



# Programmer's Guide: Full-Text Search Over RDBMS Data

A Lucid Imagination White Paper

---



© 2010 by Lucid Imagination, Inc. under the terms of Creative Commons license, as detailed at <http://www.lucidimagination.com/Copyrights-and-Disclaimers/>. Version 1.02, published 6 June 2010. Solr, Lucene and their logos are trademarks of the [Apache Software Foundation](#).

## Abstract

As data expands in the rate of its accumulation and the variety of its formats, many developers turn to databases to store it all -- including, more often than not, unstructured, non-tabular text in documents of different shapes and sizes.

While many common databases such as Oracle and MySQL offer Full Text Search functionality, going beyond a simple keyword search to provide the search experience users have come to expect can require complex programming. Often, the problems of querying text content can be much more easily solved using a search platform such as Apache Solr, the Lucene Search Server.

This paper provides pragmatic examples that will show a database veteran how to think a little differently about search with Solr, and how to get started on a real search application that produces useful search results, both by using conventional SQL-driven programming, and by using Solr to provide more powerful results with less programming effort. Readers should have a basic understanding of database programming. Samples are provided in PHP, but are applicable to any programming language.

## Table of Contents

Introduction .....	1
The Allure Of The Database.....	1
What Users Expect When They Search.....	1
The Bells and Whistles That Make All the Difference .....	2
To Database Or Not To Database, That's Not A Question.....	2
What We're Going To Cover.....	3
What You Need To Get Started .....	3
Our Example.....	5
Choosing To Go The RDBMS Way .....	5
Using Flat Text Documents Instead .....	8
The Best Of Both Worlds:	
Using Solr's DataImportHandler To Index A Database For Searching.....	12
Just The Facts: Creating A Simple Search Routine .....	18
Searching Against One Field Using SQL.....	19
Searching Against One Field Using Solr.....	22
Searching Against Multiple Fields Using SQL.....	25
Searching Against Multiple Fields Using Solr.....	29
Faceted Searching.....	31
Summary .....	38
Next Steps .....	39
Appendix: Lucene/Solr Features and Benefits.....	40

## Introduction

Back in the 1990's and early 2000's, when most of the data a user encountered through his or her browser was created specifically for the web, heavy use of a site's search function was considered a failure of the site's navigation. After all, if the navigation was clear, users wouldn't need to resort to searching, would they?

These days, making that argument would be difficult, if not impossible. For one thing, the sheer volume of information that is now available through the browser – whether publicly or just internally within a specific organization – has grown exponentially over the last decade. At the same time, users have gotten used to navigating via search engine; they are accustomed to having information come to them via that little search box, rather than poking around trying to find it themselves.

But taken together, these two trends can provide a significant challenge when it comes to making information available.

### *The Allure Of The Database*

Today it's not just about making information available on the web; it's about making it available anywhere. And for many developers and information managers, that means only one method of storage will do: a database. Storing information in a relational database management system provides advantages such as the ability to easily provide data in additional forms, including for mobile environments, a single environment for backup and management, and, of course, the ability to search. But often, this means adding items that wouldn't traditionally be considered RDBMS data, such as documents and other unstructured information.

### *What Users Expect When They Search*

The conventional wisdom used to be that the behavior of a user looking for information was straightforward. He or she either browsed to the intended target, or entered a simple search into a search box and went on to the desired content from there. Today's reality, however, is much different.

Today's users typically find information in a process known as "berrypicking". Just as one picks berries by taking a few here and a few there while wandering through the berry patch, users tend to start in one place, pick a bit of information here, a fact there, tweak their search terms based on what they see, and so on. The search they're using has to accommodate this behavior, because if users can't find what they're looking for quickly – usually within the first page or two of results – they assume the information isn't there and move on to another site.

### *The Bells and Whistles That Make All the Difference*

Fortunately, there are things you can do to help users find the information they're looking for more easily. These include:

- 1) Presenting more relevant results first
- 2) Providing better search results by carefully analyzing both the user's query and the data being searched to avoid "noise"
- 3) Enabling faceted searching, so that users can gradually narrow their search based on various attributes, such as category, price, and so on
- 4) Assisting the user in the creation of the query through the use of spell-checking and autocomplete boxes that provide possible choices from the actual indexed data
- 5) Leading the user to "related" results that don't necessarily fit the exact query, but are probably relevant anyway

### *To Database Or Not To Database, That's Not A Question*

Of course, all of these bells and whistles take programming effort, but how much programming effort depends on how you go about it.

Some databases, such as MySQL and Oracle, now offer the ability to do text analysis in order to provide somewhat relevant results, but the traditional method of executing an SQL statement and displaying the results begins to get complicated once you start dealing with different forms of unstructured data. Also, most of the other bells and whistles users expect will need to be implemented "by hand", so to speak.

A search platform such as Solr/Lucene eliminates many of these issues by simplifying search, and by providing many of these functions out of the box, but it's often overlooked

by programmers dealing with an RDBMS-centric system because we're used to thinking of it only in terms of static documents on the file system.

In fact, that's not the case. Solr includes a `DataImportHandler` that easily indexes data right from your RDBMS, so that you can provide all of the advantages of a search platform without having to give up your database.

In this paper, we'll look at creating some of the type of search capabilities users have come to expect by using both the traditional SQL-based methodology and an installation of Solr.

### *What We're Going To Cover*

In this paper, we're going to cover the following topics:

- 1) The difference between storing data in a database and indexing it with Solr. We'll also demonstrate the use of the `DataImportHandler` to index data that already exists in a database.
- 2) Creating a simple search routine. We'll start with a single one-field search and work our way up from there, first using SQL, and then Solr.
- 3) Improving the user experience with faceted search. Here we'll look at what it would take to create a faceted search using Solr. Users will be able to narrow results down by multiple categories, such as type of information and price range.

The general techniques we cover here will also give you a toe-hold when it comes to using Solr for features such as autocomplete and related searches.

### *What You Need To Get Started*

To build the application discussed in this paper, you will need a JDBC compliant database, a PHP-enabled web server and an installation of Apache Solr. The examples use the following:

- MySQL 5.x (A different database will do, as long as you have the JDBC driver for it). You can download MySQL 5.1 at <http://dev.mysql.com/downloads/mysql/>, with installation instructions at <http://dev.mysql.com/doc/refman/5.1/en/installing.html>.

- XAMPP Lite, which provides an Apache web server with PHP support. You can download XAMPP for Windows at <http://www.apachefriends.org/en/xampp-windows.html>, or for other platforms at [http://en.wikipedia.org/wiki/List of AMP packages](http://en.wikipedia.org/wiki/List_of_AMP_packages).
- Solr. The easiest way to install Solr is to download LucidWorks for Solr from <http://www.lucidimagination.com/Downloads/LucidWorks-for-Solr> and execute the \*.jar file to run the installer.

In order to keep the code examples fairly simple, we're using PHP for the sample application, but the beauty of this particular stack is that you can use any programming language, as long as you have the proper drivers for the database. (Solr is HTTP-based, so the programming language doesn't matter.)

If you intend to follow along with the examples, make sure that these tools (or whichever you choose to use) are installed and tested before you begin the next section.

In order to keep the code examples fairly simple, we're using PHP for the sample application, but the beauty of this particular stack is that you can use any programming language, as long as you have the proper drivers for the database. (Solr is HTTP-based, so the programming language doesn't matter.) Just keep in mind that these examples are written for simplicity and readability; you'll want to use these concepts with your own best practices with regard to security, unit testing, and so on.



## Our Example

To demonstrate the difference between using traditional SQL-based methods and using a search solution, we'll be creating a simple site that provides information on various products in different categories. This information can consist of both data traditionally housed in an RDBMS, such as product information displayed on a web page, and data traditionally presented as raw documents, such as user manuals and brochures.

### *Choosing To Go The RDBMS Way*

If you were to store all of this data in a traditional RDBMS, the structure might look something like this:

```
CREATE TABLE IF NOT EXISTS `products` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `product_name` varchar(255) NOT NULL,
  `product_price` double NOT NULL,
  `product_adcopy` text NOT NULL,
  `product_description` text NOT NULL,
  PRIMARY KEY (`id`)
)
CREATE TABLE IF NOT EXISTS `documents` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `cat_id` int(11) NOT NULL,
  `prod_id` int(11) NULL,
  `doc_type_id` int(11) NOT NULL,
  `document_title` varchar(255) NOT NULL,
  `document_desc` text NOT NULL,
  `document_text` text NOT NULL,
  `document_url` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
)
CREATE TABLE IF NOT EXISTS `categories` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `category_name` varchar(255) NOT NULL,
  PRIMARY KEY (`id`)
);
CREATE TABLE IF NOT EXISTS `category_products` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `cat_id` int(11) NOT NULL,
  `prod_id` int(11) NOT NULL,
```

```
        PRIMARY KEY (`id`)
    );
CREATE TABLE IF NOT EXISTS `document_types` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `document_type_name` varchar(255) NOT NULL,
    PRIMARY KEY (`id`)
);
```

#### Creating the basic SQL tables necessary for the application

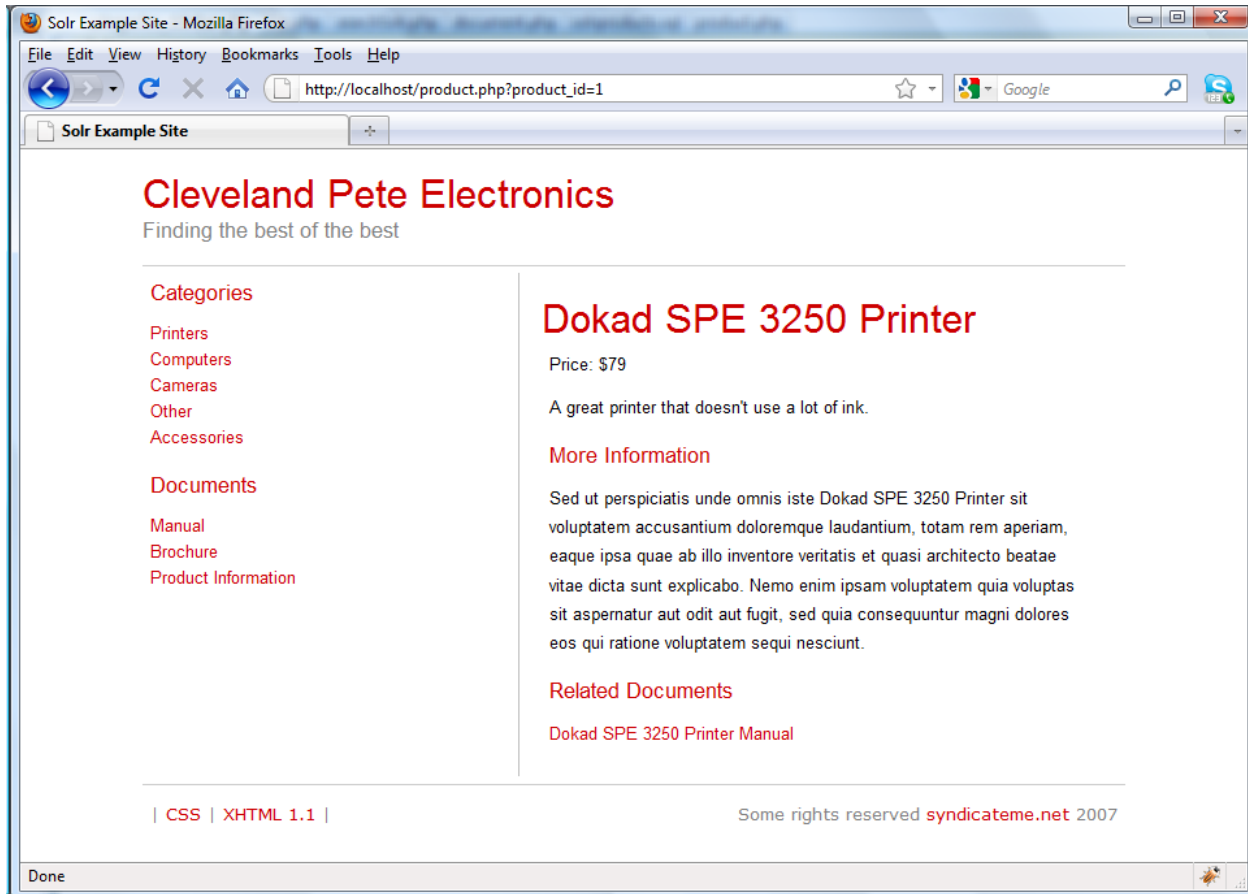
To keep things as simple as possible, we've included just the most basic information. For a product, we have the name, price, a basic "blurb," or introduction to the item (useful for pages that link to it, or, more significantly, for search results) and the full text describing the item.

For documents, we're including the document title and a description, but we're also including the full text of the document itself. This text is never meant to be displayed on the web site, but if you're going to be able to search it using SQL, it's got to be in the database. We're also including a link to the actual document, as well as information about the product to which it refers and the category to which it belongs so that later, we can pull up documents alongside products in a particular category.

Finally, we've got three auxiliary tables that help to organize the data: the `categories` themselves, `category_products`, so that products (unlike documents) can belong to more than one category, and `document_types`, which provides human-readable names for values found in the `documents.doc_type_id` column.

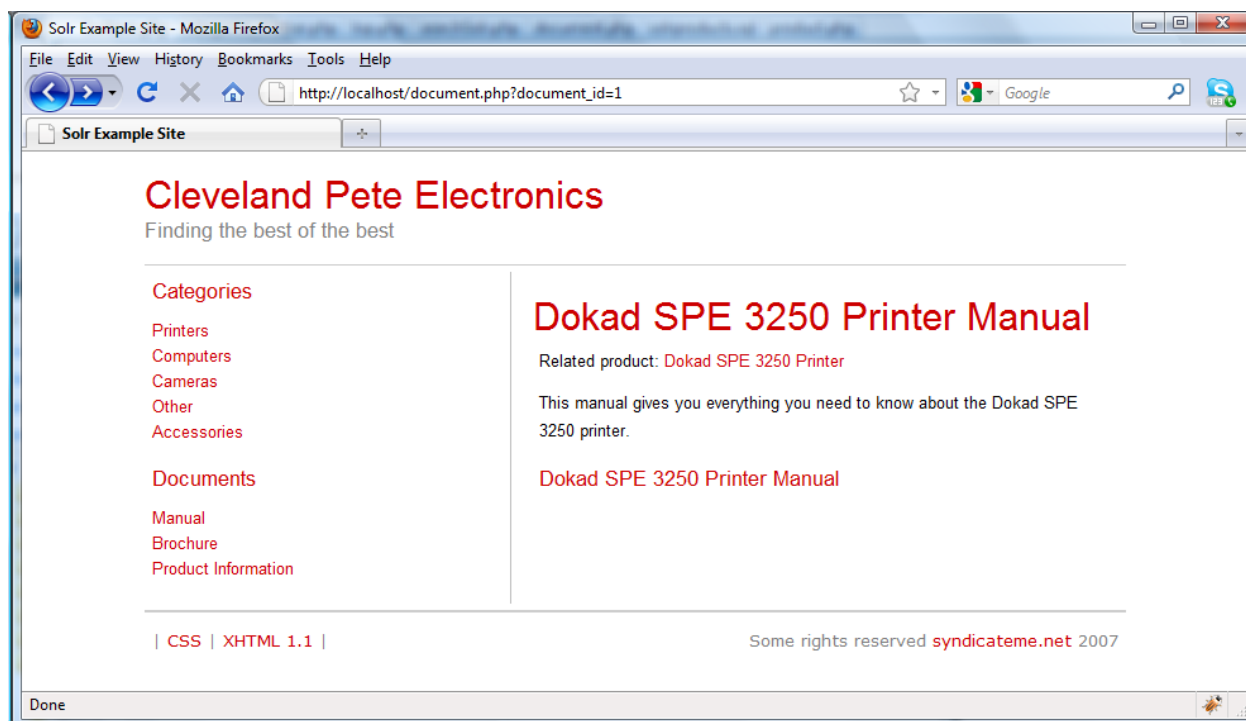
(If you're following along, I've included sample data in the code file download.)

Taken all together, we have everything we need to create both a product display page...



A sample product page created from the database

... and a document display page:



A sample document page created from the database

The document page doesn't display the document itself; instead, it links to the document based on the URL in the database.

As a web developer, everything we've covered so far is probably familiar to you. Now let's look at how a search solution such as Solr looks at data.

### *Using Flat Text Documents Instead*

Rather than including the text of your documents within the database, which could be, at the very least, somewhat problematic, it would be much more convenient to be able to simply store your documents elsewhere, but index the parts of them you'd like to search.

While Solr is designed to index and search unstructured information such as documents, we still need to provide information as to what that data represents. Is this text part of the title? The price? The general body text? In order for any search solution to provide

relevant results, it needs know what type of information, or “field”, each bit of text belongs to.

So our first task would be to define a schema for the data, just as we did in the database example. To do that, we can add the field definitions to the `schema.xml` file, found in Solr’s `conf` directory:

```
<field name="id" type="string" indexed="true" stored="true"
      required="true" />
<field name="itemId" type="string" indexed="true" stored="true" />
<field name="itemType" type="string" indexed="true" stored="true" />
<field name="docType" type="string" indexed="true" stored="true" />
<field name="cat" type="text_ws" indexed="true" stored="true"
      multiValued="true" omitNorms="true" />
<field name="catId" type="string" indexed="true" stored="true"
      multiValued="true" />
<field name="name" type="text" indexed="true" stored="true" />
<field name="price" type="tfloat" indexed="true" stored="true" />
<field name="blurb" type="text" indexed="true" stored="true" />
<field name="docTypeRaw" type="string" indexed="true"
      stored="true" />
<field name="description" type="text" indexed="true"
      stored="true" />
<field name="documentUrl" type="string" indexed="true"
      stored="true" />
<field name="text" type="text" indexed="true" stored="false"
      multiValued="true"/>
```

**Fields added to the sample Solr schema.**

(For Lucidworks, you’ll find this file in `<install_dir>/lucidworks/solr/conf`.)

The `schema.xml` file defines the various types (though a discussion of them is beyond the scope of this paper) and fields. Basically we’re telling Solr we have three types of data: `text` items, which should be analyzed, broken into tokens, and so on, `string` items, which should be indexed and stored verbatim, and a single `tfloat` item, which is a number that can be searched for in a range.

For each of these fields, we also let Solr know what to do with the data. For most of them, we want to index the field, or make it available to search on, and store it. Fields that are stored are returned with search results. So in this case, we’re indexing the field called `text`, but we’re not storing it, so it won’t be returned with the rest of the results.

The reason we're doing that is that `text` is our "default" search field, and includes the data from most of the other fields. This way, we can perform a single search and cover all of our bases. To make that happen, we tell Solr to copy the information from our fields into `text`, like so:

```
<copyField source="cat" dest="text"/>
<copyField source="docType" dest="text"/>
<copyField source="name" dest="text"/>
<copyField source="blurb" dest="text"/>
<copyField source="description" dest="text"/>
```

Finally, we need to tell Solr to use `text` as the default search field, and to use the `id` field as our unique identifier so we can avoid duplicate documents when we update the index with changes to the database:

```
<uniqueKey>id</uniqueKey>
<defaultSearchField>text</defaultSearchField>
```

All right, now we've defined our schema, so it's time to actually add the data to the index. One way to do that is to provide the information in XML format, as in:

```
<add>
<doc>
  <field name='id'>prod2</field>
  <field name='itemId'>2</field>
  <field name='itemType'>product</field>
  <field name='docType'>Product Information</field>
  <field name='docTypeRaw'>3</field>
  <field name='catId'>3</field>
  <field name='catId'>4</field>
  <field name='cat'>Cameras</field>
  <field name='cat'>Accessories</field>
  <field name='name'>Kinok UltraCam</field>
  <field name='price'>300</field>
  <field name='blurb'>Boy, the Kinok UltraCam is a great camera, and it
hooks up to your printer terrifically.</field>
  <field name='description'> Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore
magna aliqua...</field>
</doc>
<doc>
  <field name='id'>doc1</field>
```

```
<field name='itemId'>1</field>
<field name='itemType'>document</field>
<field name='docType'>User Guide</field>
<field name='docTypeRaw'>1</field>
<field name='catId'>1</field>
<field name='cat'>Printers</field>
<field name='name'>Dokad SPE 3250 Printer Manual</field>
<field name='documentUrl'>dokad3250guide.pdf</field>
<field name='blurb'>This manual gives you everything you need to know
about the Dokad SPE 3250 printer.</field>
<field name='description'> Sed ut perspiciatis unde omnis iste natus
error sit voluptatem accusantium doloremque laudantium, totam rem aperiam,
eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae
vitae dicta sunt explicabo.</field>
</doc>
</add>
```

The sample database data, exported for indexing by Solr.

Now, I want to digress for a moment. First off, these files aren't your authoritative data store, of course; this is just the information fed to Solr in order to enable indexing of that data. Second, while you could create a script or application to export the data to an XML file such as this one, we're going to discuss a better way to do that in a moment. For now we just want to look at the actual data.

But before we talk about the structure of this data, there's one very question we need to address. As you can see, this data is pretty specific to our application. What happens if you have actual documents, such as Word files or PDFs? Fortunately, this is not a problem. Solr includes Apache Tika, which extracts both the content and metadata for various types of files into a format that Solr can read. A complete discussion of Tika is beyond the scope of this paper, but you can find more information at <http://www.lucidimagination.com/Community/Hear-from-the-Experts/Articles/Content-Extraction-Tika>.

Now let's talk about the actual data. For simplicity's sake, we've included just one document for each type of data, but we can still get a good feel for how Solr looks at this information.

First off, notice that everything is enclosed in a `add` element. This tells Solr what to do with the enclosed data. As you might guess, this means you can also delete data by feeding an XML file (or even just a string of XML data) to Solr.

Next, note that all of our data is in `field` elements, with the name of the field specified in an attribute, just as we specified in the schema. Notice that we're keeping the database `id` column in the `itemId` field, and prefacing Solr's `id` field with the type of item; this is to make sure that the `id` field is always unique, even if a product and document share an `id` value in the database.

Note also that the `cat` field is a "multi-valued" field, so we can include more than one for a single document.

At this point in the process, we typically would tell Solr to index the data by POSTing it using the `post.jar` that came with Solr, and you're welcome to do that. But while you certainly could create a PHP routine that exports the data and saves it to the file system for import, there's a better way.

### *The Best Of Both Worlds:*

#### *Using Solr's DataImportHandler To Index A Database For Searching*

While manually exporting your database and indexing the resulting files would work, and may be sufficient for databases with a low volume of data and transactions, most enterprise systems need something more direct, and that's where Solr's DataImportHandler comes in.

The DataImportHandler enables you to index your data directly from the database without creating intermediary XML. Let's look at how it works.

The first step is to tell Solr you want to use the DIH. To do that, open the `solrconfig.xml` file, found in Solr's `conf` directory. This file includes all of the information on how Solr operates, including the query parser specification and the definition of various request handlers, such as those for spell checking. In this case, we're going to define a new request handler at the bottom of the file:

```
...
<requestHandler name="/productimport"
  class="org.apache.solr.handler.dataimport.DataImportHandler">
  <lst name="defaults">
    <str name="config">
      data-config-prods.xml
    </str>
  </lst>
</requestHandler>
</config>
```



Here we're creating the URL for the request handler and telling Solr to look for information on what to index in the `data-config-prods.xml` file, also in the `conf` directory. Now we need to create that information.

The `data-config-prods.xml` document is a simple XML text file containing the following:

```
<dataConfig>
  <dataSource type="JdbcDataSource"
    driver="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost/solrProducts"
    user="solr"
    password="solr"/>

  <document>
    <entity name="outerprod"
      query="select id, id as documentId, 'product' as itemType,
        'Product Information' as docType, '3' as doc_type_id,
        product_name, product_price, product_adcopy,
        product_description from products"
      transformer="TemplateTransformer">

      <field column="documentId" name="id"
        template="prod${outerprod.documentId}" />
      <field column="id" name="itemId"/>
      <field column="itemType" name="itemType" />
      <field column="docType" name="docType" />
      <field column="doc_type_id" name="docTypeRaw" />

      <entity name="innerprod"
        query="select cat_id, category_name from
          category_products, categories where categories.id =
            cat_id and prod_id = ${outerprod.id}">

        <field column="cat_id" name="catId"/>
        <field column="category_name" name="cat"/>
      </entity>
      <field column="product_name" name="name"/>
      <field column="product_price" name="price"/>
      <field column="product_adcopy" name="blurb"/>
      <field column="product_description" name="description"/>
    </entity>

    <entity name="outerdoc">
```

```

query="select id, id as documentId, 'document' as itemType,
        doc_type_id, document_title, document_url,
        document_desc, document_text from documents"
transformer="TemplateTransformer">

<field column="documentId" name="id"
        template="doc${outerdoc.documentId}" />
<field column="id" name="itemId"/>
<field column="itemType" name="itemType"/>

<entity name="innerdoccat"
        query="select id, category_name from categories where
                id = ${outerdoc.id}">
    <field column="id" name="catId"/>
    <field column="category_name" name="cat"/>
</entity>
<field column="doc_type_id" name="docTypeRaw" />
<entity name="innerdocdoctype"
        query="select document_type_name from document_types where
                id = ${outerdoc.doc_type_id}">
    <field column="document_type_name" name="docType"/>
</entity>
<field column="document_title" name="name"/>
<field column="document_url" name='documentUrl'/>
<field column="document_desc" name="blurb"/>
<field column="document_text" name="description"/>
</entity>
</document>
</dataConfig>

```

The data-config-prods.xml file tells the DataImptrHandler how to handle the data in the RDBMS.

We start by defining the datasource. In my case, it's a MySQL database running locally and called solrProducts. (Note that in order to make sure Solr could find the driver, I had to place the mysql-connector-java-5.1.6-bin.jar file in Solr's lib directory at <install\_dir>/lucidworks/solr/lib.)

Next, we'll define the two entities, products and documents, that we want to import. In each case, we specify a query to be run against the database, and then we specify fields based on columns returned by that query. To deal with inner queries, such as those to get category names, we can define an inner entity and reference the outer entity by name.

Also, to prefix the id values to keep them unique over the entire index, we're using a TemplateTransformer. The transformer appends our text (prod or doc) to the data coming from the query (\${outerprod.documentId} or \${outerdoc.documentId}).

Now restart Solr to pick up the changes to `solrconfig.xml`, and we're ready to go.

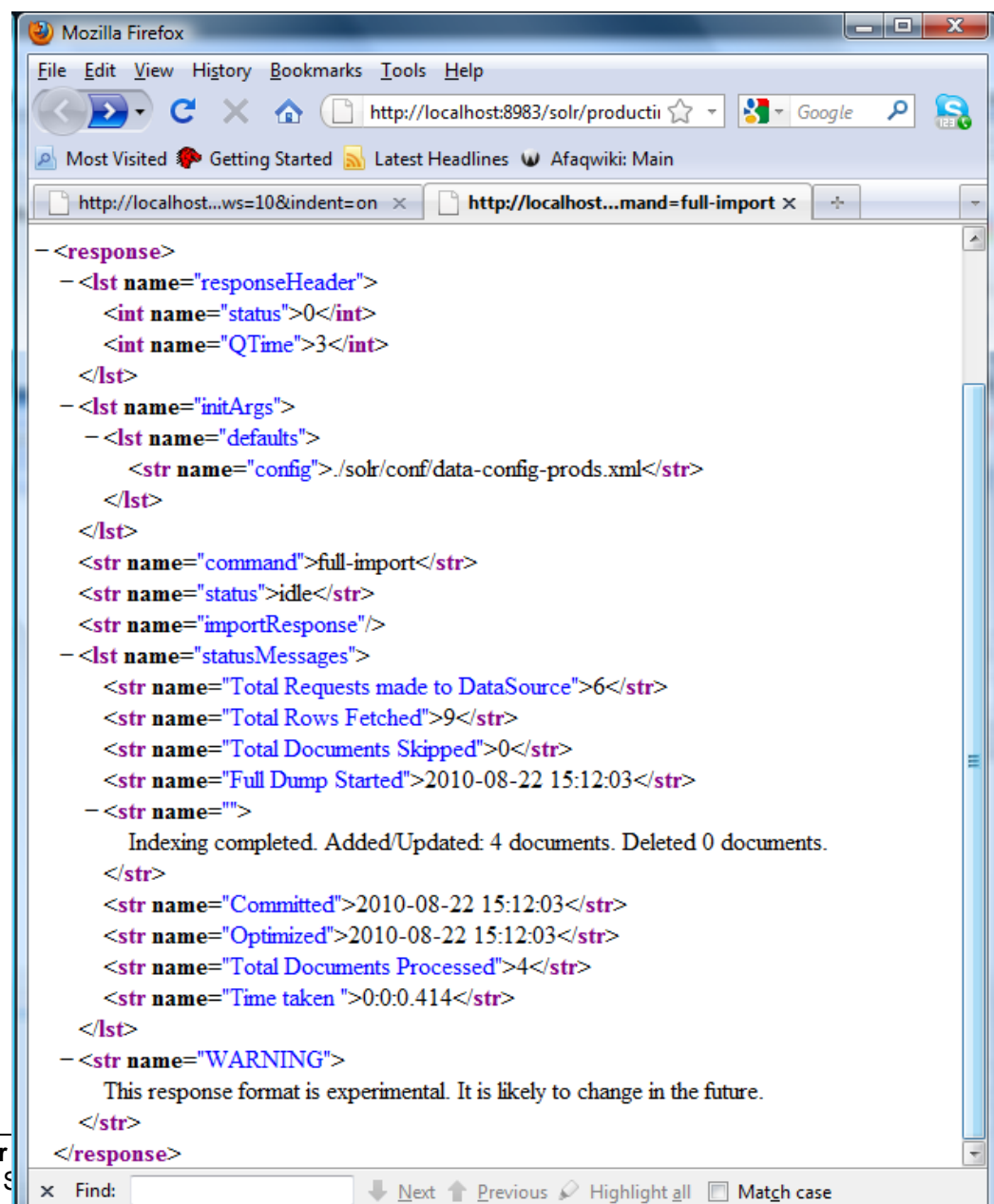
In order to complete the import, we'll need to tell Solr to run the `DataImportHandler`. We can do that from the browser by calling:

<http://localhost:8983/solr/productimport?command=full-import&clean=false>

Assuming that you've done a default installation of Solr, it will be running on port 8983, in the `solr` webapp, so in this case we're calling the `productimport` request handler with the `full-import` command, and telling Solr not to clean up the index by deleting existing documents. If all goes well, you should see a response something like this:

Running the `DataImportHandler` provides feedback via XML in the browser.

Full-Text Search Over  
Programmer's Guide • 5



```
<?xml version="1.0"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">3</int>
  </lst>
  <lst name="initArgs">
    <lst name="defaults">
      <str name="config">./solr/conf/data-config-prods.xml</str>
    </lst>
  </lst>
  <str name="command">full-import</str>
  <str name="status">idle</str>
  <str name="importResponse"/>
  <lst name="statusMessages">
    <str name="Total Requests made to DataSource">6</str>
    <str name="Total Rows Fetched">9</str>
    <str name="Total Documents Skipped">0</str>
    <str name="Full Dump Started">2010-08-22 15:12:03</str>
  </lst>
  <str name="">
    Indexing completed. Added/Updated: 4 documents. Deleted 0 documents.
  </str>
  <str name="Committed">2010-08-22 15:12:03</str>
  <str name="Optimized">2010-08-22 15:12:03</str>
  <str name="Total Documents Processed">4</str>
  <str name="Time taken ">0:0:0.414</str>
</lst>
<str name="WARNING">
  This response format is experimental. It is likely to change in the future.
</str>
</response>
```



Once the import has started, you can find out about its progress by visiting:

<http://localhost:8983/solr/productimport?command=status>

We can make sure that the data's been imported by running a simple generic query against the index. We'll talk more about queries in the next section, but for now, simply open this URL in your browser: [http://localhost:8983/solr/select?q=\\*. \\*](http://localhost:8983/solr/select?q=*.)

You should see a response something like this:



A simple query shows how many documents were imported.

In my case there were a total of four records in the database, and I can see that all four of them are returned by the query.

The DataImportHandler also enables you to update existing data using the `delta-import` command rather than `full-import`. (For more information on using `delta-import`, see [http://wiki.apache.org/solr/DataImportHandler#Using\\_delta-import\\_command](http://wiki.apache.org/solr/DataImportHandler#Using_delta-import_command).)

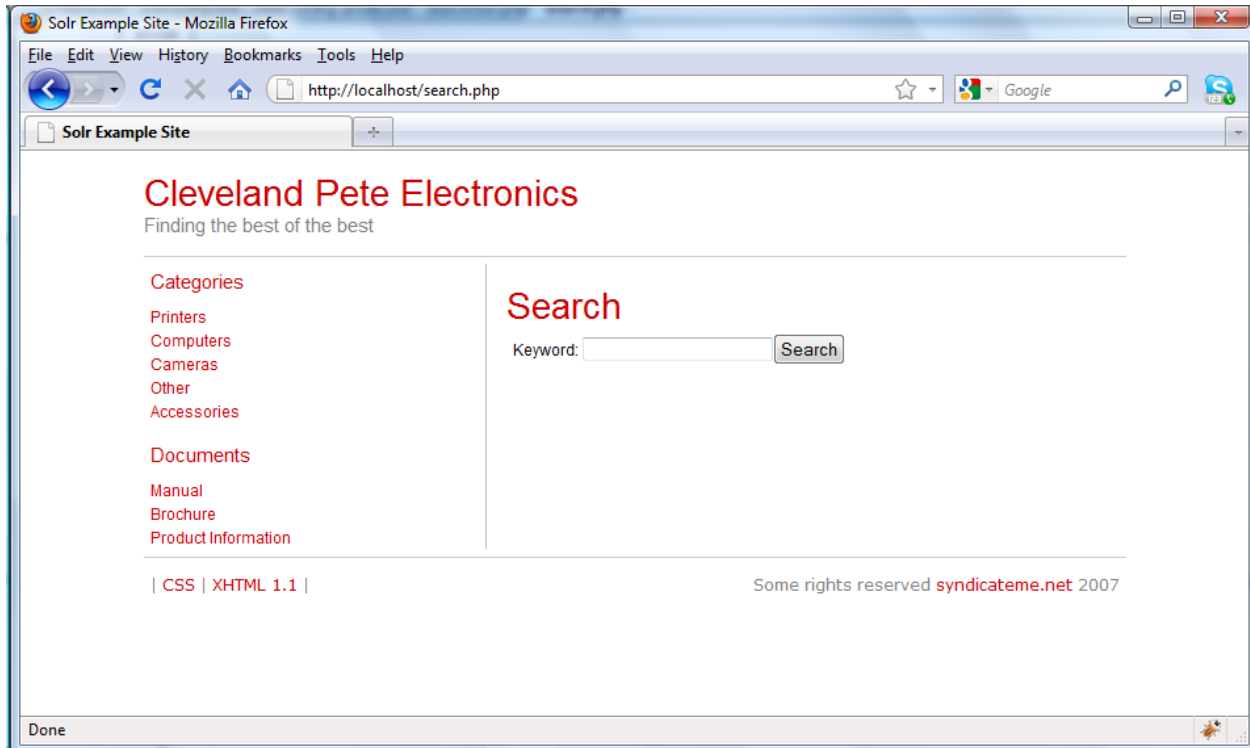
The DataImportHandler caches configuration information, so if you make changes to `data-config-prods.xml`, you'll need to reload it using the `reload-config` command so that Solr picks up the changes; like so:

<http://localhost:8983/solr/productimport?command=reload-config>

All right, now that we've got our data indexed and ready, let's look at creating a basic search routine.

## Just The Facts: Creating A Simple Search Routine

The purpose of all that indexing was, of course, to enable the user to find our data, so now let's look at what it takes to build a simple, basic search routine, first using SQL, and then using Solr. We'll start with a simple search page:



A simple search page provides a foundation for us

Now let's make it work.

## Searching Against One Field Using SQL

These days, creating a simple page that searches against a database is a fairly well-understood problem. In PHP, you can accomplish it in a straightforward way:

```
<?php
include("../includes/top.php");
?>
<h1>Search</h1>
<form action='search.php' method='get' id='searchForm'>
<p>Keyword: <input type='text' name='keyword' id='keyword' />
      <input type='submit' value='Search' class='submitButton' /></p>
</form>
<?php
if (isset($_GET['keyword'])) {
```

```
$term = $_GET['keyword'];
$resultsFound = false;

$SQL = "SELECT * FROM products where (product_name like
      '%".mysql_real_escape_string ($term)."% ' or product_adcopy like
      '%".mysql_real_escape_string ($term)."% ' or product_description
      like '%".mysql_real_escape_string ($term)."%')";
$results = mysql_db_query($db, $SQL, $cid);
if (!$results) { echo( mysql_error()); }

while ($row = mysql_fetch_array($results)) {
    $prodId = $row["id"];
    $prodName = $row["product_name"];
    $prodDesc = $row["product_description"];
    echo ("<h2><a href='products.php?id=$prodId'>$prodName</a></h2>");
    echo ("<p>$prodDesc</p>");
    $resultsFound = true;
}

$SQL = "SELECT * FROM documents where (document_title like
      '%".mysql_real_escape_string($term)."% ' or document_desc like
      '%".mysql_real_escape_string($term)."% ' or document_text like
      '%".mysql_real_escape_string($term)."%')";
$results = mysql_db_query($db, $SQL, $cid);
if (!$results) { echo( mysql_error()); }

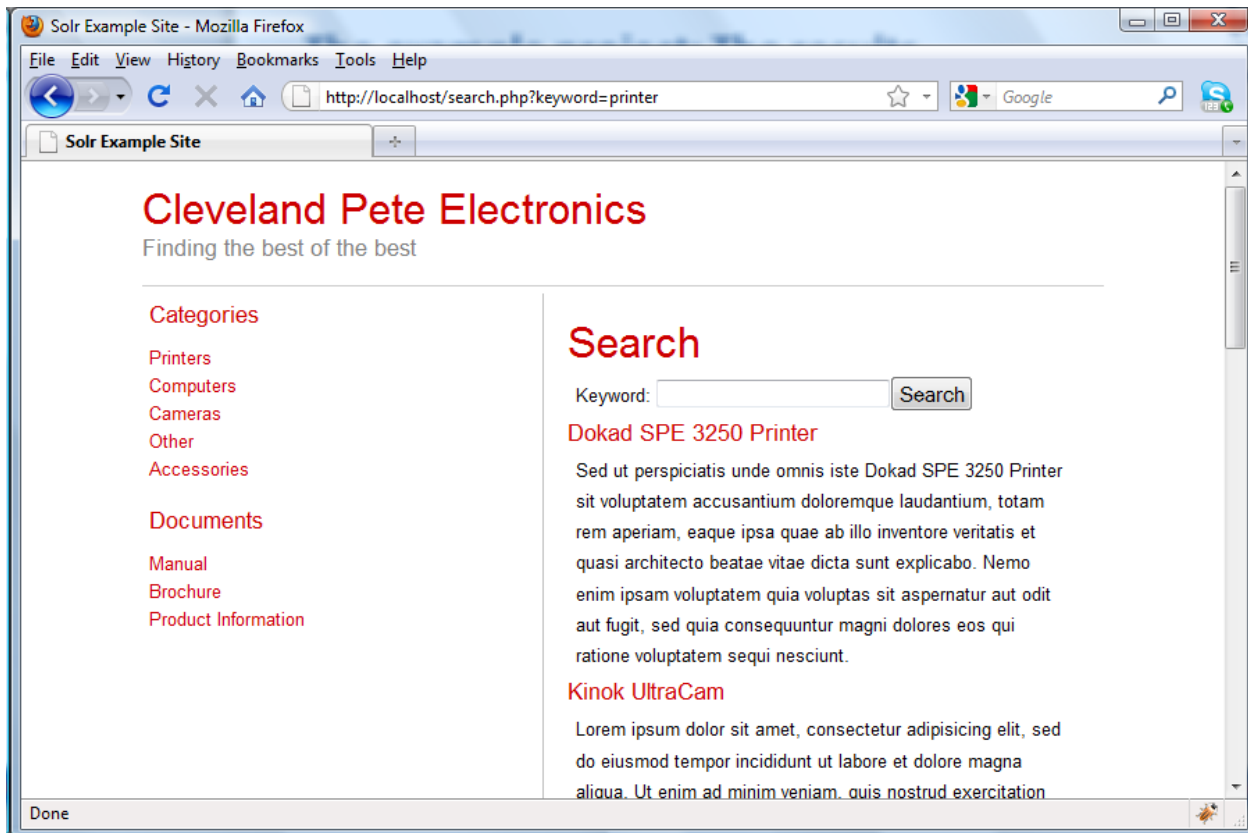
while ($row = mysql_fetch_array($results)) {
    $docId = $row["id"];
    $docTitle = $row["document_title"];
    $docDesc = $row["document_desc"];
    echo ("<h2><a href='document.php?id=$docId'>$docTitle</a></h2>");
    echo ("<p>$docDesc</p>\n");
    $resultsFound = true;
}
if (!$resultsFound){
    echo "<h2>No results found.</h2>";
}
}
include("../includes/bottom.php");
?>
```

Searching for a single term using SQL involves building a query for each table



Note again that this is obviously not necessarily the best way to structure a script like this; all of the code in this paper is designed for simplicity and readability; if this were a real application you'd be using objects, error checking, and so on.

I've moved all the extraneous details such as connecting to the database out into the include files, but here you can see we're simply talking about executing a query and displaying the results, making sure to escape the text entered by the user to prevent SQL injection attacks. If you load the page and search for "printer", you'll see something like this:



#### A simple search gives us simple results

The page works, which is nice, but it has a few limitations and complications.

- First off, since we're dealing with two different types of data in two different tables, we need to search them separately. You might be able to perform some SQL contortions to get a single query out of it, but with the possibility of many-to-one relationships, a simple join is out of the question, and a solution is certainly not something that comes off the top of your head. Consider also that we're just dealing with two types of information in this example; in a real environment, you're likely to have many more to deal with.
- Second, the actual query has been hard-coded in; without changing the code, the only thing you're going to be able to do here is search for a text string within the name (or title) and descriptions. And if your table definitions change, your display code will very likely break.
- Third, since we need to process each table separately, we need to process both queries before we even know whether there are any results, and the results themselves are segregated by type rather than being intermingled. That means that even if the database could provide results sorted by relevance, the final results won't be able to take advantage of that capability.

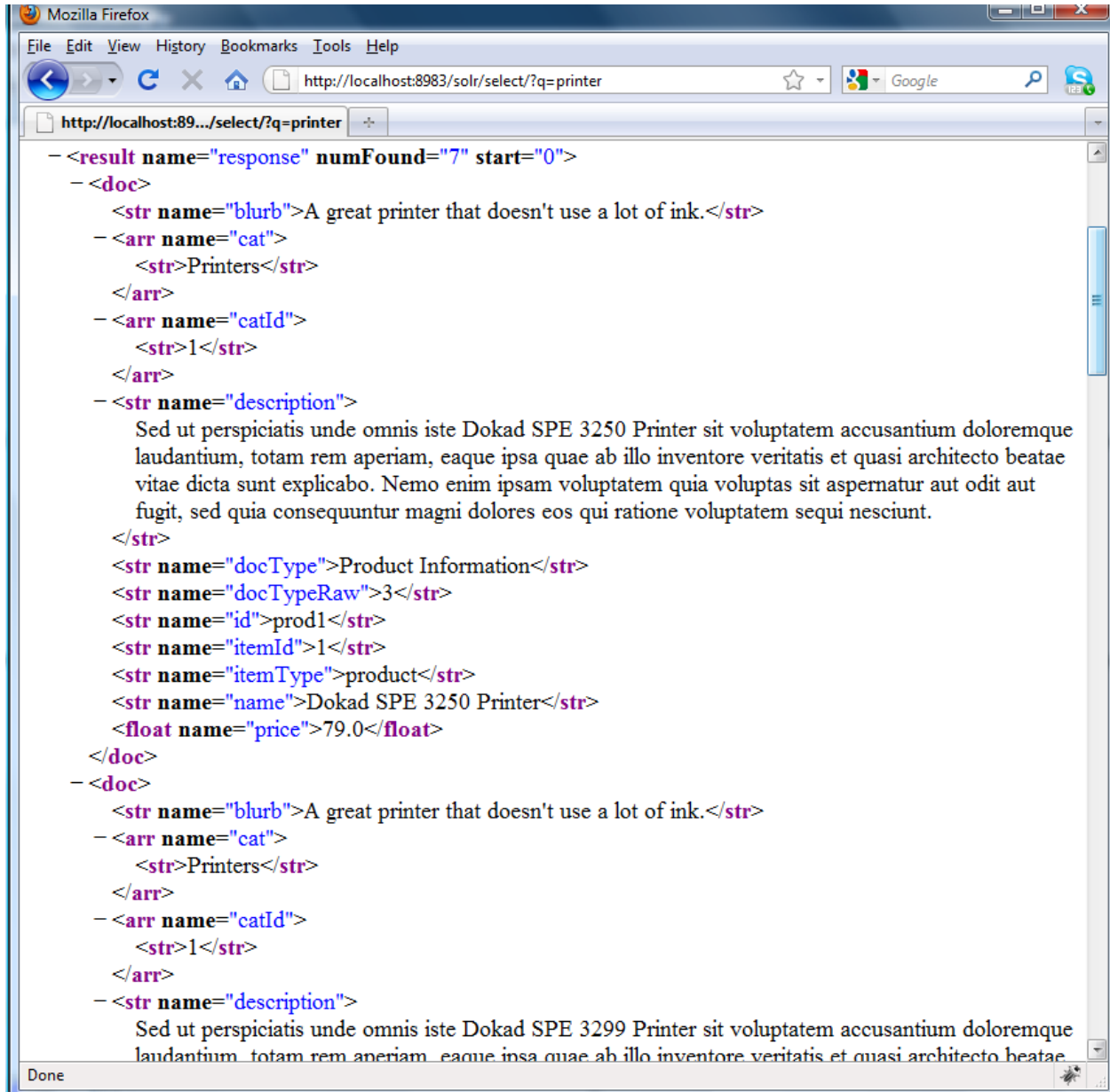
### *Searching Against One Field Using Solr*

So what does the programming for this same page look like using Solr?

First off, it's important to understand that Solr results are simply data returned over HTTP. So to see the results of searching for "printer", we can simply enter

<http://localhost:8983/solr/select/?q=printer>

into the browser. The results are something like this:



Searching via Solr results in an XML response

Notice that for each result, we get a doc element with a series of `str` (for strings) or `arr` (for array) elements – one for each field we told Solr to store when we indexed the document.

I've shown the results this way for two reasons: first, it's the default manner in which Solr provides results, and second, it's easy to see exactly what the results are. But working with XML directly in your results isn't always the best way. Fortunately, Solr provides other options by way of the `wt` parameter. So by changing our query to

<http://localhost:8983/solr/select/?q=printer&wt=json>

we get our results back in a JSON object, which we can then act on as an object from within PHP, Javascript, or virtually any other scripting language.

So let's take that information and create the search results page, as follows:

```
<?php
include("../includes/top.php");
?>
<h1>Search</h1>
<form action='searchSolr.php' method='get' id='searchForm'>
<p>Keyword: <input type='text' name='keyword' id='keyword' />
           <input type='submit' value='Search' class='submitButton' /></p>
</form>
<?php

if (isset($_GET['keyword'])) {
    $term = $_GET['keyword'];

    $jsonurl =
        "http://localhost:8983/solr/select/?q=".urlencode($term)."&wt=json";
    $json = file_get_contents($jsonurl,0,null,null);
    $json_output = json_decode($json);

    $response = $json_output->response;
    $resultsFound = $response->numFound;

    if ($resultsFound > 0) {

        echo ("<h2>$resultsFound results found</h2><br />");

        foreach($response->docs as $thisDoc){
            $pageName = $thisDoc->itemType;
            $itemId = $thisDoc->itemId;
```

```
$name = $thisDoc->name;
$desc = $thisDoc->description;

echo ("<h2>".
      "<a href='". $pageName. ".php? ". $pageName. "_id=$itemId'>".
      "$name</a></h2>");
echo ("<p>$desc</p>");
}
} else {
    echo "<h2>No results found</h2>";
}
}
include("../includes/bottom.php");
?>
```

#### The Solr-based search page

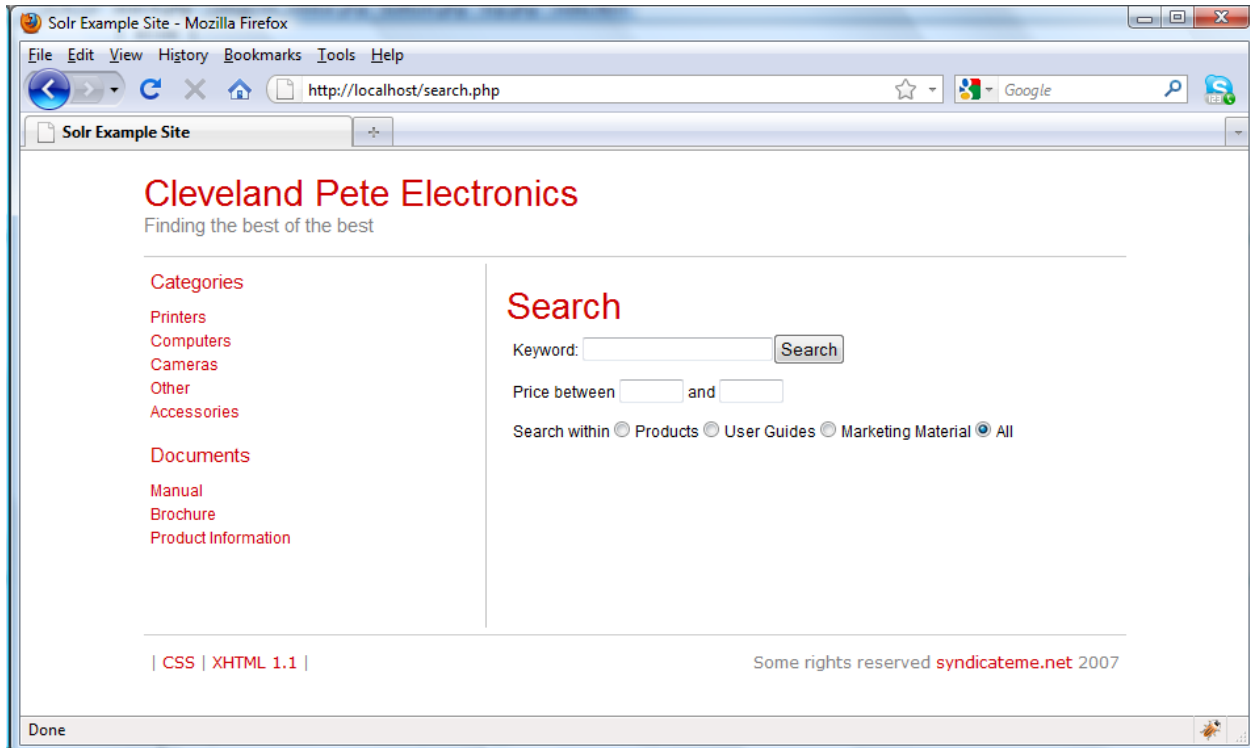
Solr returns the results of a query over HTTP, so as long as you can make an HTTP call, you can get the results from any application. For many people, this means using a utility such as cURL, but in this case it's simpler to just use `file_get_contents()`, which returns a string of JSON data we can then decode into a PHP object. Once we've done that, we can use the object to extract the appropriate information from each returned `doc` object.

Notice that other than encoding the user's input so that it could be included in a URL, we didn't have to do any special machinations on the search term. Because of the way Solr is structured, we didn't have to worry about the same type of damage that can be inflicted by a SQL injection attack; there's no way to structure the contents of a query so that it does anything but search.

Notice that the code here is no longer dependent on the actual query; we can search on multiple fields or perform other searching tricks without affecting the results display. Similarly, all of our data, of no matter how many types, is all displayed in one set. Certainly there may be situations where we don't want that – and Solr does provide a way to do the opposite – but if you want results sorted by relevance (as Solr does by default), this becomes much more important.

### *Searching Against Multiple Fields Using SQL*

So far, we've done a simple search for text within our products and documents. But what happens when we give users the option for “advanced” search, enabling them to search against different fields? For example, consider this page:



Adding just a few more fields gives the user more control.

Adding the form and validation to make sure that prices are submitted as numbers is straightforward:

```
<?php
include("../includes/top.php");
?>
<script type="text/javascript"
  src="http://code.jquery.com/jquery-1.4.2.min.js"></script>
<script type="text/javascript">

$(document).ready(function(){

    $("#searchForm").submit(function(e){
        var lower = $("#lower").val();
        var upper = $("#upper").val();

        if (isNaN(lower) || isNaN(upper)){
```

```
        alert("Upper and lower price values must be numeric.");
        return false;
    } else {
        return true;
    }
});

});

</script>

<h1>Search</h1>
<form action='search.php' method='get' id='searchForm'>
<p>Keyword: <input type='text' name='keyword' id='keyword' /><input
type='submit' value='Search' class='submitButton' /></p>
<p>Price between <input type='text' name='lower' id='lower' size='4' />
and <input type='text' name='upper' id='upper' size='4' /></p>
<p>Search within
    <input type='radio' name='docType' value='3' /> Products
    <input type='radio' name='docType' value='1' /> User Guides
    <input type='radio' name='docType' value='2' /> Marketing Material
    <input type='radio' name='docType' value='*' checked='checked' /> All</p>
</form>
<?php

    if (isset($_GET['keyword'])) {
        ...
```

#### [Adding additional fields to the search page](#)

All we've done here is add three new fields to the form, along with a simple Javascript routine that ensures the form won't be submitted with non-numerical data for the upper and lower price values.

Adding this functionality to the results, however, is a bit more complicated:

```
...
if (isset($_GET['keyword'])) {

    $term = $_GET['keyword'];
    $upper = $_GET['upper'];
    $lower = $_GET['lower'];
    $docType = $_GET['docType'];

    $resultsFound = false;
    $priceUsed = false;
```

```

if ($docType == '3' || $docType == '*'){
    $SQL = " SELECT * FROM products where (product_name like
        '%".mysql_real_escape_string ($term)."% ' or product_adcopy
        like '%".mysql_real_escape_string ($term)."% ' or
        product_description like '%".mysql_real_escape_string
        ($term)."% ')" ;

    if ($lower.$supper != ''){
        $priceWhere = "";
        if ($lower != ''){
            $priceWhere .=
                " and product_price >= ".mysql_real_escape_string($lower);
        }
        if ($supper != ''){
            $priceWhere .=
                " and product_price <= ".mysql_real_escape_string($supper);
        }
        $SQL .= $priceWhere;
        $priceUsed = true;
    }

    $results = mysql_db_query($db, $SQL, $cid);
    if (!$results) { echo( mysql_error()); }

    while ($row = mysql_fetch_array($results)) {
        $prodId = $row["id"];
        $prodName = $row["product_name"];
        $prodDesc = $row["product_description"];

        echo ("<h2><a href='products.php?id=$prodId'>".
            "$prodName</a></h2>");
        echo ("<p>$prodDesc</p>");
        $resultsFound = true;
    }
}

if (($docType == '1' || $docType == '2' || $docType == '*')
    && !$priceUsed){

    $SQL = " SELECT * FROM documents where (document_title like
        '%".mysql_real_escape_string($term)."% ' or document_desc like
        '%".mysql_real_escape_string($term)."% ' or document_text like
        '%".mysql_real_escape_string($term)."% ')" ;

```



```
if ($docType != '*'){
    $SQL .= " and doc_type_id = ".mysql_real_escape_string($docType);
}

$results = mysql_db_query($db, $SQL, $cid);
if (!$results) { echo( mysql_error()); }

while ($row = mysql_fetch_array($results)) {
    $docId = $row["id"];
    $docTitle = $row["document_title"];
    $docDesc = $row["document_desc"];

    echo ("<h2><a href='document.php?id=$docId'>$docTitle</a></h2>");
    echo ("<p>$docDesc</p>\n");
    $resultsFound = true;
}

if (!$resultsFound){
    echo "<h2>No results found.</h2>";
}

include("../includes/bottom.php");
?>
```

#### Adding additional fields to the search criteria

First we have to determine whether to search products and/or documents, and then we have to detect a price search and adjust the queries accordingly. Finally, we need to make sure that regardless of the user's document type choice, documents aren't searched if the user asked for a price, because by definition, documents don't have a price.

Now, that is an awful lot of work for just two additional search fields. Imagine if you needed to do something complicated!

### *Searching Against Multiple Fields Using Solr*

So in order to do this using Solr, what do we have to change in the results?

Nothing, as it happens.

Because we're not handling the actual analysis of the data, all we need to do is adjust the actual query, and we can do that in the same Javascript we use to validate the pricing numbers:

```

...
$(document).ready(function(){

    $("#searchForm").submit(function(e){

        var lower = $("#lower").val();
        var upper = $("#upper").val();
        var keyword = $("#keyword").val();
        var docType = $("input[name='docType']:checked").val();

        if (isNaN(lower) || isNaN(upper)){
            alert("Upper and lower price values must be numeric.");
            return false;
        } else {
            if (keyword == "") keyword = "*:~*";
            var query = "text:"+keyword+
                " AND docTypeRaw:"+docType+priceQuery;

            $("#keyword").val(query);
            return true;
        }
    });

});
...

```

#### Adding additional search fields to the Solr query

All we're doing in this case is building the query right in the validation routine, then inserting that value into the keyword field, and submitting the form as usual. We need make absolutely no changes to the display of the search results.

Now, I should point out here that adding the price and docType to the actual query isn't the ideal way of accomplishing this. In this case, we wind up with a search URL of, say:

[http://localhost:8983/solr/select/?q=printer+AND+price:\[250+TO+500\]+AND+docTypeRaw:3](http://localhost:8983/solr/select/?q=printer+AND+price:[250+TO+500]+AND+docTypeRaw:3)

While there's nothing actually wrong with that, there's a better way to do it, and that's by using filter queries, as in:

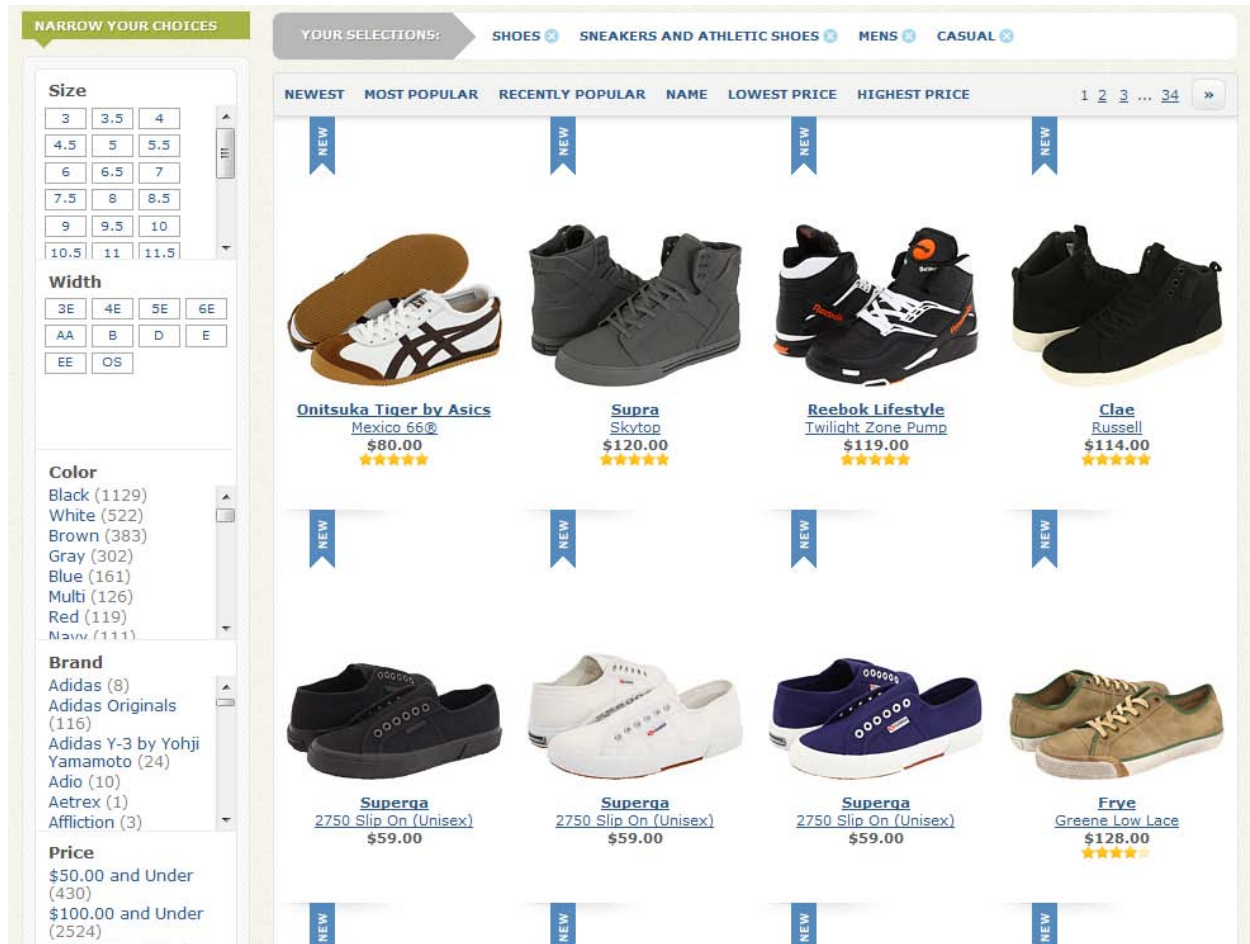
[http://localhost:8983/solr/select/?q=printer&fq=price:\[250+TO+500\]&fq=docTypeRaw:3](http://localhost:8983/solr/select/?q=printer&fq=price:[250+TO+500]&fq=docTypeRaw:3)

The advantage here, besides being conceptually cleaner, is that our filters on price and document type won't affect relevancy rankings. Solr can also cache them for better performance.

## Faceted Searching

And while we're on the subject of using filters, we should take a minute to look at one of the most exciting parts of using Solr rather than building using SQL, and that's faceted search.

Faceted search is way of classifying results by various criteria to help the user narrow down his or her search. Consider, for example, this actual page from the Zappos.com web site, which runs on Solr:



#### Faceted search on zappos.com

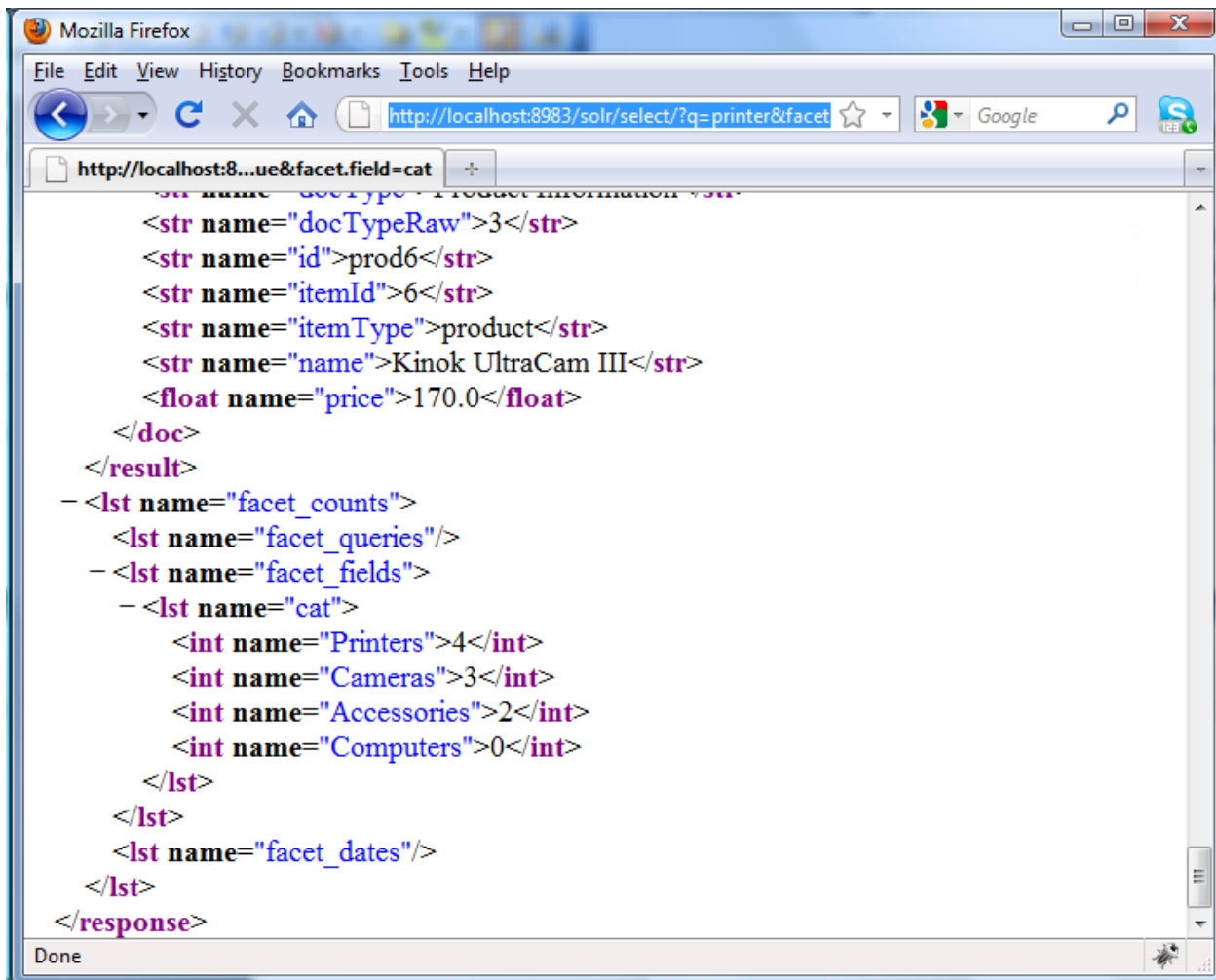
I did a search for men's sneakers, and then filtered by the "casual" theme. On the left, you'll see I can further filter my results by size, width, color, brand, and price range. The advantage here is that I will never wind up with "no results", because only the valid options are showing.

Now, I'm not going to torture all of us by duplicating this setup using SQL, but imagine for a moment that you were going to create this page that way. We're looking at a minimum of six queries, all managed separately and taking into account previous filtering criteria. My head hurts just thinking about it.

So let's look at what it takes to do this using Solr. Query-wise, getting the information necessary for displaying a page like this is pretty simple:

<http://localhost:8983/solr/select/?q=printer&facet=true&facet.field=cat>

Adding these two parameters to the search adds the faceting information to our results, as in:



Faceting information added to the search

Here you can see the various categories, with their values and how many results there are for this search. So if we were displaying categories as an option, we wouldn't display the Computers category:

```
if (isset($_GET['keyword'])) {
    $term = $_GET['keyword'];
    $filter = $_GET['filter'];
    if ($cat != "") {
        $filter = '&fq=cat:'. $_GET['cat'];
    }

    $jsonurl =
        "http://localhost:8983/solr/select/?q=".urlencode($term).$filter.
        "&facet=true&facet.field=cat&wt=json";
    $json = file_get_contents($jsonurl, 0, null, null);
    $json_output = json_decode($json);

    $response = $json_output->response;
    $resultsFound = $response->numFound;

    if ($resultsFound > 0) {

        echo("<h2>$resultsFound results found</h2><br />");

        $catFacets = $json_output->facet_counts->facet_fields->cat;
        $cats = array_chunk($catFacets, 2);

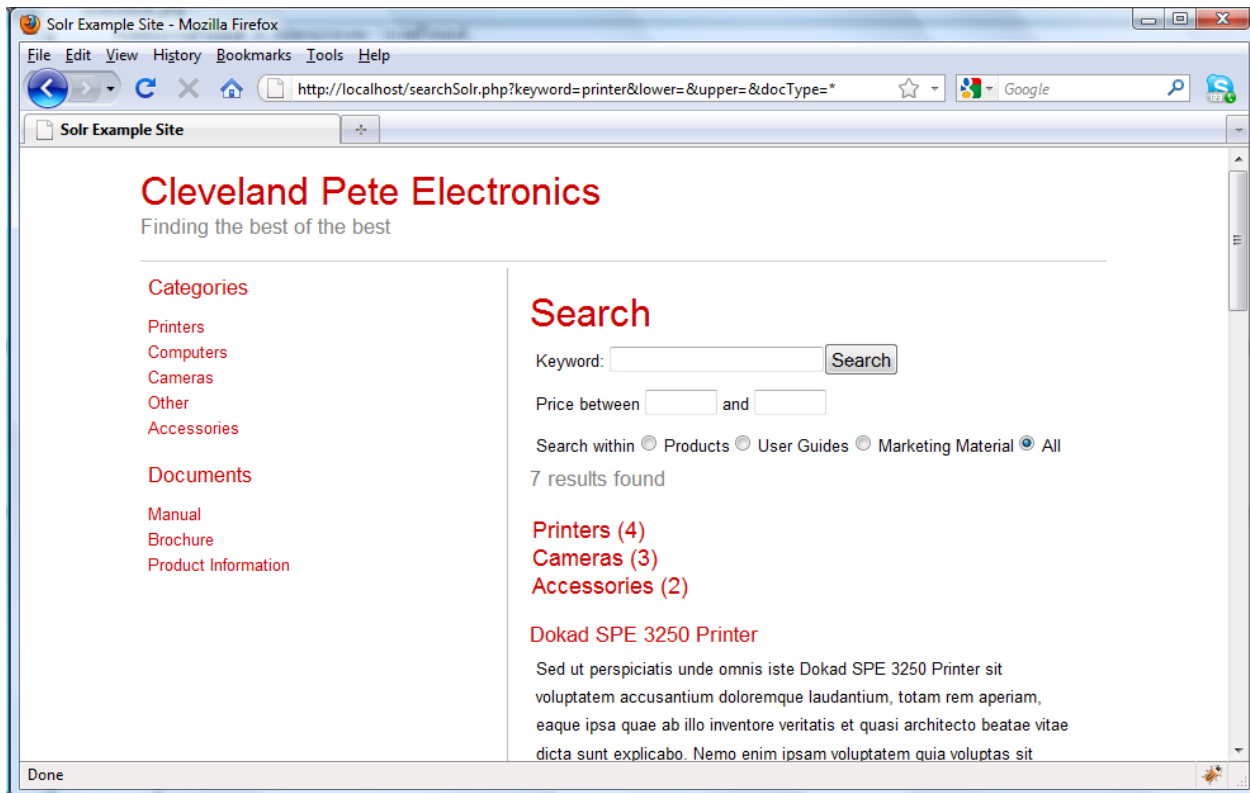
        foreach ($cats as $thisCat) {
            $catName = $thisCat[0];
            $catNumber = $thisCat[1];
            if ($catNumber > 0) {
                echo "<a href='searchSolr.php?keyword=".urlencode($term).
                    "&cat=".$catName."'>".
                    $catName." (".$catNumber.)</a><br />";
            }
        }
        echo "<br />";

        foreach($response->docs as $thisDoc) {
            $pageName = $thisDoc->itemType;
        }
    }
}
```

...

[Adding faceted searching](#)

So we've added the cat facet to the search, which means we can check for categories, and once we've checked for results, we can display a link that provides the appropriate category to filter on. When the user clicks on of those links, we add a filter for that category. The result looks something like this:



#### Adding faceting to our search

We can also add filtering by price range with a query such as:

[http://localhost:8983/solr/select/?q=printer&facet=true&facet.field=cat  
&facet.query=price:\[\\*+T0+250\]&facet.query=price:\[251+T0+500\]  
&facet.query=price:\[501+T0+\\*\]](http://localhost:8983/solr/select/?q=printer&facet=true&facet.field=cat&facet.query=price:[*+T0+250]&facet.query=price:[251+T0+500]&facet.query=price:[501+T0+*])

Here we're using a series of facet queries on price to create ranges, and Solr does the work of figuring out which items belong in which range. So we wind up with a page such as:

```
if ($cat != ""){  
    $filter = '&fq=cat:'.$_GET['cat'];
```

```

}
$priceRange = $_GET['priceRange'];
if ($priceRange != ""){
    $filter .= '&fq='.urlencode('price:['.$priceRange.']');
}

$jsonurl =
    "http://localhost:8983/solr/select/?q=".urlencode($term).$filter.
    "&facet=true&facet.field=cat&facet.query=" .
    urlencode("price:[* TO 250]")."&facet.query=" .
    urlencode("price:[251 TO 500]")."&facet.query=" .
    urlencode("price:[501 TO *]")."&wt=json";
...
if ($resultsFound > 0){

    echo("<h2>$resultsFound results found</h2><br />");

    echo "<table width='100%'>";
    echo "<tr><td>Category</td><td>Price</td></tr><tr><td>";

    $catFacets = $json_output->facet_counts->facet_fields->cat;
    $cats = array_chunk($catFacets, 2);
    ...

    echo "</td><td>";

    $priceFacets = $json_output->facet_counts->facet_queries;
    foreach ($priceFacets as $rangeName => $rangeNumber){
        if ($rangeNumber > 0){
            $rangeString = substr($rangeName, 7, -1);
            echo "<a href='searchSolr.php?keyword=".urlencode($term).
                "&priceRange=".$rangeString."'>".$rangeString.
                " (".$rangeNumber.)</a><br />";
        }
    }

    echo "</td></tr></table>";

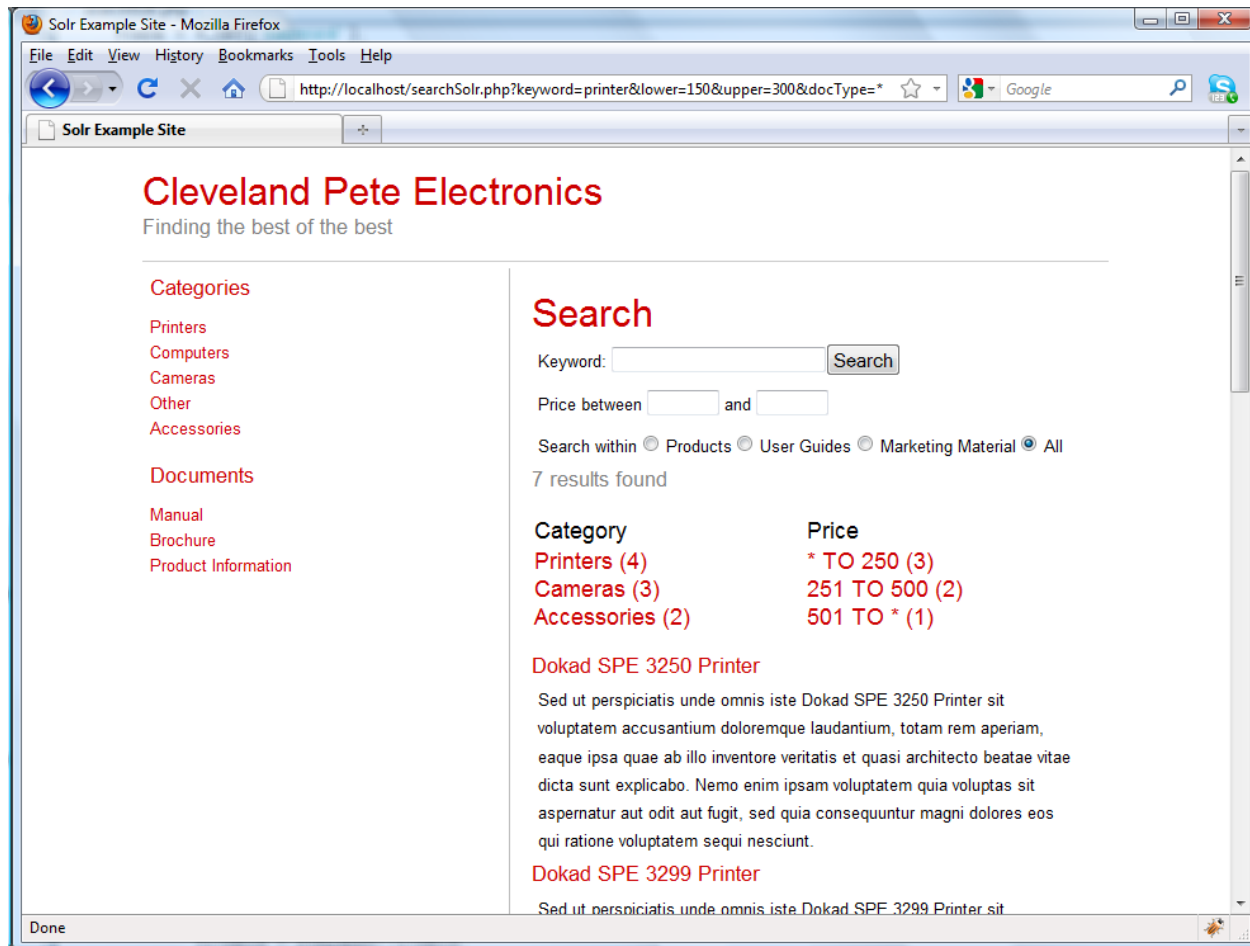
    ...

```

#### Adding facet queries

This is the same basic principle we used to add category facets, so we wind up with a page something like this:





### Adding facet queries

There's obviously more you can do with this, such as improving the presentation of price ranges, carrying through filters more dynamically, and so on, and we'll leave this as an exercise for the reader.

But you get the general idea; faceted searching, a nightmare to pull off in SQL, basically comes free out of the box with Solr.

## Summary

The world wide web has been running on databases for a very long time, and it's probably unreasonable to expect that to change now, when so much information is already housed in an RDBMS. But just because the rest of your application runs on SQL doesn't mean your search has to.

As we've seen, a dedicated search solution such as Solr can make your life a whole lot easier by streamlining the search process itself. Because the actual search is offloaded to Solr, all you need to do is provide the appropriate query and display the results. And it doesn't matter if your data and documents are stored in an RDBMS; the `DataImportHandler` makes indexing for Solr a fairly straightforward task.

But really, we've just skimmed over the tip of the iceberg in this paper. Just as Solr makes the process of searching almost trivial compared to doing it "by hand" with SQL, it also simplifies the process of providing much more advanced search capabilities.

Because at the end of the day, it's about whether your user can find the information he or she is looking for, and find it fast. That means that more relevant results need to appear first, and that's how Solr works. It means that you may need sophisticated stemming algorithms so different forms of the user's terms are searched. It may mean customizing the list of "stop words", or common words such as "the" that are ignored in a search; Solr lets you customize that behavior right down to the field level.

All those things are behind the scenes, and your user might not even realize they're happening. But they will realize it if you provide them with faceted searching, enabling them to narrow their searches by categories or other criteria. They'll notice if you automatically provide results that may not contain their search terms, but are still related. They'll make fewer mistakes and have fewer unsuccessful searches if you provide them with "autocomplete" capabilities that see what they've already typed and provide suggestions from terms that exist in your index. Solr makes providing these capabilities straightforward, and definitely puts them within your reach.

Today's users expect a high level of functionality from their searches. Even if your application is RDBMS-based, with a dedicated search solution such as Solr, you can give it to them.

## Next Steps

For more information on how Lucid Imagination can help search application developers, employees, customers, and partners find the information they need, please visit [www.lucidimagination.com](http://www.lucidimagination.com) to access blog posts, articles, and reviews of dozens of successful implementations.

Certified Distributions from Lucid Imagination are complete, supported bundles of software that include additional bug fixes, performance enhancements, along with our free 30-day Get Started program. Coupled with one of our support subscriptions, a Certified Distribution can provide a complete environment to develop, deploy, and maintain commercial-grade search applications. Certified Distributions are available at [www.lucidimagination.com/Downloads](http://www.lucidimagination.com/Downloads).

Please e-mail specific questions to:

Support and Service: [support@lucidimagination.com](mailto:support@lucidimagination.com)

Sales and Commercial: [sales@lucidimagination.com](mailto:sales@lucidimagination.com)

Consulting: [consulting@lucidimagination.com](mailto:consulting@lucidimagination.com)

Or call: 1.650.353.4057

## Appendix: Lucene/Solr Features and Benefits

Solr and Lucene are complementary technologies that offer very similar underlying capabilities. In choosing a search solution that is best suited for your requirements, key factors to consider are application scope, development environment, and software development preferences.

Solr is the Lucene Search Server. It presents a web service layer built atop Lucene using the Lucene search library and extending it to provide application users with a ready-to-use search platform. Solr brings with it operational and administrative capabilities like web services, faceting, configurable schema, caching, replication, and administrative tools for configuration, data loading, statistics, logging, cache management, and more. It fully encapsulates Lucene's speed, relevancy ranking, complete query capabilities, portability, scalability, and low overhead indexes and rapid incremental indexing.

In the overwhelming majority of cases, Solr provides the starting point for most developers who are building a Lucene-based search application. It comes ready to run in a servlet container such as Tomcat or Jetty, making it ready to scale in a production Java environment.

With convenient ReST-like/web-service interfaces callable over HTTP, and transparent XML-based configuration files, Solr can greatly accelerate application development and maintenance. In fact, Lucene programmers have often reported that they find Solr contains "the same features I was going to build myself as a framework for Lucene, but already very well implemented." Using Solr, enterprises can customize the search application according to their requirements, without involving the cost and risk of writing the code from the scratch.

As functional siblings, Solr and Lucene have become popular alternatives for search applications; the two differ mainly in the style of application development used. Key benefits of search with Lucene/Solr include:

- **Search Quality: Speed, Relevance, and Precision** Lucene/Solr provides near-real-time search and strong relevance ranking to deliver contextually relevant and accurate results very quickly. Tailor-made coding for relevancy ranking and sophisticated search capabilities like faceted search help users in sorting, organizing, classifying, and structuring retrieved information to ensure that search delivers desired results. Search with Lucene/Solr also provides proximity

operators, wildcards, fielded searching, term/field/document weights, find-similar functions, spell checking, multilingual search, and much more.

- **Lower Cost and Greater Flexibility, Plug and Play Architecture** Lucene/Solr reduces recurring and nonrecurring costs, lowering your TCO. As open source software, it does not require purchase of a license and is freely available for use. The open source code can be used as is, modified, customized, and updated as appropriate to your needs. Solr is easily embedded in your enterprise's existing infrastructure, reducing costs of installation, configuration, and management.
- **Open Source Platform for Portability and Easy Deployment** Because Lucene/Solr is an open-source software solution, it is based on open standards and community-driven development processes. It is highly portable and can run on any platform that supports Java. For instance, you can build an index on Linux and copy it to a Microsoft Windows machine and search there. This unsurpassed portability enables you to keep your search application and your company's evolving infrastructure in tandem. Lucene, in turn, has been implemented in other environments, including C#, C, Python, and PHP. At deployment time, Solr offers very flexible options; it can be easily deployed on a single server as well as on distributed, multiserver systems.
- **Largest Installed Base of Applications, Increasing Customer Base** Lucene/Solr is the most widely used open source search system and is installed in around 4,000 organizations worldwide. Publicly visible search sites that use Lucene/Solr include CNET, LinkedIn, Monster, Digg, Zappos, MySpace, Netflix, and Wikipedia. Lucene/Solr is also in use at Apple, HP, IBM, Iron Mountain, and Los Alamos National Laboratories.
- **Large Developer Base and Adaptability** As community developed software, Lucene/Solr provides transparent development and easy access to updates and releases. Developers can work with open source code and customize the software according to business-specific needs and objectives. Its open source paradigm lets Lucene/Solr provide developers with the freedom and flexibility to evolve the software with changing requirements, liberating them from the constraints of commercial vendors.
- **Commercial-Grade Support for Mission Critical Search Applications from Lucid Imagination** Lucid Imagination provides the expertise, resources, and services that are needed to help enterprises deploy and develop Lucene-based search solutions efficiently and cost-effectively. Lucid helps enterprises achieve optimal search performance and accuracy with its broad range of expertise, which includes indexing and metadata management, content analysis, business rule application, and natural language processing. Lucid Imagination also offers certified distributions of Lucene and Solr, commercial-grade SLA-based support, training, high-level consulting and value-added software extensions to enable customers to create powerful and successful search applications.