Migrating from MySQL to Amazon SimpleDB

Contents

Contents 2
Introduction 3
The Simple Customer Application 3
Reasons for Migrating 5
Migrating MySQL Schemas to Amazon SimpleDB 6
Migration from a CRUD Perspective 10
  Creating or Inserting Data 11
  Reading or Querying Data 14
    Iterating through Results 16
    Working with JOINs 18
  Updating Data 19
  Deleting Data 21
Additional Benefits of Migrating to Amazon SimpleDB 22
Conclusion 22
Appendix 23
  Source Code 23
  Amazon Machine Image 23
  Migrating Simple Customer data 23

Introduction

Amazon SimpleDB is a web service that provides core database functions like fast, real-time lookup and querying of structured data. Amazon SimpleDB leverages Amazon's cloud infrastructure to offer developers a highly available, scalable, fault-tolerant data store.

Many developers find Amazon SimpleDB to be a highly effective data storage solution for building new web applications, since it automatically takes care of the many chores associated with managing and scaling a database. Developers are not forced to worry about data modeling (Amazon SimpleDB schemas are flexible), index maintenance and performance tuning (both are done automatically), or data replication (also done automatically).

Amazon SimpleDB can be an ideal solution for existing applications as well. Using an implicitly scalable data storage solution like Amazon SimpleDB can often give new life to existing applications. By using a cloud-based data storage solution, even monolithic applications can be refactored–often greatly increasing their scalability, lowering operating costs and significantly reducing time spent on database administration.

This document focuses on the migration scenario for existing applications. To help illustrate this process, we use an existing application named **Simple Customer** (http://www.simplecustomer.com) as an example, and use snippets of source code to illustrate specific aspects of the migration process. We chose Simple Customer because it is open source and relative simple. The entire application is available for further review (see the Appendix for details).

The Simple Customer Application

Simple Customer is an open-source customer relationship management application that was developed with PHP and MySQL. The Dashboard view is shown below:

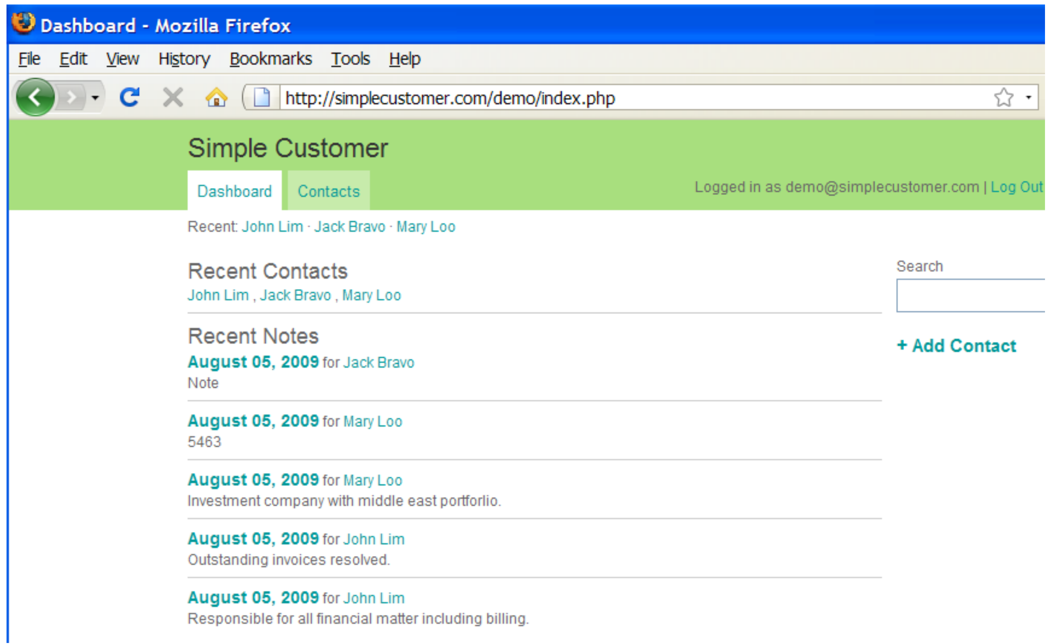


Figure 1 - Simple Customer Dashboard

It utilizes an architecture typical for web applications: the application code is hosted on a web server and the data is stored in MySQL. A high-level illustration of this architecture is shown below. Depending on scalability and performance requirements, these components may or may not reside on the same physical machines.

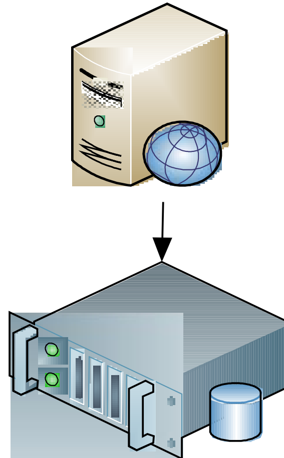


Figure 2 - Simple Customer high-level architecture

Fundamentally, this same basic architecture underlies many commercial software products and services.

The simplicity of this sample application is ideal for illustrating concepts; and the concepts examined in this document can be easily extrapolated for more complex, real-world scenarios.

Reasons for Migrating

Keeping a traditional relational database management system (RDMS) running–even on a small scale–is not a trivial undertaking. At a minimum, someone has to be responsible for monitoring the database, capturing backups and applying updates. On a larger scale, it is not unusual for a team to be assigned the responsibility of clustering and replicating a database to ensure that it scales appropriately.

All of this requires a significant investment in financial and human resources. Moreover, this investment must be made up front. Operational teams must be hired and hardware capacity must be purchased to satisfy the forecasted demand. The implicit inaccuracy of forecasting forces cautious organizations to overbuy capacity and over allocate resources.

However, almost any software application must be able to store and query for data, so the difficulties involved with operating an RDMS are inherent–something organizations traditionally have had to endure. Often, this financial and organizational overhead comes at the expense of innovation–and ultimately of creating more value for the customer.

Amazon SimpleDB is a hosted cloud-based web service that offers an alternative to traditional relational databases. This service takes a streamlined approach and provides just the core functionality needed to store and query data–all of the complex and obscure operations frequently found in a traditional database system are gone.

By virtue of its being XML-based, data can be rapidly stored–and easily retrieved or edited–through a simple set of web service API calls using any modern programming language and platform.

Your database resides in the Amazon Web Services cloud, so the complexity and expense of maintaining an in-house solution is eliminated. This enables you to focus on your unique, value-added application development, rather than on commoditized, tedious database administration.

By virtue of Amazon's cloud, Amazon SimpleDB automatically scales upward and downward to meet your incoming traffic; your requests are always served with a predictable level of performance.

Amazon SimpleDB utilizes the full spectrum of Amazon's high-availability data centers, so data stored in Amazon SimpleDB is geographically dispersed and automatically replicated, ensuring the availability and durability of your data.

Amazon SimpleDB provides highly flexible support for your current and future application development efforts.

In a traditional relational database, the smallest schema change can cascade across many aspects of your software development effort. With Amazon SimpleDB, you have a much more flexible and extensible attribute-based system. Even when attributes change, the system automatically indexes your data accordingly. This is because Amazon SimpleDB does not require predefined schemas.

The ability to store structured data without first defining a schema eliminates the need to refactor your database as your applications evolve.

Lastly, as with all of the Amazon Web Services products, you only pay for what you use. This frees you from many of the complexities of safety-net capacity planning, transforms large capital expenditures into much smaller operating costs and eliminates the need to overbuy capacity to handle periodic traffic spikes.

Migrating MySQL Schemas to Amazon SimpleDB

The schema that the original MySQL-driven version of Simple Customer uses is relatively simple; it comprises four tables, three of which are related using foreign keys. This schema is illustrated below.

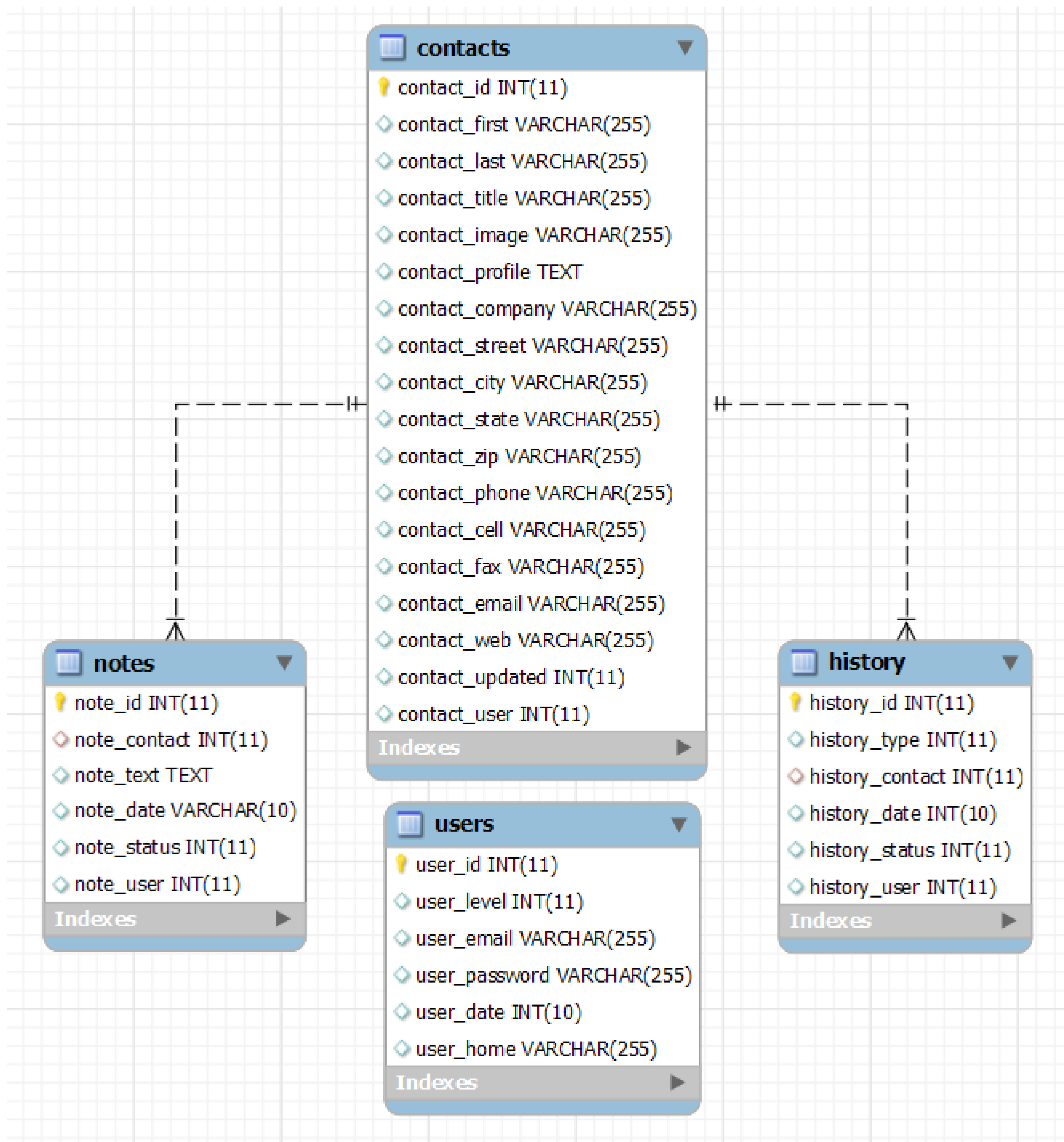


Figure 3 - Simple Customer Data Model

When porting the application backend from MySQL to Amazon SimpleDB, our primary concern is data integrity. Amazon SimpleDB does not enforce the use of schemas, and this makes data migration relatively simple.

The conceptual framework of Amazon SimpleDB data storage is intentionally streamlined and easy to learn:

- **Domains** help you organize your data.
- **Items** represent your data represented.
- **Attributes** are name-value pairs associated with items.

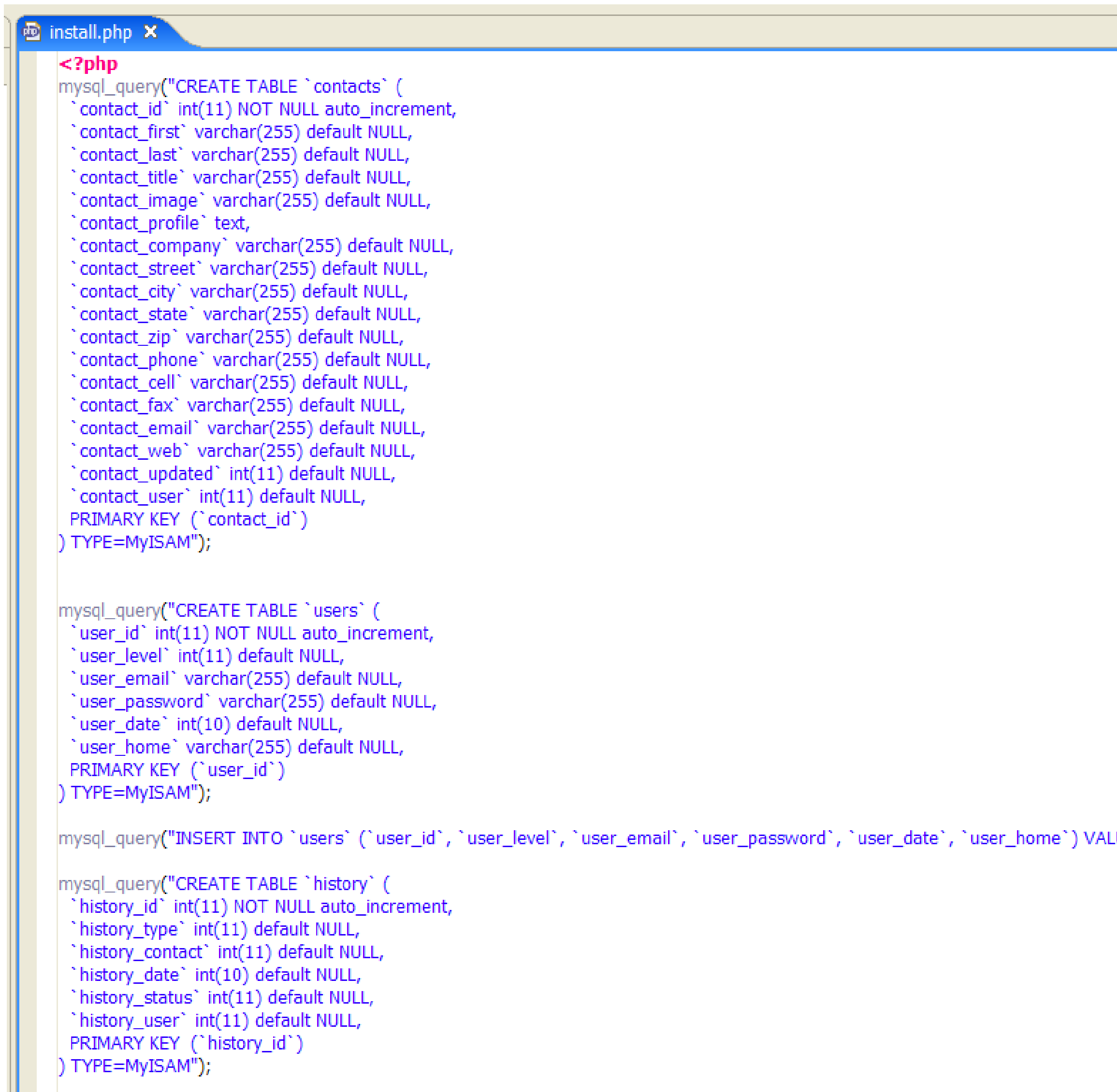
You query for items that match certain attribute values.

These concepts can be roughly matched to concepts in relational databases as illustrated in the following table.

Amazon SimpleDB	Traditional Relational Database
Domain	Table
Item	Row in a table
Attribute	Column value in a row

The use of domains and the schema-less nature of Amazon SimpleDB becomes apparent if you compare the source code for a MySQL-driven version of Simple Customer with an Amazon SimpleDB-driven version of Simple Customer.

Below is a snippet of source code from the **install.php** file in the original MySQL version of Simple Customer:



```
install.php
<?php
mysql_query("CREATE TABLE `contacts` (
 `contact_id` int(11) NOT NULL auto_increment,
 `contact_first` varchar(255) default NULL,
 `contact_last` varchar(255) default NULL,
 `contact_title` varchar(255) default NULL,
 `contact_image` varchar(255) default NULL,
 `contact_profile` text,
 `contact_company` varchar(255) default NULL,
 `contact_street` varchar(255) default NULL,
 `contact_city` varchar(255) default NULL,
 `contact_state` varchar(255) default NULL,
 `contact_zip` varchar(255) default NULL,
 `contact_phone` varchar(255) default NULL,
 `contact_cell` varchar(255) default NULL,
 `contact_fax` varchar(255) default NULL,
 `contact_email` varchar(255) default NULL,
 `contact_web` varchar(255) default NULL,
 `contact_updated` int(11) default NULL,
 `contact_user` int(11) default NULL,
 PRIMARY KEY (`contact_id`)
) TYPE=MyISAM");


mysql_query("CREATE TABLE `users` (
 `user_id` int(11) NOT NULL auto_increment,
 `user_level` int(11) default NULL,
 `user_email` varchar(255) default NULL,
 `user_password` varchar(255) default NULL,
 `user_date` int(10) default NULL,
 `user_home` varchar(255) default NULL,
 PRIMARY KEY (`user_id`)
) TYPE=MyISAM");

mysql_query("INSERT INTO `users` (`user_id`, `user_level`, `user_email`, `user_password`, `user_date`, `user_home`) VALI

mysql_query("CREATE TABLE `history` (
 `history_id` int(11) NOT NULL auto_increment,
 `history_type` int(11) default NULL,
 `history_contact` int(11) default NULL,
 `history_date` int(10) default NULL,
 `history_status` int(11) default NULL,
 `history_user` int(11) default NULL,
 PRIMARY KEY (`history_id`)
) TYPE=MyISAM");
```

Figure 4 – Install.php

These statements are used to create the MySQL tables illustrated previously in Figure 3; the column names and data types define the type of data these tables will store.

Any changes to the stored data necessitates changes to this code as well. Often ancillary software packages are used to keep schema conceptual changes synchronized with the source code.

The schema-less model underlying Amazon SimpleDB provides much more extensibility. For example, storing additional user-specified fields for a customer contact is trivial. You simply write application code to store additional attributes for each applicable item; there is no change needed to installation or deployment, and there is no possibility of breaking existing functionality.

Below is a snippet of source code from the Amazon SimpleDB-driven version of Simple Customer.

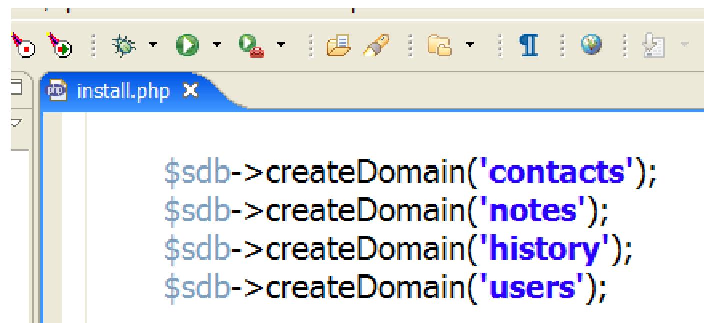


Figure 5 – Amazon SimpleDB code for Simple Customer

The only "installation" needed is to create the domains. The structure of the domains will be determined by the data that they will store. The operation to create a domain is **idempotent**, so there is no need for any additional logic to check whether the domain already exists.

In this example, a domain is created to match each table. But, because of the flexible nature of Amazon SimpleDB, it is entirely possible to use a single domain to store all of the data for this application. We have merely matched the number of domains and tables to clarify the example.

What may not have been immediately apparent in the two snippets of source code is that the way to programmatically access Amazon SimpleDB versus MySQL is quite different.

In order to access MySQL from a programming language like PHP, you need a driver. The driver acts as the interface between a high-level programming language like PHP, C# or Java, and the network protocol that MySQL understands.

MySQL is an extremely popular relational database management system, so there are drivers available for nearly every platform. Almost all default installations of PHP provide a preconfigured MySQL driver.

However, the need for a driver increases the complexity of the overall configuration. Also, ports need to be configured (port 3306 by default in this case) to permit MySQL network traffic. Again, although this is a simple and common procedure, it does add to the overall configuration taxonomy.

From a programmatic standpoint, Amazon SimpleDB is just a simple web service; it receives requests via HTTP (or HTTPS) and returns XML as a response. No special software or drivers are needed to interact with Amazon SimpleDB. This is a subtle, yet important benefit.

Database drivers are often a source of complexity and error during software deployment. It is not uncommon to accidentally use different versions of a database driver in the development and production environments. This simple mistake can introduce difficult-to-reproduce errors into your application—complicating your debugging efforts.

With Amazon SimpleDB, this complexity and risk is eliminated because you are simply using a web service API.

Sending HTTP requests and parsing the XML response is a simple operation for most programming languages, but it can be somewhat tedious.

For the sake of convenience, it is common to use libraries that offer predefined, language-specific wrappers for the Amazon SimpleDB operations. The wrapper is essentially a thin layer of software that performs the HTTP invocations and subsequent XML parsing.

Amazon Web Services has a large and vibrant developer community. The open source wrapper used for the Amazon SimpleDB-driven version of Simple Customer (http://sourceforge.net/projects/php-sdb/) was produced within this community.

It is important to emphasize that this particular example uses PHP/MySQL, but Amazon SimpleDB–and the rest of the Amazon Web Services–is completely platform agnostic. Any platform or language that can send HTTP requests and process XML responses can use the Amazon Web Services.

Migration from a CRUD Perspective

Migrating from MySQL to Amazon SimpleDB requires subtle changes in how data is created, read, updated and deleted; this series of activities is commonly referred to with the acronym CRUD.

Creating or Inserting Data

The data stored in Amazon SimpleDB is represented as **items** and these **items** have **attributes**, which define their value. From a RDMS perspective, items and attributes are roughly analogous to the rows and columns in a table, respectively.

In the Simple Customer application, you create new customer contacts by entering data in fields to complete a form as illustrated below:

Figure 6 - Creating a new contact

The fields from this form are used to create each new contact. In the original MySQL-driven version of Simple Customer, these fields are used to create an INSERT statement. The following source code is from the **contact.php** source file.

```
{
    mysql_query("INSERT INTO contacts (contact_first, " .
                                       "contact_last, " .
                                       "contact_title, " .
                                       "contact_image, " .
                                       "contact_profile, " .
                                       "contact_company, " .
                                       "contact_street, " .
                                       "contact_city, " .
                                       "contact_state, " .
                                       "contact_zip, " .
                                       "contact_phone, " .
                                       "contact_cell, " .
                                       "contact_email, " .
                                       "contact_web, " .
                                       "contact_updated) VALUES
    (
    '".trim(addslashes($_POST['contact_first']))."',
    '".trim(addslashes($_POST['contact_last']))."',
    '".trim(addslashes($_POST['contact_title']))."',
    '".addslashes($picture)."',
    '".trim(addslashes($_POST['contact_profile']))."',
    '".trim(addslashes($_POST['contact_company']))."',
    '".trim(addslashes($_POST['contact_street']))."',
    '".trim(addslashes($_POST['contact_city']))."',
    '".trim(addslashes($_POST['contact_state']))."',
    '".trim(addslashes($_POST['contact_zip']))."',
    '".trim(addslashes($_POST['contact_phone']))."',
    '".trim(addslashes($_POST['contact_cell']))."',
    '".trim(addslashes($_POST['contact_email']))."',
    '".trim(addslashes($_POST['contact_web']))."',
    '".time()."'
    )
    ");
```

Figure 7 - INSERT SQL Statement

The Amazon SimpleDB approach is somewhat similar; attributes are used to represent the fields we are storing for the customer contact.

Based on the domain and attributes specified, the invocation of the **putAttributes** operation actually stores the data in Amazon SimpleDB.

```
$attributes = array(  "contact_id"       => array("value" => $id, "replace" => "false"),
                      "contact_first"    => array("value" => $first_name, "replace" => "false"),
                      "contact_last"     => array("value" => $last_name, "replace" => "false"),
                      "contact_title"    => array("value" => $title, "replace" => "false"),
                      "contact_street"   => array("value" => $street, "replace" => "false"),
                      "contact_city"     => array("value" => $city, "replace" => "false"),
                      "contact_state"    => array("value" => $state, "replace" => "false"),
                      "contact_zip"      => array("value" => $zip,  "replace" => "false"),
                      "contact_phone"    => array("value" => $phone,  "replace" => "true"),
                      "contact_cell"     => array("value" => $cell,  "replace" => "false"),
                      "contact_email"    => array("value" => $email,   "replace" => "false"),
                      "contact_web"      => array("value" => $web, "replace" => "false"),
                      "contact_update"   => array("value" => $update, "replace" => "false"),
                      "contact_profile"  => array("value" => $profile,  "replace" => "false"),
                      "contact_company"  => array("value" => $company,  "replace" => "false"));

$sdb->putAttributes('contacts', $id, $attributes);
```

Figure 8 - PutAttributes in Amazon SimpleDB

Amazon SimpleDB attributes are a simple, yet very powerful concept. The **replace** parameter in the definition of an attribute determines whether the incoming value updates an existing value or not. Since this is an insert operation, the replace value is set to false.

The last line of the code snippet actually stores the contact into Amazon SimpleDB by invoking the **putAttributes** API.

Notice that **putAttributes** accepts three parameters: a domain name, an item name and the attributes. The item name is specified by the **$id** parameter, and it illustrates a unique characteristic of Amazon SimpleDB.

In a traditional relational database, each row that is inserted into a table is assigned a unique identifier (usually a numeric value). This sounds like an innocuous operation, but it has a negative impact on scalability. The RDMS has to ensure that each incoming row gets a unique value. This becomes difficult when rows are being inserted in parallel. Typically, locking algorithms are used to ensure that duplicate identifiers are not given out; but any kind of locking, no matter how small, negatively impacts scalability.

Amazon SimpleDB chooses a much simpler approach. By shifting the responsibility of creating a unique identifier to the application code, the creation of a unique value is a trivial operation for any programming language. In this example, we can use the intrinsic PHP function **uniqid** to generate a unique identifier.

```
{
  if ((isset($_POST["MM_insert"])) && ($_POST["MM_insert"] == "form1"))
  {
    $id = uniqid();
    $first_name = trim(addslashes($_POST['contact_first']));
```

Figure 9 – uniquid function

For the purpose of this simple example, the uniqueness that this function guarantees is suitable. For more stringent applications, you could use a UUID instead.

When you are migrating existing data from a relational database to Amazon SimpleDB, it is common to simply reuse the identifier that was generated for you. This is a common approach when migrating existing data to Amazon SimpleDB (as illustrated in the Appendix). Reuse will likely result in different patterns for your identifiers, but this is merely cosmetic. Amazon SimpleDB simply requires a unique identifier for each item you store. It is immaterial whether that identifier follows a pattern or not.

By shifting the responsibility for uniquely identifying data, Amazon SimpleDB can provide scalability efficiently. The tradeoff is negligible, because generating a unique value is a trivial procedure for nearly all programming languages.

Reading or Querying Data

Like any reasonably sized application, Simple Customer needs relatively complex SQL queries to implement its features. The query illustrated below searches for contacts:

```
if (isset($_GET['s']))
{
   $cwhere =   "WHERE contact_first LIKE '%".addslashes($_GET['s'])."%' OR " .
               "contact_last LIKE '%".addslashes($_GET['s'])."%' OR contact_email " .
               "LIKE '%".addslashes($_GET['s'])."%' OR contact_company LIKE '%".addslashes($_GET['s'])."%'";
}
```

Figure 10 - SQL to search for contacts

Amazon SimpleDB employs a SQL-like query syntax, so in many cases, exactly the same SQL phrasing is used to query both MySQL and Amazon SimpleDB.

Thus, for Simple Customer, we used the relatively complex SQL illustrated in Figure 10 as-is for Amazon SimpleDB. This greatly reduced the effort needed to perform the migration.

Amazon SimpleDB imposes, however, a few minor requirements on the dialect of SQL that it understands. The most obvious is how ordering is specified. A snippet of the source code that builds the SQL ordering clauses in the original version of Simple Customer is listed below. This code was taken from the **contacts.php** file:

```
contacts.php
<?php require_once('includes/config.php');
include('includes/sc-includes.php');
$pagetitle = Contact;

//SORTING
$name = "name_up";
if (isset($_GET['name_up'])) {
$sorder = "ORDER BY contact_last ASC";
$name = "name_down";
} elseif (isset($_GET['name_down'])) {
$sorder = "ORDER BY contact_last DESC";
}

$email = "email_up";
if (isset($_GET['email_up'])) {
$sorder = "ORDER BY contact_email ASC";
$email = "email_down";
} elseif (isset($_GET['email_down'])) {
$sorder = "ORDER BY contact_email DESC";
}
```

Figure 11 – Ordering clauses

Amazon SimpleDB understands the **ORDER BY** clause, but it needs to specify the attribute being ordered in the **WHERE** clause as well. The solution is to simply add a **WHERE** clause for this attribute that always evaluates to `true`—as in the following example.

```
contacts.php
$pagetitle = Contact;

//
// Sort queries
//
$name = "name_up";
if (isset($_GET['name_up'])) {
$sorder = "WHERE contact_last != '' ORDER BY contact_last ASC";
$name = "name_down";
}
elseif (isset($_GET['name_down'])) {
    $sorder = "WHERE contact_last != '' ORDER BY contact_last DESC";
}
```

Figure 12 - Amazon SimpleDB ordering

This simple addition allows us to reuse the rest of the SQL with Amazon SimpleDB.

Iterating through Results

MySQL and Amazon SimpleDB return results from querying in different ways. Typically, when using MySQL, it is common to retrieve the results from a query one row at a time.

The source code listed below is from the **contacts.php** file of the original version of Simple Customer.

One row of results–one customer contact in this case–is retrieved each time through this loop. The call to the **mysql_fetch_assoc** method retrieves the next set of results.

```
<?php do { $row_count++; ?>
    <tr <?php if ($row_count%2) { ?>bgcolor="#F4F4F4"<?php } ?>>
      <td style="padding-left:5px"><a href="contact-details.php?id=<?php echo $row
      <td><?php echo $row_contacts[contact_phone] ? $row_contacts['contact_phoı
      <td><a href="mailto:<?php echo $row_contacts['contact_email']; ?>"><?ph
      <td>
        <input name="d[<?php echo $row_contacts['contact_id']; ?>]" type="checkb
      </td>
    </tr>
    <?php } while ($row_contacts = mysql_fetch_assoc($contacts)); ?>
```

Figure 13 - Iterating through MySQL results

Amazon SimpleDB returns the results of a query differently, so small changes in interpreting those results are required.

Amazon SimpleDB returns the complete set of results[1] from the **SELECT** operation. In the following code snippet, all of the results from the query are contained in the **$query_contacts** variable.

```
}

$query_contacts = "SELECT * FROM contacts $sorder";

$row_contacts = $sdb->select("contacts", $query_contacts);
$totalRows_contacts = count($row_contacts);
```

Figure 14 - Results from Amazon SimpleDB

1 Amazon SimpleDB limits the response to a **SELECT** operation to 1 MB in size. If the response is larger than 1 MB, the response will also include a token. This token can be passed to additional requests to retrieve the rest of the data. This is similar to the paging functionality that many database drivers provide. In the Simple Customer case, for the sake of simplicity the assumption was made that the results would not exceed this limit.

The format of the XML response is formally defined by its WSDL (http://sdb.amazonaws.com/doc/2009-04-15/AmazonSimpleDB.wsdl).

An informal example of how the response is structured follows.

```
<SelectResponse  xmlns="http://sdb.amazonaws.com/doc/2009-04-15">
  <SelectResult>
    <Item>
      <Name>Item_03</Name>
      <Attribute>
        <Name>Category</Name>
        <Value>Clothes</Value>
      </Attribute>
      <Attribute>
        <Name>Subcategory</Name>
        <Value>Pants</Value>
      </Attribute>
      <Attribute>
        <Name>Name</Name>
        <Value>Sweatpants</Value>
      </Attribute>
      <Attribute>...
      <Attribute>...
      <Attribute>...
      <Attribute>...
    </Item>
    <Item>...
  </SelectResult>
  <ResponseMetadata>
    <RequestId>b1e8f1f7-42e9-494c-ad09-2674e557526d</RequestId>
    <BoxUsage>0.0000219907</BoxUsage>
  </ResponseMetadata>
</SelectResponse>
```

Figure 15 – Sample response

It is up to the wrapper to decide how these XML results are deserialized, but most wrappers, including the one used in this case, choose a dictionary or hash table structure to store the results.

More specifically, the **$row_contacts** variable is a list of dictionaries, with each dictionary representing a customer contact and its attributes. The code below illustrates how to index the results returned by Amazon SimpleDB to display the email addresses of each customer contact.

```
</td>
<?php $row_count = 0; do {?>
   <tr <?php if ($row_count%2) { ?>bgcolor="#F4F4F4"<?php } ?>>
     <td style="padding-left:5px"><a href="contact-details.php?id=<?php echo $row_contacts[$row_count]['
     <td><?php echo $row_contacts[$row_count]['Attributes'][contact_phone] ? $row_contacts[$row_count
     <td><a href="mailto:<?php echo $row_contacts[$row_count]['Attributes']['contact_email']; ?>"><
     <td>
       <input name="d[<?php echo $row_contacts[$row_count]['Attributes']['contact_id']; ?>]" type="che
     </td>
   </tr>
   <?php $row_count++; } while ($row_count < $totalRows_contacts); ?>
```

Figure 16 - Indexing into Amazon SimpleDB results

The HTML formatting from the original Simple Customer was preserved during the migration; the only material changes made were in how the results were accessed.

Working with JOINs

As illustrated earlier in Figure 3, the original MySQL-driven Simple Customer schema uses a foreign-key relationship between the **contacts, notes,** and **history** tables.

There is no concept of a JOIN in Amazon SimpleDB. This is done in order to increase the scalability and performance of SimpleDB. However, if your existing MySQL application uses JOINs, determining how to accommodate this difference requires some thought during migration.

In many cases, the use of multi-value attributes is enough to enable the scenarios where a JOIN is usually used. The data you normally would store in a related table could be de-normalized into multiple values in a single attribute. However, in the Simple Customer case, a tuple of data is kept for each note and history that is associated with a contact, so denormalizing with multi-value attributes is not practical. For our Simple Customer migration, we simply write application code to simulate the JOIN operation.

In the MySQL-driven version of Simple Customer, a JOIN query returns a list of the notes being stored and their associated customer information.

```
mysql_select_db($database_contacts, $contacts);
$query_notes = "SELECT * FROM notes INNER JOIN contacts ON note_contact = contact_id " .
               "$nwhere ORDER BY note_date DESC LIMIT 0, 20";
```

Figure 17 - JOIN contacts and notes

In the Amazon SimpleDB-driven version, an additional query is needed to retrieve the customer contact information associated with a given note as illustrated below.

```
foreach ($row_notes as $note)
{ ?>
   <div <?php if ($note['Attributes']['note_date'] > time()-1) { ?>id="newnote"<?php }
      <span class="datedisplay">
      <a href="contact-details.php?id=<?php echo $note['Attributes']['contact_id']; ?>&
      <?php
         $contact_id = $note['Attributes']['contact_id'];
         $contact_attributes = $sdb->getAttributes('contacts', $contact_id);

         echo $contact_attributes['contact_first'];
      ?> 
```

Figure 18 - Simulating a JOIN

In this example, instead of a **SELECT** query, the **getAttributes** operation is used to retrieve the attributes of an item. Since the item name ($contact_id) is known, it is possible to directly access its data using **getAttributes**. This is more efficient than using a query operation.

One of the components of the pricing model for Amazon SimpleDB is the amount of time it takes to process your requests, so there is a financial benefit to using Amazon SimpleDB efficiently.

Whenever you know the item name, you can save time by using **getAttributes** to retrieve the attributes for that data.

Updating Data

Data in Amazon SimpleDB is created and updated with the **putAttributes**[2] operation. The dual use of this operation is relatively unique compared to the usual update pattern with a traditional RDMS.

The most interesting application of the **putAttributes** operation is in the context of the import feature in Simple Customer. The data imported could represent new customer contacts, or updates to existing customer contacts, so the Simple Customer application must decide whether to insert or update data.

2Multiple items can be created or updated using the **BatchPutAttributes** operation. This operation is conceptually the same as **putAttributes**; it just works on up to 25 items (a batch) rather than on an individual level.

Figure 19 - Importing contacts

The following code (abbreviated for clarity) is located in the **batch.php** source file of the original Simple Customer application.

```
$checkc = mysql_num_rows(mysql_query("SELECT * FROM contacts WHERE contact_id = ".$data[0].""));

if ($checkc > 0) {

   //
   // UPDATE CURRENT RECORDS
   //
}
else {
   if ($row > 1)
   {
      //
      // INSERT NEW RECORDS
      //
```

Figure 20 - MySQL update pattern

This pattern is common when using a traditional RDMS like MySQL. A query is first executed to determine if the data given represents an existing item or not. The most efficient way to execute this query is to specify that only the number of results should be returned. If the number of results is 0, then the data indicates that a new customer contact should be inserted. Otherwise, the data should be used to update an existing customer contact.

Amazon SimpleDB makes this type of logic much simpler. The **putAttributes** operation requires a domain name, an item name and a set of attributes as its parameters. If the item name already exists in the domain, then the attributes are used to update the existing item. Otherwise, the item is inserted into the domain. The **replace** flag specified for each attribute enables you to further define how to treat the incoming data. If the **replace** flag is `false`, then new data will be appended to existing data; if the **replace** flag is `true`, then the new data will replace the existing data.

The logic structure from the original MySQL-driven Simple Customer is replaced with a single invocation of the **putAttributes** operation—Amazon SimpleDB does the work necessary to determine if this is an update or insertion.

```
$sdb->putAttributes('contacts', $data[0], $attributes);
```

Figure 21 - Insert or update data

Deleting Data

Amazon SimpleDB uses the **deleteAttributes** operation to update or delete items. Items in a domain do not have to share the same set of attributes, so you can invoke **deleteAttributes** to remove attributes for individual items.

If **deleteAttributes** is specified with no attributes, or if an item is left with no attributes, then that item is deleted completely. Unlike **putAttributes**, there is no batch operation for **deleteAttributes**. Each item must be deleted individually. In some cases, where mass deletions of data are common, it is more efficient to simply delete and re-create the domain.

These semantics are quite different than what MySQL offers and must be taken into account during a migration. We can illustrate this difference by the way customer contacts are deleted from Simple Customer.

In the original MySQL-driven version, the **DELETE** operation is used to delete the contact fields, notes and history for a given customer contact. This is a very simple operation.

The snippet of source code illustrated below is taken from the **delete.php** file.

```
mysql_query("DELETE FROM contacts WHERE contact_id = ".$_GET['contact']."");
mysql_query("DELETE FROM history WHERE history_contact = ".$_GET['contact']."");
mysql_query("DELETE FROM notes WHERE note_contact = ".$_GET['contact']."");
```

Figure 22 - DELETE operation

Application code must be written for the Amazon SimpleDB-driven version of Simple Customer, and items must be deleted individually from the **contacts, notes** and **history** domains.

```
$histories = $sdb->select('history', "SELECT * FROM history WHERE history_contact = '$contact_id'");

foreach ($histories as $history)
{
   $sdb->deleteAttributes('history', $history['Name']);
}

$notes = $sdb->select('notes', "SELECT * FROM notes WHERE contact_id = '$contact_id'");
foreach ($notes as $note)
{
   $sdb->deleteAttributes('notes', $note['Name']);
}

//
// Now we can delete the contact
//
$sdb->deleteAttributes('contact', $contact_id);
```

Figure 23 - Deleting items from Amazon SimpleDB

Additional Benefits of Migrating to Amazon SimpleDB

Hopefully this document has made it easier to plan the migration of your application to Amazon SimpleDB and illuminated some of the unique considerations that arise when using the service.

Migrating to Amazon SimpleDB has a number of benefits in addition to those enumerated earlier in this document. For instance, Amazon SimpleDB is always referenced the same way–it does not matter if the web application aspect to Simple Customer resides in a local datacenter, in Amazon Web Service, or in another hosting provider. As long as there is Internet access, the web application can communicate with Amazon SimpleDB.

Conclusion

Amazon SimpleDB does not duplicate every feature found in a traditional RDMS like MySQL, but there are significant benefits in terms of availability, reliability, and scalability.

In order to avoid the performance bottlenecks that often accompany centralized databases, Amazon SimpleDB focuses on delivering only the core functionality needed for storing and working with structured data, while ensuring dynamic scalability at low cost.

When complete, the work involved in migrating your data store from MySQL to Amazon SimpleDB will also allow you to enjoy the benefits of a highly available managed service. With data modeling, index maintenance, redundancy and performance tuning taken care of, you can place your focus squarely on application development.

Appendix

Source Code

The source code for this application is hosted at: http://code.google.com/p/sdb-simple-customer/

Updates will be made to this location periodically.

Amazon Machine Image

This application is also available in Amazon EC2 as an AMI (**ami-e14aab88**).

This AMI expects to receive an Amazon access key id and secret through the launch user data in the following format:

<your key>***<your secret>

Once the instance is running, request the **install.php** page to create the necessary Amazon SimpleDB domains.

Migrating Simple Customer data

The Simple Customer application has a feature to export existing data into a comma separated values (CVS) file format. This can be done using the **Export** link, which is found on the **Contacts** tab of the application.

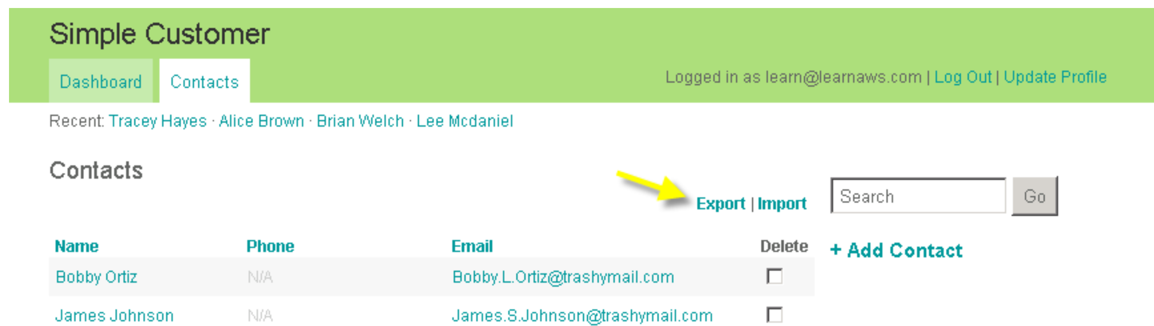


Figure 24 – Contacts tab

The CSV file format is ideal because it is typically easy to parse. In general, most MySQL administration utilities have a way to export a database to a CSV or similar file format.

The code necessary to parse the CSV format from Simple Customer is relatively generic; the PHP intrinsic function **explode** is used to parse each comma separate value.

The code shown below omits error-checking for the sake of brevity.

```
<?php
$file = fopen('C:\simplecustomer.csv', 'r');

while (!feof($file))
{
    $line = fgets($file);

    //
    // Read each contact
    //
    $contact = explode(",", $line);

    $id         = str_replace("\"", '', $contact[0]);
    $firstName  = str_replace("\"", '', $contact[1]);
    $lastName   = str_replace("\"", '', $contact[2]);
    $street     = str_replace("\"", '', $contact[5]);
    $city       = str_replace("\"", '', $contact[6]);
    $state      = str_replace("\"", '', $contact[7]);
    $zip        = str_replace("\"", '', $contact[8]);
    $email      = str_replace("\"", '', $contact[9]);
```

Figure 25 - Parsing CSV

Once the contact values have been extracted, migrating the data over to Amazon SimpleDB is simply a matter of invoking the **putAttributes** API function.

```
$attributes =  array("contact_id"   => array("value" => $id, "replace" => "false"),
                     "contact_first" => array("value" => $firstName, "replace" => "false"),
                     "contact_last" => array("value" => $lastName, "replace" => "false"),
                     "contact_street" => array("value" => $street, "replace" => "false"),
                     "contact_city" => array("value" => $city, "replace" => "false"),
                     "contact_state" => array("value" => $state, "replace" => "false"),
                     "contact_zip" => array("value" => $zip,    "replace" => "false"),
                     "contact_email" => array("value" => $email,    "replace" => "false"));
//
// Insert into our contacts domain
//
print "Adding $id \r\n";
$sdb->putAttributes('contacts', $id, $attributes);
```

Figure 26 - Migrating data

The migration code is so simple because Amazon SimpleDB uses a unified type system, i.e. all data is stored as strings. Whatever format that MySQL chose to export your data will be retained in Amazon SimpleDB. This makes your migration easier because it minimizes application logic changes.