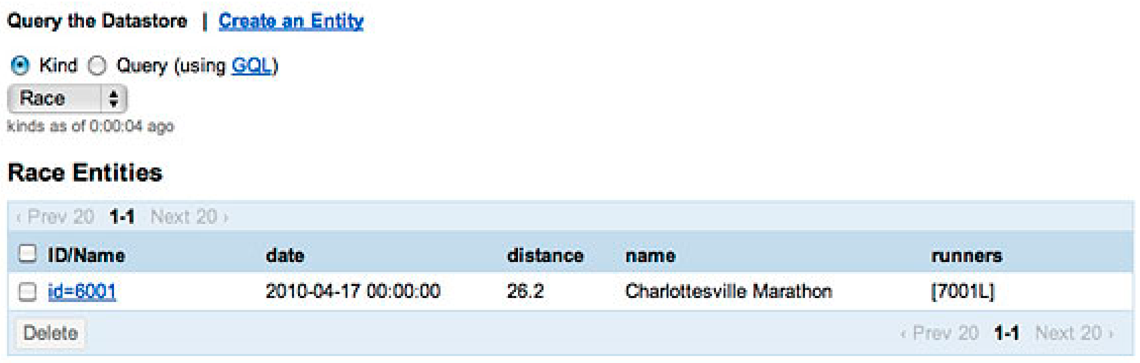
Now all I have to do is create a Runner instance and add it to a Race. If I want the relationship to be bidirectional, as my object model in Figure 1 shows, then I can add Race instances to Runners as well, shown in Listing 13:

Listing 13. Runners with their races

```
def runner = new Runner([fname:"Chris", lname:"Smith", date:34])
runner.save()

race.addRunner(runner)
race.save()

runner.addRace(race)
runner.save()
```

After adding a new Runner to the race and the call to Race's save, the datastore has been updated with a list of IDs as shown by the screenshot in Figure 3:

Figure 3. Viewing the new property of runners in a race



Query the Datastore | Create an Entity

● Kind ○ Query (using GQL)

Race

kinds as of 0:00:04 ago

Race Entities

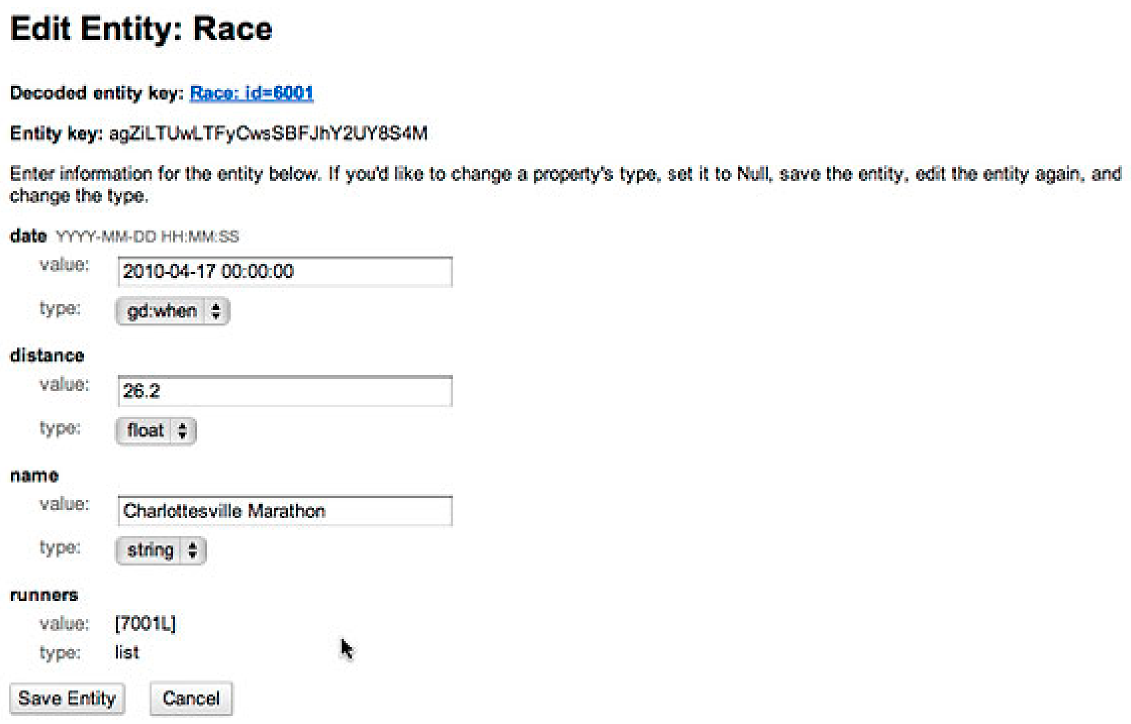
‹ Prev 20 1-1 Next 20 ›

ID/Name	date	distance	name	runners
id=6001	2010-04-17 00:00:00	26.2	Charlottesville Marathon	[7001L]

Delete ‹ Prev 20 1-1 Next 20 ›

By closely examining the data in Google App Engine, you can see that a Race entity now has a list of Runners, as shown in Figure 4.

Figure 4. Viewing the new list of runners



Edit Entity: Race

Decoded entity key: Race: id=6001

Entity key: agZiLTUwLTFyCwsSBFJhY2UY8S4M

Enter information for the entity below. If you'd like to change a property's type, set it to Null, save the entity, edit the entity again, and change the type.

date YYYY-MM-DD HH:MM:SS
value: 2010-04-17 00:00:00
type: gd:when

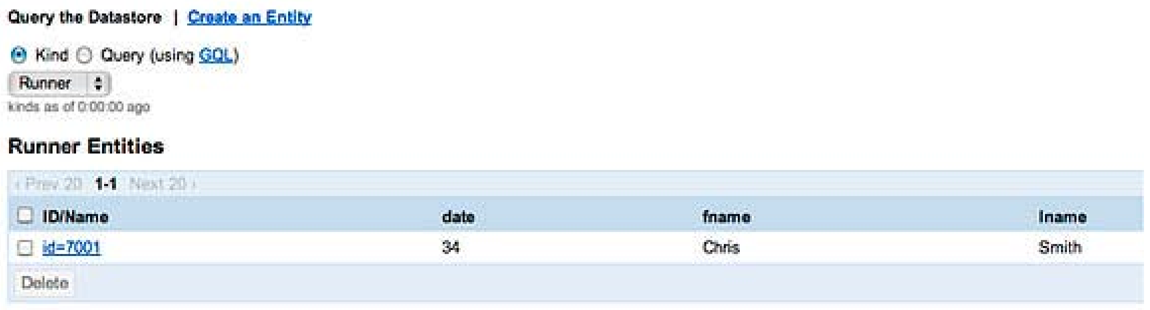
distance
value: 26.2
type: float

name
value: Charlottesville Marathon
type: string

runners
value: [7001L]
type: list

Likewise, before adding a Race to a newly created Runner instance, the property doesn't exist, as shown in Figure 5.

Figure 5. A runner without a race



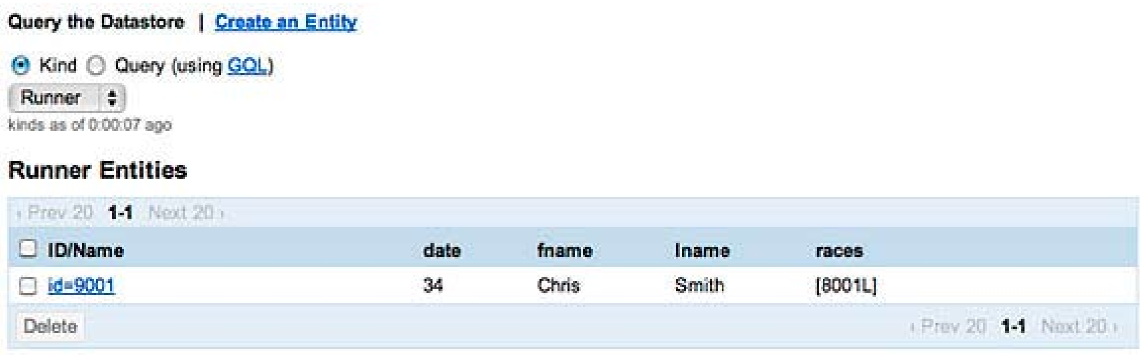
Runner Entities

‹ Prev 20 1-1 Next 20 ›

ID/Name	date	fname	lname
id=7001	34	Chris	Smith

Yet, after associating a Race to a Runner, the datastore adds the new list of races ids.

Figure 6. A runner off to the races



Runner Entities

‹ Prev 20 1-1 Next 20 ›

ID/Name	date	fname	lname	races
id=9001	34	Chris	Smith	[8001L]

The flexibility of the schemaless datastore is refreshing — properties are auto-added to the underlying store on demand. As a developer, I have no need to update or change a schema, much less deploy one!

Pros and cons of NoSQL

There are, of course, pros and cons to schemaless data modeling. One advantage of the Back to the Races application is that it's quite flexible. If I decide to add a new property to a Runner (such as SSN), I don't have to do much — in fact, if I include it in the constructor's parameters, it's there. What happens to the older instances that weren't created with an SSN? Nothing. They have a field that is null.

On the other hand, I've clearly traded consistency and integrity for efficiency. The application's current data architecture leaves me with no constraints — I could theoretically create an infinite number of instances of the same object. Under Google App Engine's key handling, they'd all have unique keys, but everything else would be identical. Worse, cascading deletes don't exist, so if I used the same technique to model a one-to-many relationships, and the parent were removed, I could end up with stale children. Of course, I could implement my own integrity checking — but that's key: I'd have to do it myself (much like I did everything else).

Speed readers

Speed is an important factor in the NoSQL-versus-relational argument. For a modern website passing data for potentially millions of users (think of Facebook's 400 million users and counting) the relational model is simply too slow, not to mention expensive. NoSQL's datastores, by contrast, are extremely fast when it comes to reads.

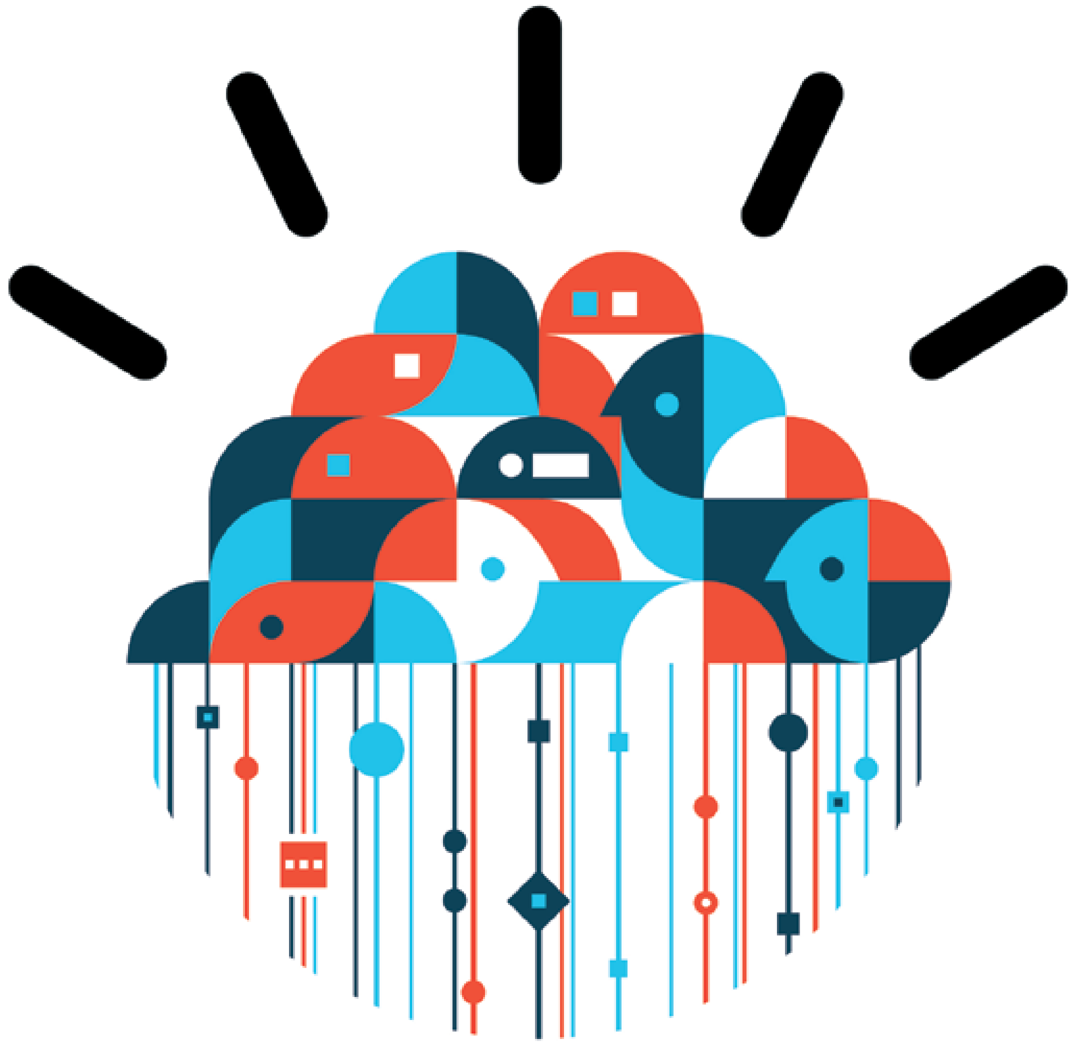
Using a schemaless datastore requires discipline. If I create various types of Races — some with names, some without, some with a date property, and others with a race_date property — I will just be shooting myself (and everyone else who leverages my code) in the foot.

Of course, it's still possible to use JDO and JPA with Google App Engine. Having used both the relational and schemaless models on multiple projects, I can say that Gaelyk's low-level API is the most flexible and fun to work with. Another advantage of using Gaelyk is gaining a closer understanding of Bigtable and schemaless datastores in general.

In conclusion

Fads come and go, and sometimes it's safe to ignore them (sage advice coming from a guy with a wardrobe full of leisure suits). But NoSQL looks less like a fad, and more like an emerging foundation for highly scalable Web application development. NoSQL databases won't replace the RDBMS, however; they'll supplement it. Myriad successful tools and frameworks live on top of relational databases, and RDBMSs themselves don't appear to be in any danger of waning in popularity.

What NoSQL databases do, finally, is present a timely alternative to the object-relational data model. They show us that something else is possible, and — for specific, highly compelling use cases — better. Schemaless databases are most appropriate for multinode Web apps that need speedy data retrieval and scalability. As a nifty side-effect, they're teaching developers to approach data modeling from a domain-oriented perspective, rather than a relational one. ■



Is cloud computing secure computing?

The emerging computing option known as "cloud" has been embraced by consumers and businesses alike—delivering everything from music and photos through a public cloud to business e-mail and data storage through a private, corporate cloud.

Cloud is now poised to create new business models across industries and to move into areas that matter at a societal level, such as healthcare, finance, government services and national defense.

Whether it's a public cloud on the Internet or a private cloud that dwells behind a corporate firewall, cloud resources are subject to the same risks as any shared resource, and they require the same protections. To capture the promise of cloud computing, issues of privacy, authentication and security must be addressed. And a one-size-fits-all approach—one that treats enterprise e-mail the same as tweets, or healthcare data the same as uploaded photos—will not work.

The good news is, enterprises and governments around the world are realizing this, and are pioneering approaches that should bring cloud computing into the mainstream of the way our world works.

The United States Air Force (USAF) has adopted a new project to design and demonstrate a mission-oriented private cloud environment. The ten-month project demonstrates the advanced security and analytics technologies currently in use in commercial sectors.

McGill University Health Centre is implementing a private storage cloud to securely house patient data. Over 800,000 patient cases at multiple sites are provided to clinicians around the clock—providing a strategic and single view of data, including clinical images.

This list could go on, with examples in areas from retail to banking to education. If we do this right—if we build in reliability, security and privacy by design—then our cloud-based systems, both public and private, have the potential to bring new heights of intelligence to the way our world works.

Let's build a smarter planet. Join us and see what others are doing at **ibm.com/smarterplanet**

Java Development 2.0: Cloud Storage with Amazon's SimpleDB, Part 1

By Andrew Glover

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.

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.

Now, 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

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).

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 later in this eBook).

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,

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.)

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, Items 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 Strings? 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 Items 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!

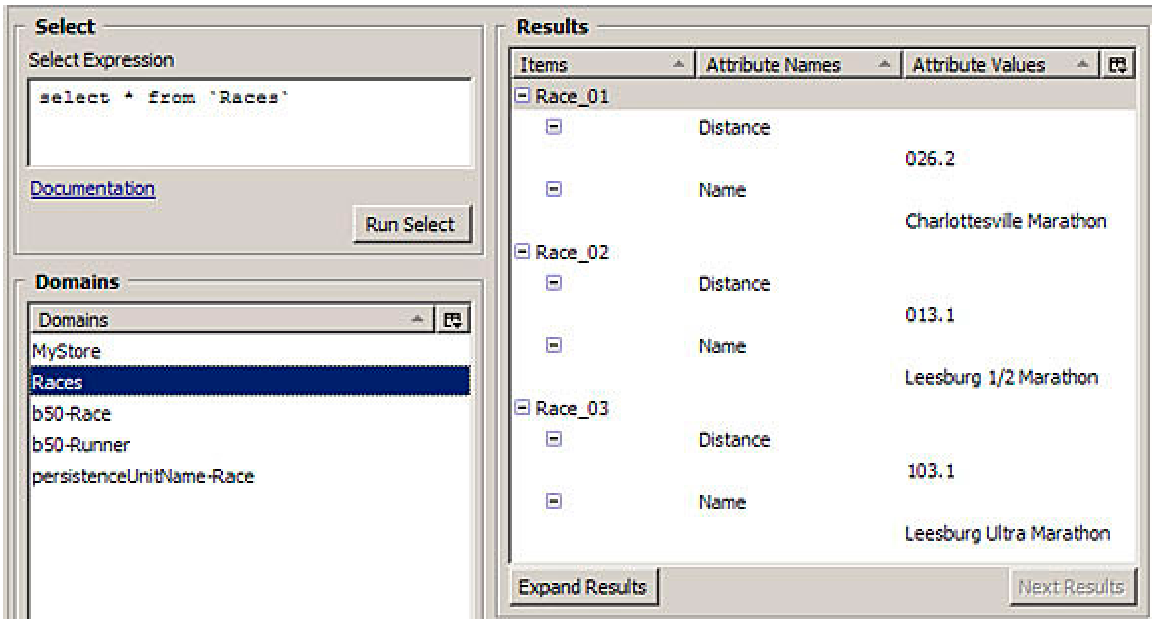
Lexicographic searching

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



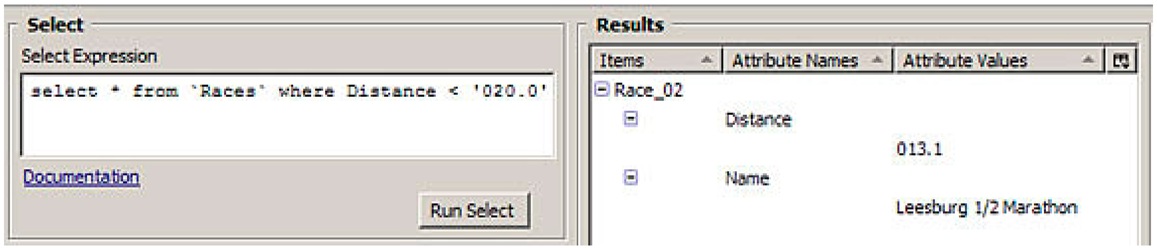
```
select * from `Races`
```

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'
```

Items	Attribute Names	Attribute Values
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! ■

Java Development 2.0: Cloud Storage with Amazon's SimpleDB, Part 2

By Andrew Glover

In the first half of this introduction to SimpleDB, I showed you how to leverage Amazon's own API to model a CRUD-style racing application. Aside from the obvious uniqueness for most Java developers of Amazon's string-only approach to data types, you might have found yourself looking at the Amazon API with some skepticism. After all, the APIs for leveraging a relational database are by now fairly standard and well-thought-out — and perhaps more importantly, they are familiar.

Behind the scenes, many relational frameworks today implement the Java Persistence API. This makes modeling domain objects for almost any type of Java application both easy and familiar across the range of RDBMSs. It's natural to be resistant to learning a new approach to domain modeling when you've already mastered one that works — and the good news is that with SimpleDB, you don't have to.

In this second half of my introduction to SimpleDB, I'll show you how to refactor the racing application from Part 1 to be compliant with the JPA specification. Then we'll port the application to SimpleJPA and discover some of the ways that this innovative, open source platform can make adapting to NoSQL domain modeling, and cloud-based storage, a little easier.

Why SimpleDB?

Amazon's SimpleDB is a simple, yet massively scalable and reliable cloud-based datastore. Due to its non-relational/NoSQL foundation, SimpleDB is both flexible and lightning fast. As part of the Amazon Web services family, SimpleDB uses HTTP as its underlying communication mechanism, so it's able to support language bindings ranging from the Java language to Ruby, C#, and Perl. SimpleDB is also inexpensive: Under SimpleDB's licensing, you pay only for the resources you consume, which differs from the more traditional method of buying a license up front for predicted usage and space. As part of the emerging class of NoSQL, or non-relational, datastores, SimpleDB is related to Google's Bigtable or CouchDB.

Hibernate and JPA: A brief history

Numerous Java developers today leverage Hibernate (and Spring) for data persistence. In addition to being a bellwether of open source success, Hibernate has changed the field of ORM for good. Before we had Hibernate, Java developers had to deal with the quagmire of EJB entity beans; before that, we basically rolled our own ORMs or bought one from a vendor like IBM. Hibernate washed away all that complexity and cost in favor of the POJO-based modeling platform many of us take for granted today.

The Java Persistence API (JPA) was created in response to the popularity of Hibernate's innovation of leveraging POJOs for data modeling. Today, EJB 3.0 implements JPA, and so does Google App Engine. Even Hibernate itself is a JPA implementation, assuming you use the Hibernate EntityManager.

Given how comfortable Java developers have become with modeling data-centric applications using POJOs, it makes sense that a datastore like SimpleDB should give us a similar option. After all, it's kind of like a database, isn't it?

Data modeling with objects

In order to use SimpleJPA, we need to do a little work on our Racer and Runner objects, bringing them up to speed with the JPA specification. Fortunately, the basics of JPA are pretty simple: you decorate normal POJOs with annotations and an EntityManager implementation takes care of the rest — no XML required.

Two of the main annotations used by JPA are @Entity and @Id, which specify a POJO as persistent and delineate its identity key, respectively. For the purpose of converting our racing application to JPA, we'll also need two annotations used for managing relationships: @ OneToMany and @ManyToOne.

In the first half of this article, I showed you how to persist runners and races. I never made use of any objects to represent those entities, however — I just used Amazon's raw API to persist the properties of both. If I wanted to model a simple relationship between a race and its runners, I could do so as shown in Listing 1:

Listing 1. A simple Race object

```
public class Race {
 private String name;
 private String location;
 private double distance;
 private List<Runner> runners;

 //setters and getters left out...
}
```

In Listing 1, I've specified a Race object with four properties, the last of which is a Collection of runners. Next, I can create a simple Runner object (shown in Listing 2) that holds each runner's name (I'll keep it really simple for now) and SSN along with the Race instance she or he is racing in.

Listing 2. A simple Runner related to a Race

```
public class Runner  {
 private String name;
 private String ssn;
 private Race race;

 //setters and getters left out...
}
```

As you can see in Listings 1 and 2, I've logically modeled a many-to-one relationship between runners and a race. In a real-world situation, it would probably be more appropriate to make the link many-to-many (don't runners usually run more than one race?), but I'm keeping it easy. I've also left out the constructors, setters, and getters for now. I'll show them to you later.

Annotations in JPA

Getting these two objects ready for SimpleJPA isn't terribly challenging. First, I have to signify my intent to make them persistable by adding the @Entity annotation to each one. I also need to delineate the relationships properly using @ OneToMany in the Race object and @ManyToOne in the Runner object.

The @Entity annotation is attached at the class level and the relationship annotations are attached at the getter level. All of this is demonstrated in Listings 3 and 4:

Listing 3. A JPA-annotated Race

```
@Entity
public class Race {
 private String name;
 private String location;
 private double distance;
 private List<Runner> runners;

 @OneToMany(mappedBy = "race")
 public List<Runner> getRunners() {
  return runners;
 }

 //other setters and getters left out...
}
```

In Listing 3, I've decorated the getRunners method with a @OneToMany annotation. I've also specified that the relationship can be found using the race property on the entity Runner.

In Listing 4, I'll similarly annotate the getRace method in the Runner object.

Listing 4. A JPA-annotated Runner

```
@Entity
public class Runner  {
 private String name;
 private String ssn;
 private Race race;

 @ManyToOne
 public Race getRace() {
  return race;
 }

 //other setters and getters left out...
}
```

Most datastores (relational or not) need some way of delineating uniqueness among data. So if I want to make these two objects persistent in a datastore, I at least have to add IDs to them. In Listing 5, I've added an id property of type BigInteger to the Race domain object. I will also do the same for Runner.

Listing 5. Adding an ID to Race

```
@Entity
public class Race {
 private String name;
 private String location;
 private double distance;
 private List<Runner> runners;
 private BigInteger id;

 @Id
 public BigInteger getId() {
  return id;
 }

 @OneToMany(mappedBy = "race")
 public List<Runner> getRunners() {
  return runners;
 }

 //other setters and getters left out...
}
```

The @Id annotation in Listing 5 doesn't provide any information about how the ID is managed. The program will assume I'm doing that manually, rather than using an EntityManager, for example.

Enter SimpleJPA

So far, I haven't done anything specific to SimpleDB. The Race and Runner objects are generically annotated with JPA annotations and could be persisted in any datastore that is supported by a JPA implementation. Options include Oracle, DB2, MySQL, and (as you've probably guessed by now) SimpleDB.

SimpleJPA is an open source implementation of JPA for Amazon's SimpleDB. While it doesn't support the entire JPA specification (for example, you can't join in JPA queries), it supports a large enough subset to be worth exploring.

A big advantage of using SimpleJPA is that it attempts to seamlessly handle the lexicographic issues I discussed in the first half of this article. SimpleJPA does the string conversion and any subsequent padding (if required) for objects that rely on numeric types. For the most part, this means that you don't have to change your domain model to reflect String types. (There is one exception to that rule, which I'll explain in a moment.)

Because SimpleJPA is a JPA implementation, you can easily use JPA-compliant domain objects with it. SimpleJPA only requires that you use String IDs, meaning that your id property must be a java.lang.String. To make things easier, SimpleJPA provides the base class IdedTimestampedBase, which manages the domain object's ID property, as well as the date attributes created and updated. (Under the hood, SimpleDB generates a unique Id.)

Porting the app to SimpleJPA

To make the Race and Runner classes compliant with SimpleJPA, I could either extend SimpleJPA's handy base class or change each class's id property from BigInteger to String. I've opted for the first option, as shown in Listing 6:

Listing 6. Changing Race to use SimpleJPA's IdedTimestampedBase base class

```
@Entity
public class Race extends IdedTimestampedBase{
 private String name;
 private String location;
 private double distance;
 private List<Runner> runners;

 @OneToMany(mappedBy = "race")
 public List<Runner> getRunners() {
  return runners;
 }

 //other setters and getters left out...
}
```

I won't show you the same code for Runner, but feel free to run through it on your own: just extend IdedTimestampedBase and remove the id property from the Runner.

Updating the IDs for Race and Runner is the first step of making the racing application compliant with SimpleJPA. Next, I need to exchange primitive datatypes (like double, int, and float) for objects like Integer and BigDecimal.
I'll start with the distance property of Race. I've found BigDecimal to be more reliable than Double (in the current release of SimpleJPA), so I changed Race's distance property to a BigDecimal, as shown in Listing 7:

Listing 7. Changing distance to BigDecimal

```
@Entity
public class Race extends IdedTimestampedBase{
 private String name;
 private String location;
 private BigDecimal distance;
 private List<Runner> runners;

 @OneToMany(mappedBy = “race”)
 public List<Runner> getRunners() {
  return runners;
 }

 //other setters and getters left out...
}
```

Now both Runner and Race are ready to be persisted via a SimpleJPA implementation.

Using SimpleJPA with SimpleDB

Manipulating your domain objects against SimpleDB with SimpleJPA isn’t any different from going against a normal relational database with a JPA implementation. If you’ve ever done any application development with JPA, then nothing about it should surprise you. The only thing that might be new is configuring SimpleJPA’s EntityManagerFactoryImpl, which will require your Amazon Web Services credentials and the prefix name for your SimpleDB domain. (Another option would be to provide a properties file containing your credentials on the classpath.)

Using the prefix name you specify when creating an instance of SimpleJPA’s EntityManagerFactoryImpl will result in SimpleDB domains that start with your prefix followed by a dash, and then your domain object’s name. So, if I specify “b50” for my prefix, then when I create a Race item in SimpleDB, the domain will be “b50-Race”.

Once you’ve created an instance of SimpleDB’s EntityManagerFactoryImpl, everything else is driven by the interface. You’ll need an EntityManager instance, which you obtain from the EntityManagerFactoryImpl, as shown in Listing 8:

Listing 8. Obtaining an EntityManager

```
Map<String,String> props = new HashMap<String,String>();
props.put(“accessKey”,”...”);
props.put(“secretKey”,”..”);

EntityManagerFactoryImpl factory =
  new EntityManagerFactoryImpl("b50", props);

EntityManager em = factory.createEntityManager();
```

Manipulating domain objects

Once you have a handle to an EntityManager, you can manipulate domain objects at will. For instance, I can create a Race instance like so:

Listing 9. Creating a Race

```
Race race = new Race();
race.setName("Charlottesville Marathon");
race.setLocation("Charlottesville, VA");
race.setDistance(new BigDecimal(26.2));
em.persist(race);
```

In Listing 9, SimpleJPA handles all the HTTP requests to create Race in the cloud. Using SimpleJPA means that I can also retrieve the race using a JPA query, as shown in Listing 10. (Remember that you cannot do joins with these queries, but I can still search with numbers.)

Listing 10. Finding a race by distance

```
Query query = em.createQuery("select o from Race o where o.distance = :dist");
query.setParameter("dist", new BigDecimal(26.2));

List<Race> races = query.getResultList();
for(Race race : races){
 System.out.println(race);
}
```

From numbers to strings

SimpleJPA's under-the-hood number-to-string magic is especially nice; for instance, if you enable query printing in SimpleJPA, you can see what queries it issues to SimpleDB. The query submitted is shown in Listing 11. Note how distance is encoded.

Listing 11. SimpleJPA handles numbers nicely!

```
amazonQuery: Domain=b50-Race, query=select * from `b50-Race`
  where `distance` = '09223372036854775834199999999999999928946'
```

Automatic padding and encoding makes things a lot easier, don't you think?

Relationships in SimpleJPA

Even though SimpleDB doesn't permit domain joins in queries, you can still have related items across domains. Like I showed you in Part 1, you can simply store the key of a related object in another and then retrieve that object when you need it. That's what SimpleJPA does, too. For instance, earlier I showed you how to link Runners to a Race using JPA annotations. Thus, I can create an instance of a Runner, add the existing race instance to it, and then persist the Runner instance, as shown in Listing 12:

Listing 12. Relationships with SimpleJPA

```
Runner runner = new Runner();
runner.setName("Mark Smith");
runner.setSsn("555-55-5555");
runner.setRace(race);
race.addRunner(runner);

em.persist(runner);
em.persist(race); //update the race now that it has a runner
```

Also note from Listing 12 that I have to update the Race instance to reflect the fact that I added a Runner instance to it (also note, I added a addRunner method to Race that simply adds a Runner to the internal Collection of Runners).

Once again, if I search for a race by its distance, I can also get a listing of its runners like Listing 13:

Listing 13. More relationship fun!

```
Query query = em.createQuery("select o from Race o where o.distance = :dist");
query.setParameter("dist", new BigDecimal(26.2));

List<Race> races = query.getResultList();

for(Race races : race){
 System.out.println(race);
 List<Runner> runners = race.getRunners();
 for(Runner rnr : runners){
  System.out.println(rnr);
 }
}
```

Using an EntityManager instance enables me to delete entities via the remove method, shown in Listing 14:

Listing 14. Removing a class instance

```
Query query = em.createQuery("select o from Race o where o.distance = :dist");
query.setParameter("dist", new BigDecimal(26.2));

List<Race> races = query.getResultList();

for(Race races : race){
 em.remove(race);
}
```

While I've removed a Race instance in Listing 14, any related Runners aren't removed. (I, of course, can handle this by using a JPA's EntityListeners annotation, which means I can hook into a removal event and use it to remove Runner instances.)

In conclusion

This whirlwind tour of SimpleDB has shown you how to manipulate objects in the non-relational datastore using both the Amazon Web services API and SimpleJPA. Simple JPA implements a subset of the Java Persistence API to make object persistence in SimpleDB easier. One of the conveniences of using SimpleJPA, as you've seen, is that it automatically converts primitive types to the string objects that SimpleDB recognizes. SimpleJPA also handles SimpleDB's no-join rules for you automatically, making it easier to model relationships. SimpleJPA's extensive listener interfaces also make it possible to implement logical data integrity rules, which you've probably come to expect from the relational world.

The bottom line on SimpleJPA is that it can help you access significant, inexpensive scalability quickly and easily. With SimpleJPA, you can leverage the knowledge you already have from years of working with frameworks like Hibernate in a non-relational, cloud-based storage environment. ■

Resources

By Andrew Glover

• Java development 2.0: This developerWorks series explores technologies and tools that are redefining the Java development landscape, including Gaelyk (December 2009), Google App Engine (August 2009), and CouchDB (November 2009).
http://www.ibm.com/developerworks/views/java/libraryview.jsp?search_by=Java+development+2.0

• "NoSQL Patterns" (Ricky Ho, Pragmatic Programming Techniques, November 2009): An overview and listing of NoSQL databases, followed by a more in-depth look at the common architecture of NoSQL datastores.
http://horicky.blogspot.com/2009/11/nosql-patterns.html

• "Cloud computing with Amazon Web Services, Part 5: Dataset processing in the cloud with SimpleDB" (Prabhakar Chaganti, developerWorks, February 2009): Learn basic Amazon SimpleDB concepts and explore some of the functions provided by boto, an open source Python library for interacting with SDB.
http://www.ibm.com/developerworks/library/ar-cloudaws5/

• "Cloud computing with Amazon Web Services, Part 1: Introduction" (Prabhakar Chaganti, IBM developerWorks, July 2008): Explore how the services provide a compelling alternative for architecting and building scalable, reliable applications.
http://www.ibm.com/developerworks/library/ar-cloudaws1/

• Google App Engine for Java: Part 3: Persistence and relationships" (Richard Hightower, developerWorks, August 2009): Rick Hightower explains the shortcomings of Google App Engine's current Java-based persistence framework and discusses some of the workarounds.
http://www.ibm.com/developerworks/java/library/j-gaej3.html

• "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.
http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

• "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.
http://labs.google.com/papers/bigtable.html

•"Saying Yes to NoSQL; Going Steady with Cassandra" (John Quinn, Digg Blogs, March 2010): Digg's VP of engineering explains the decision to switch from MySQL to Cassandra.
http://highscalability.com/blog/2010/3/10/saying-yes-to-nosql-going-steady-with-cassandra-at-digg.html

• "Sharding with Max Ross" (JavaWorld podcast, July 2008): Andrew Glover talks with Google's Max Ross about the technique of sharding and the development of Hibernate Shards.
http://www.javaworld.com/podcasts/jtech/2008/072408jtech.html

• "Is the Relational Database Doomed?" (Tony Bain, ReadWriteEnterprise, February 2009): With non-relational databases cropping up both inside and outside of the cloud, a clear message is emerging: "If you want vast, on-demand scalability, you need a non-relational database."
http://www.readwriteweb.com/enterprise/2009/02/is-the-relational-database-doomed.php

• "The Vietnam of Computer Science" (Ted Neward, June 2006): Addresses the challenges associated with mapping objects to relational models.
http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx

• developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.
http://www.ibm.com/developerworks/java/