JNBridge CELEBRATING 10 YEARS OF JAVA & .NET

Building a LINQ Provider for HBase MapReduce

HBase/ Hadoop — JNBridgepro — LINQ Query

Building a LINQ Provider for HBase MapReduce

Summary

HBase is a distributed, scalable, big data storage and retrieval system developed as part of the Apache Hadoop project. As a Java framework, HBase applications must use Java APIs, resulting in single-platform solutions. Cross-platform solutions, particularly those that provide front-end data query, analysis and presentation, like Microsoft Excel, or query languages like LINQ, the .NET Language Integrated Query framework, are currently not supported.

This latest project from JNBridge Labs explores building a simple .NET LINQ provider for HBase using the Java API for creating, managing and scanning HBase tables and the .NET LINQ provider interfaces. In addition, the project investigates LINQ support for HBase MapReduce jobs written in Java by adding an extension to the LINQ query syntax.

By continuing the research into Hadoop and .NET interoperability introduced in the previous project, Creating .NET-based Mappers and Reducers for Hadoop, JNBridge Labs champions interoperability between Windows and any Java solution running on any system. The potential of products like DryadLINQ—canceled when Microsoft shelved the Dryad HPC project in favor of Hadoop—is still relevant, but only if that potential isn't constrained to Hadoop running on particular systems. Microsoft's port of Hadoop from Linux to Azure and Windows Server is single-system. Interoperability between Hadoop and LINQ or Excel means supporting Hadoop on any system.

Introduction

Apache HBase is a *big table* implementation using the Apache Hadoop platform. Hadoop, in addition to providing distributed, parallel processing of petabyte-size data-sets, also provides the distributed file system, HDFS, where HBase tables are stored. An HBase table is a distributed, multi-dimensional sorted map containing structured data. A web access record is the obvious canonical example:

```
125.125.125.125 – [10/Oct/2011:21:15:05 +0500] "GET /index.html HTTP/1.0" 200 1043
"http://www.ibm.com/" "Mozilla/4.05 [en] (WinNT; I)"
```

While the above record can be thought of as a row with several columns, the point is that it's in a single table; there's no relationship to other tables because there are no other tables. Nor is there a need to worry about indexes. Designing efficient schema that enable efficient queries isn't the point. That's because a row isn't an object like an employee or a bank account, it's a single datum and only has value as part of a data set, which can have several million rows. The queries one makes to these tables tend to be reductions to simple statistics, i.e. sums. In other words, we're talking brute-force, something that Apache Hadoop, MapReduce algorithms and parallel processing make straight-forward and fast.

LINQ, or Language INtegrated Query, is a Microsoft .NET feature that provides query capabilities in the .NET languages. Actually, it's syntactic sugar around the set of static query extensions defined in the class System.Linq.Enumerable that can be used to query any .NET collection of objects that implement *IEnumerable<T>*. LINQ is also extendable; it's possible to implement a LINQ provider for just about anything as long as there's a convenient abstraction that allows queries. In the case of HBase, a LINQ provider makes perfect sense.

This current offering from JNBridge Labs will explore building a simple LINQ provider for HBase using the HBase client Java API for creating, managing and scanning tables. In addition, the LINQ provider will also support HBase MapReduce jobs written in Java using the HBase table mapper/reducer templates and Hadoop. This differs from the previous Hadoop lab that demonstrated using mappers and reducers written in .NET, but called by the MapReduce Job on the Java-side.

Getting Started

Here are the components required for the lab. Also, some important links, resources and configuration tips.

Apache Hadoop Stack

The first requirement is access to a Hadoop/HBase implementation. Installing and configuring Hadoop and HBase can be an exercise, particularly if the platform is Windows rather than Linux. The best option is to download the free Linux virtual images from Cloudera. The VM runs CentOS 5.8 and contains Cloudera's distribution of the entire Apache Hadoop stack already installed and configured. While it's not truly distributed among many machines, each component of the Hadoop stack runs as a separate process, so it's possible to add more worker nodes. The VM is available for VMWare, KVM and VirtualBox. Make sure that the VM's access to the internet is bridged, *do not use NAT*. Also, because Java server applications running on Linux are notoriously finicky when it comes to DNS and the loop-back interface, make sure to modify the */etc/hosts* file on the Linux and Windows machines. This will ensure that the HBase client API can actually connect to the Hadoop/HBase implementation through ZooKeeper.

On the Linux VM, remove the loopback IP address, 127.0.0.1, replacing it with the IP address of the VM, then recycle the VM.

```
# 127.0.0.1       localhost.localdomain localhost
192.168.1.3       localhost.localdomain localhost
```

On the Windows development machine, add this to the hosts file (\Windows\System32\drivers\etc\hosts):

```
192.168.1.3       localhost.localdomain
```

Development Machine

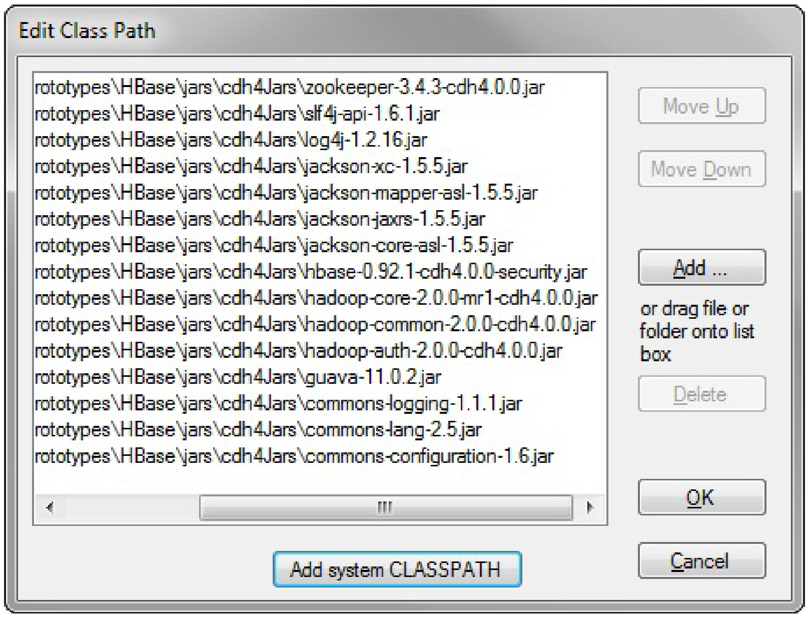
Visual Studio 2010, Eclipse Juno and, of course, JNBridgePro 6.1. JNBridgePro will be used to build .NET proxies of the HBase client API as well as a Java class that implements MapReduce. This will enable calling Java from the .NET code that implements the LINQ provider.

Calling Java from .NET: Creating proxies using JNBridgePro

Since the LINQ provider is written in C#/.NET and needs to call a Java class API, the first step is to use the JNBridgePro plug-in for Visual Studio to create an assembly of proxies that represent the Java API. When a proxy of a Java class is instantiated in .NET, the real Java object is instantiated in the Java Virtual Machine. The JNBridgePro run-time manages communications, i.e. invoking methods, and syncing garbage collection between the .NET CLR and the JVM.

For this development step, as well as during run-time, a bunch of Hadoop, HBase and ZooKeeper JAR files must be available on the Windows machine. While these can be obtained by downloading the zip archives from the Apache project, it's best to use the JAR files from the Cloudera distribution, thus avoiding version-itis. These can be scraped from the VM (look in */usr/lib/hadoop/lib*, */usr/lib/hbase/lib*, etc.)

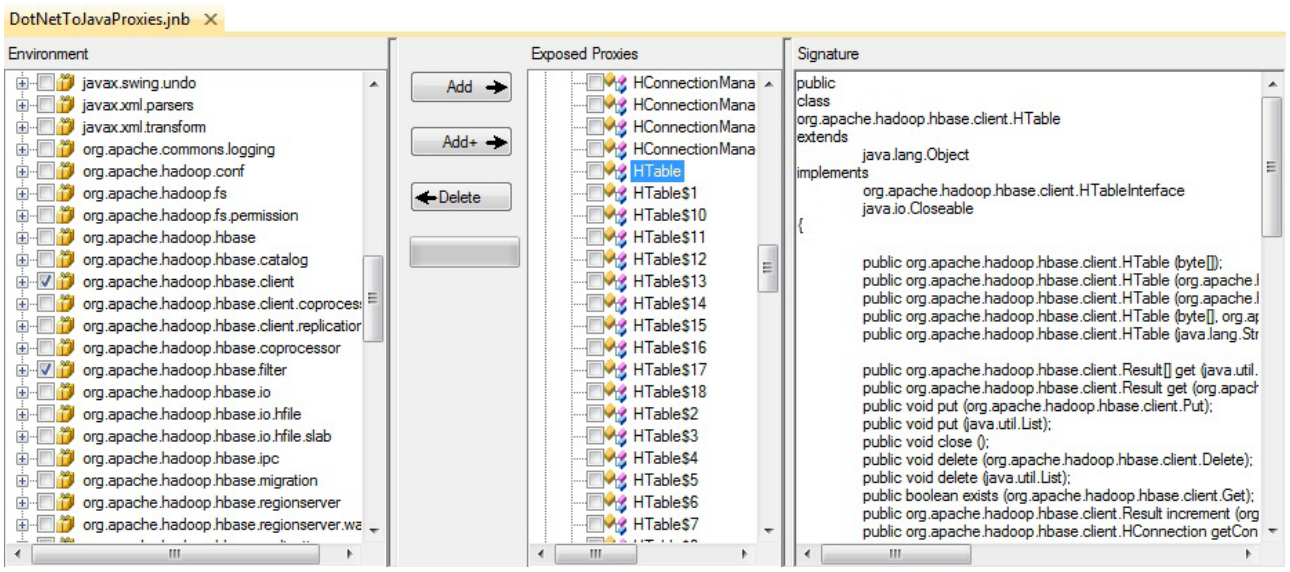
This is a screen shot of the *Edit Class Path* dialog for the JNBridgePro Visual Studio plug-in.



```
rototypes\HBase\jars\cdh4Jars\zookeeper-3.4.3-cdh4.0.0.jar
rototypes\HBase\jars\cdh4Jars\slf4j-api-1.6.1.jar
rototypes\HBase\jars\cdh4Jars\log4j-1.2.16.jar
rototypes\HBase\jars\cdh4Jars\jackson-xc-1.5.5.jar
rototypes\HBase\jars\cdh4Jars\jackson-mapper-asl-1.5.5.jar
rototypes\HBase\jars\cdh4Jars\jackson-jaxrs-1.5.5.jar
rototypes\HBase\jars\cdh4Jars\jackson-core-asl-1.5.5.jar
rototypes\HBase\jars\cdh4Jars\hbase-0.92.1-cdh4.0.0-security.jar
rototypes\HBase\jars\cdh4Jars\hadoop-core-2.0.0-mr1-cdh4.0.0.jar
rototypes\HBase\jars\cdh4Jars\hadoop-common-2.0.0-cdh4.0.0.jar
rototypes\HBase\jars\cdh4Jars\hadoop-auth-2.0.0-cdh4.0.0.jar
rototypes\HBase\jars\cdh4Jars\guava-11.0.2.jar
rototypes\HBase\jars\cdh4Jars\commons-logging-1.1.1.jar
rototypes\HBase\jars\cdh4Jars\commons-lang-2.5.jar
rototypes\HBase\jars\cdh4Jars\commons-configuration-1.6.jar
```

These are the JAR files required to create the .NET proxies. During run-time, three additional JAR files must be included in the JVM's class path when initiating the bridge between the JVM and the CLR: *avro-1.5.4.jar, commons-httpclient-3.1.jar* and *slf4j-nop-1.6.1.jar* (the last JAR file inhibits logging by Hadoop and HBase. If you're interested in the volume of information logged during a MapReduce computation, just remove this JAR file from the class path).

This is a screen shot of the JNBridgePro proxy tool in Visual Studio. The left hand pane shows all the namespaces found in the JAR files shown in the above dialog. The required namespaces are *org.apache.hadoop.hbase.client* and *org.apache.hadoop.hbase.filter*. In addition, individual classes like *org.apache.hadoop.hbase.HBaseConfiguration* are required (see the link at the end of this document to download the source).



javax.swing.undo
javax.xml.parsers
javax.xml.transform
org.apache.commons.logging
org.apache.hadoop.conf
org.apache.hadoop.fs
org.apache.hadoop.fs.permission
org.apache.hadoop.hbase
org.apache.hadoop.hbase.catalog
org.apache.hadoop.hbase.client
org.apache.hadoop.hbase.client.coproces
org.apache.hadoop.hbase.client.replicatior
org.apache.hadoop.hbase.coprocessor
org.apache.hadoop.hbase.filter
org.apache.hadoop.hbase.io
org.apache.hadoop.hbase.io.hfile
org.apache.hadoop.hbase.io.hfile.slab
org.apache.hadoop.hbase.ipc
org.apache.hadoop.hbase.migration
org.apache.hadoop.hbase.regionserver
org.apache.hadoop.hbase.regionserver.wa

HConnectionMana
HConnectionMana
HConnectionMana
HConnectionMana
HTable
HTable$1
HTable$10
HTable$11
HTable$12
HTable$13
HTable$14
HTable$15
HTable$16
HTable$17
HTable$18
HTable$2
HTable$3
HTable$4
HTable$5
HTable$6
HTable$7

```
public
class
org.apache.hadoop.hbase.client.HTable
extends
            java.lang.Object
implements
            org.apache.hadoop.hbase.client.HTableInterface
            java.io.Closeable
{

            public org.apache.hadoop.hbase.client.HTable (byte[]);
            public org.apache.hadoop.hbase.client.HTable (org.apache.l
            public org.apache.hadoop.hbase.client.HTable (org.apache.l
            public org.apache.hadoop.hbase.client.HTable (byte[], org.ap
            public org.apache.hadoop.hbase.client.HTable (java.lang.Str

            public org.apache.hadoop.hbase.client.Result[] get (java.util.
            public org.apache.hadoop.hbase.client.Result get (org.apach
            public void put (org.apache.hadoop.hbase.client.Put);
            public void put (java.util.List);
            public void close ();
            public void delete (org.apache.hadoop.hbase.client.Delete);
            public void delete (java.util.List);
            public boolean exists (org.apache.hadoop.hbase.client.Get);
            public org.apache.hadoop.hbase.client.Result increment (org
            public org.apache.hadoop.hbase.client.HConnection getCon
```

By clicking on the **Add+** button, the chosen classes, as well as every dependent class, will be found and displayed in the center pane. The right-hand pane displays the public members and methods of the Java HTable class. The last step is to build the proxy assembly, HBaseProxies.dll.

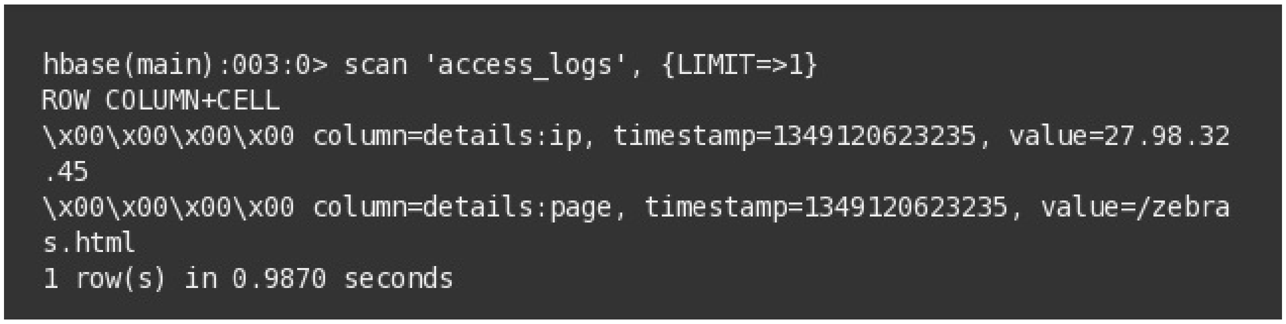
Creating and populating an HBase Table

It would be nice to have an HBase table loaded with data and provide an opportunity to test calling various HBase Java APIs from .NET. To keep things simple—after all, this is an example—the data will consist of an IP address, like "88.240.129.183" and the requested web page, for example "/zebra.html". The code for creating and populating the table, shown below, is in the project, *LoadData*.

```
using org.apache.hadoop.hbase;
using org.apache.hadoop.hbase.client;
using org.apache.hadoop.hbase.util;
using org.apache.hadoop.conf;

com.jnbridge.jnbproxy.JNBRemotingConfiguration.specifyRemotingConfiguration(
    com.jnbridge.jnbproxy.JavaScheme.sharedmem
    , @"C:\Program Files\Java\jre6\bin\client\jvm.dll"
```

```
        , @"C:\Program Files\JNBridge\JNBridgePro v6.1jnbcore\jnbcore.jar"
        , @"C:\Program Files\JNBridge\JNBridgePro v6.1\jnbcore\bcel-5.1-jnbridge.jar"
        , @"C:\HBase\jars\cdh4Jars\zookeeper-3.4.3-cdh4.0.0.jar;...");

Configuration hbaseConfig = HBaseConfiguration.create();
hbaseConfig.set("hbase.zookeeper.quorum", "192.168.1.3");
HBaseAdmin admin = new HBaseAdmin(hbaseConfig);
HTableDescriptor desc = new HTableDescriptor("access_logs");
desc.addFamily(new HColumnDescriptor("details"));
admin.createTable(desc);
HTable htable = new HTable(hbaseConfig, "access_logs");
htable.setAutoFlush(false);
htable.setWriteBufferSize(1024 * 1024 * 12);
int totalRecords = 100000;
Random rand = new Random();
string[] pages = { "/dogs.html", "/cats.html", "/rats.html", "/bats.html", "/zebras.html"
        , "/t-rex.html", "/lizard.html", "/birds.html", "/beetles.html", "/elephants.html"};
string[] ipAdrs = { "88.240.129.183", "144.122.180.55", "85.100.75.104", "78.34.27.101"
        , "23.121.45.220", "206.27.34.178", "174.82.96.35", "13.234.26.45"
        , "245.213.200.10", "167.38.92.14", "27.98.32.45", "22.48.92.13" };
for (int i = 0; i < totalRecords; i++)
{
        Put put = new Put(Bytes.toBytes(i));
        put.add(Bytes.toBytes("details"), Bytes.toBytes("ip")
        , Bytes.toBytes(ipAdrs[rand.Next(ipAdrs.Length)]));
        put.add(Bytes.toBytes("details"), Bytes.toBytes("page")
        , Bytes.toBytes(pages[rand.Next(pages.Length)]));
        htable.put(put);
}
htable.flushCommits();
htable.close();
```

The above C# code uses the type 'sbyte', not 'byte'. That's because the Java type 'byte' is signed, in .NET it's unsigned. Since this is .NET calling a Java method, the proxies return type 'sbyte'. Bringing up the HBase shell on the Linux VM, we can dump the first row of the HBase table, *access_logs*.



```
hbase(main):003:0> scan 'access_logs', {LIMIT=>1}
ROW COLUMN+CELL
\x00\x00\x00\x00 column=details:ip, timestamp=1349120623235, value=27.98.32
.45
\x00\x00\x00\x00 column=details:page, timestamp=1349120623235, value=/zebra
s.html
1 row(s) in 0.9870 seconds
```

The row is comprised of two cells, *ip* and *page*, both grouped under a column, *details*. A column is a semantic grouping of cells, hence the use of the term ‘family’ that prevails. The hex numbers on the left, both zero, are the index to the first row. The timestamp allows a third dimension, providing queries based on a time span.

Building a custom LINQ Provider

LINQ, in its simplest form, is a set of extension methods to the generic interface IEnumerable<T>. Consider the following class, really no more than a structure:

```
public class AccessRecord {
    // Properties.
    public string ipAddress { get; set; }
    public string page { get; set; }
    // Constructor.
    internal AccessRecord(string ip, string page) {
        this.ipAddress = ip;
        this.page = page;
    }
}
```

A collection of AccessRecord instances using the generic *List<T>* template, which inherits from IEnumerable<T>, can be queried using the static extension methods defined in the class *System.Linq. Enumerable*. For example, consider determining the frequency—the count—of the pages requested by a given IP address.

```
List<AccessRecord> lst = new List<AccessRecord>();
var query1 = lst.Where(rec => rec.ipAddress == “88.240.129.183”).OrderBy(rec => rec.page)
    .GroupBy(rec => rec.page)
    .Select(grps => new {ip = grps.Key, count = grps.Count()});
```

Alternatively, using the syntactic sugar in C#, the query can be stated in this form.

```
var query1 = from rec in records
    where rec.ipAddress == “88.240.129.183”
    orderby rec.page
    group rec by rec.page into grps
    select new { ip = grps.Key, count = grps.Count() };
```

While this is pretty cool, it will not scale well. The list of access records could number 10,000,000. Moreover, there’s the overhead of building the initial data structure in the first place. That’s why LINQ supplies a framework for building custom query providers. The entire point is to have both the data and the query computations happen someplace else, like HBase.

Implementing *IOrderedQueryable<T>* and *IQueryProvider*

A LINQ provider requires implementing two interfaces: *IQueryProvider* and either *IQueryable<T>* or

IOrderedQueryable<T>. If the LINQ provider needs to support the *OrderBy* and *ThenBy* query operators, then the IOrderedQueryable interface must be implemented (actually, because IOrderedQueryable inherits from IQueryable, all three interfaces must be implemented).

The implementation of IOrderedQueryable<T> can be found in the class, *QueryableHBaseData*. The important method in this class is the implementation of *IEnumerable<T>.GetEnumerator()*. Consider the code that displays the results of the above queries by *enumerating* over the returned collection.

```
foreach (var pg in query1)
{
    Console.WriteLine(" page: " + pg.ip + " " + pg.count);
}
```

The actual query is executed when the GetEnumerator() method is called on the instance of QueryableHBaseData.

```
public IQueryProvider Provider { get; private set; }
public Expression Expression { get; private set; }
public IEnumerator<TData> GetEnumerator()
{
    return (Provider.Execute<IEnumerable<TData>>(Expression)).GetEnumerator();
}
```

The *Provider* property is the implementation of IQueryProvider, which can be found in the class, *HBaseQueryProvider*. The *Expression* property is the query to be executed. The *HBaseQueryProvider*. Execute() method is responsible for executing the query. So far, all of this is straight forward. The hard part in implementing a LINQ provider is parsing, walking and decorating an expression tree represented by an instance of *System.Linq.Expressions.Expression*.

The innermost-where expression

The reason for walking the expression tree is to find the innermost where expression. A *where* operator is a filter—it's what makes a query, well, a query. Most custom LINQ providers find the *innermost-where* expression and partially evaluate the lambda expression argument to retrieve the filter predicate values. The predicate values, as well as some other information, is used to query the real data source and return the results as a collection of *IQueryable<T>*. The expression tree is then rewritten, essentially replacing the original data source with the collection of IQueryable objects that now resides in memory. The new expression tree is then re-executed using the HBaseQueryProvider class. All of this is implemented in the method *HBaseQueryContext.Execute()*.

```
internal static object Execute(Expression expression, bool IsEnumerable)
{
    // Find the call to Where() and get the lambda expression predicate.
    InnermostWhereFinder whereFinder = new InnermostWhereFinder();
    MethodCallExpression whereExpression = whereFinder.GetInnermostWhere(expression);
    LambdaExpression lambdaExpression =
        (LambdaExpression)((UnaryExpression)(whereExpression.Arguments[1])).Operand;
    // Send the lambda expression through the partial evaluator.
```

```
        lambdaExpression = (LambdaExpression)Evaluator.PartialEval(lambdaExpression);
        // Get the column name(s)
        ColumnFinder cf = new ColumnFinder(lambdaExpression.Body);
        List<string> columns = cf.Columns;
        // Call HBase and get the results.
        AccessRecord[] records = HBaseHelper.GetColumnsFromHBase(
            columns.First<string>()
            , cf.IsIP, cf.IsPage);
        IQueryable<AccessRecord> queryableRecords = records.AsQueryable<AccessRecord>();
        // Copy the expression tree that was passed in, changing only the first
        // argument of the innermost MethodCallExpression.
        ExpressionTreeModifier treeCopier = new ExpressionTreeModifier(queryableRecords);
        Expression newExpressionTree = treeCopier.CopyAndModify(expression);
        if (IsEnumerable)
            return queryableRecords.Provider.CreateQuery(newExpressionTree);
        else
            return queryableRecords.Provider.Execute(newExpressionTree);
    }
```

The class *ColumnFinder* takes the lambda expression that represents the predicate to the where operator and returns the right hand side of the predicate expression, in our example *record.ip == "88.240.129.183"*. The left hand side of the predicate expression, the field to test against, is obtained through the properties *ColumnFinder.IsIP* and *ColumnFinder.IsPage*. In keeping with the above example queries, this data is used to call *HBaseHelper.GetColumnsFromHBase()* with arguments 88.240.129.183, true and false.

Scanning and filtering an HBase table

Now that all the boring stuff is done, we can make some calls to the HBase client Java API from .NET using the proxy assembly. The call to *HBaseHelper.GetColumnsFromHBase()* results in calling this code.

```
    using org.apache.hadoop.hbase;
    using org.apache.hadoop.hbase.client;
    using org.apache.hadoop.hbase.util;
    using org.apache.hadoop.io;
    using org.apache.hadoop.conf;
    using org.apache.hadoop.hbase.filter;
    public static AccessRecord[] hbaseColumnScan(string column, bool isIP, bool isPage)
    {
        HTable tbl = null;
        Scan scn = null;
        SingleColumnValueFilter svcFilter = null;
        ResultScanner rs = null;
        string pg = null;
        Configuration hbaseConfig = HBaseConfiguration.create();
        hbaseConfig.set("hbase.zookeeper.quorum", "192.168.1.3");
        try
        {
            tbl = new HTable(hbaseConfig, “access_logs”);
            scn = new Scan();
            scn.setCaching(500);
            sbyte[] scanColumn = null;
            if ( isIP )
            scanColumn = ipCol;
            else if ( isPage )
            scanColumn = pageCol;
            svcFilter = new SingleColumnValueFilter(family
            , scanColumn
            , CompareFilter.CompareOp.EQUAL
            , Bytes.toBytes(column));
            scn.setFilter(svcFilter);
            rs = tbl.getScanner(scn);
        }
        catch (Exception ex)
        {
            Console.WriteLine(“Unable to perform column scan: “ + ex.Message);
        }
        List<AccessRecord> records = new List<AccessRecord>();
        for (Result r = rs.next(); r != null; r = rs.next())
        {
            pg = Bytes.toString(r.getValue(family, pageCol));
            records.Add(new AccessRecord(column, pg));
        }
        rs.close();
        tbl.close();
        return records.ToArray<AccessRecord>();
    }
```

Testing the LINQ provider

Here’s the example query using the HBase LINQ provider.

```
    QueryableHBaseData<AccessRecord> records =
        new QueryableHBaseData<AccessRecord>();

    var query1 = from record in records
        where record.ipAddress == “88.240.129.183”
        orderby record.page
        group record by record.page into grps
        select new { ip = grps.Key, count = grps.Count() };
    Console.WriteLine(“ IP address 88.240.129.183 visted these pages”);
```

```
foreach (var pg in query1)
{
    Console.WriteLine(“ page: “ + pg.ip + “ “ + pg.count);
}
```

This query scans the HBase table for all records where the IP address equals 88.240.129.183. However, in order to sort the requested pages, then group and count them, the query uses the standard Enumerable extensions against an array in memory. Considering the potential size of the results from a simple table scan, the current approach of implementing only the innermost-where operator is probably inefficient. It also doesn’t make use of MapReduce, almost the whole point in using HBase, especially considering that the example query is tailor-made for MapReduce.

Using MapReduce in a LINQ query

Since the LINQ provider is already calling Java, it should be simple to write a MapReduce implementation in Java using the HBase table mapper and reducer classes. It also gives us an excuse to write some Java code.

```
public class FrequencyMapRed {

    private static final byte[] family = Bytes.toBytes(“details”);
    private static final byte[] ipCol = Bytes.toBytes(“ip”);
    private static final byte[] pageCol = Bytes.toBytes(“page”);

    static class Mapper1 extends TableMapper<Text, IntWritable>
    {
        private static final IntWritable one = new IntWritable(1);
        private Text txt = new Text();
        @Override
        public void map(ImmutableBytesWritable row, Result values, Context context)
            throws IOException
        {
            String val = new String(values.getValue(family, pageCol));
            txt.set(val);
            try {
                context.write(txt, one);
            } catch (InterruptedException e) {
                throw new IOException(e);
            }
        }
    }

    public static class Reducer1 extends TableReducer<Text, IntWritable, ImmutableBytesWritable>
    {
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException
        {
            int sum = 0;
```

```
                    for (IntWritable val : values) {
                        sum += val.get();
                    }
                    Put put = new Put(Bytes.toBytes(key.toString()));
                    put.add(family, Bytes.toBytes("total"), Bytes.toBytes(sum));
                    context.write(null, put);
                    }
                }
                public static void executeMapRed(String columnToCount
                    , String filterColumn
                    , String filterColumnValue)
                    throws Exception
                {
                Configuration conf = HBaseConfiguration.create();
                conf.set("hbase.zookeeper.quorum", "localhost.localdomain");
                Job job = null;
                try {
                    job = new Job(conf, "Hbase_FreqCounter1");
                    job.setJarByClass(FrequencyMapRed.class);
                    Scan scan = new Scan();
                    scan.setCaching(500);
                    scan.setCacheBlocks(false);
                    SingleColumnValueFilter svcFilter = new SingleColumnValueFilter(family
                        , Bytes.toBytes(filterColumn)
                        , CompareFilter.CompareOp.EQUAL
                        , Bytes.toBytes(filterColumnValue));
                    scan.setFilter(svcFilter);
                    TableMapReduceUtil.initTableMapperJob("access_logs", scan, Mapper1.class, Text.class,
                        IntWritable.class, job);
                    TableMapReduceUtil.initTableReducerJob("summary", Reducer1.class, job);
                    job.waitForCompletion(true);
                }
                catch (Exception ex) {
                    throw ex;
                }
            }
        }
```

Notice that the results of the MapReduce computation are placed in a table called *summary*. The table is created once and reused to hold results each time the MapReduce code executes. This table must be created using the HBase shell.

```
hbase(main):008:0> create 'summary', {NAME=>'details', VERSIONS=>1}
0 row(s) in 1.3160 seconds

hbase(main):009:0>
```

Adding the Java MapReduce to the proxy assembly

The above class can be archived to a JAR file, *FrequencyMapRed.jar*. Since the method FrequencyMapRed.executeMapRed() must be called from .NET, the JAR file needs to be added to the class path in the JNBridgePro Visual Studio plug-in. The class can be added to the proxy assembly by using the *Add Classes from Classpath dialog*. Once the proxy assembly is rebuilt, the new Java method that implements the MapReduce frequency computation can be called from .NET.

Modifying the LINQ provider to support MapReduce

The above MapReduce frequency implementation provides the same result as three LINQ query operators: *Where*, *OrderBy* and *GroupBy*. The innermost-where strategy did not implement the *OrderBy* and *GroupBy* operators. Trying to shoe-horn MapReduce to standard query operators is problematic. The problem is solved by optimizing the query into a graph of MapReduce jobs. Some query optimizers for big data implementations do exactly that, but for the scope of this example, it would be simpler to add a new query operator called *frequency*.

Adding a new extension to IEnumerable is easy. The problem is that it would only be available when using the method call form of the query, i.e. *Where().Frequency().Select()*. It would not be available using the query form because C# doesn't supply the syntax. However, the query form does support invoking delegates. Using this strategy leads to the following code found in the source file, *LinqFrequency.cs*.

```
public class HBase
{
    static public Func<AccessRecord, IEnumerable<KeyValuePair<string, int>>> frequency
        = MyFrequency;
    public static IEnumerable<KeyValuePair<string, int>> MyFrequency(AccessRecord recs)
    {
        IEnumerable<KeyValuePair<string, int>> results = null;
        try
        {
            results = executeMapReduce(
                (recs.page == “ip” ? “page” : “ip”)
                , recs.page, recs.ipAddress);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        return results;
        }
        public static IEnumerable<KeyValuePair<string, int>> executeMapReduce(
            string frequencyForThisCol
            , string filterThisCol
            , string withThisValue)
        {
            try
            {
                FrequencyMapRed.executeMapRed(frequencyForThisCol, filterThisCol, withThisValue);
            }
            catch (Exception ex)
            {
                throw ex;
            }
            Configuration hbaseConfig = HBaseConfiguration.create();
            hbaseConfig.set("hbase.zookeeper.quorum", "192.168.1.3");
            HTable htable = new HTable(hbaseConfig, "summary");
            Scan scan = new Scan();
            ResultScanner scanner = htable.getScanner(scan);
            Result r;
            string page;
            int count;
            List<KeyValuePair<string, int>> results = new List<KeyValuePair<string, int>>();
            for (r = scanner.next(); r != null; r = scanner.next())
            {
                page = Bytes.toString(r.getRow());
                count = Bytes.toInt(r.getValue(Bytes.toBytes("details"), Bytes.toBytes("total")));
                results.Add(new KeyValuePair<string, int>(page, count));
            }
            return results;
        }
    }
```

The above code defines a delegate that takes type AccessRecord as an argument and returns an IEnumerable wrapping a KeyValuePair type consisting of a string and an integer.

Using the *Where* and *SelectMany* query operators

The Java MapReduce job uses a table scan, in fact it's the same scan used to implement the innermost-where LINQ provider. That means the MapReduce is really providing the same functionality, albeit much more efficiently, as the *OrderBy* and *GroupBy* query operators. Therefore, a LINQ query using the frequency delegate defined above does not require the two operators. In fact, the Where operator is not really needed. However, it would be nice to use the *Where* query operator as a means to define the predicate used by the table scan, even though the scan is running wholly on the Java side.

To support this, the method *HBaseQueryContext.Execute()* needs to be modified to determine if the SelectMany query operator will invoke the frequency delegate. If there's no *SelectMany* operator in the expression tree, or, if there is, but it's not invoking the frequency delegate, then the call to *HBaseHelper.*

GetColumnsFromHBase() is made as before.

However, if the *SelectMany* operator is present and it's invoking the frequency delegate, then a single AccessRecord instance is used to hold the arguments for the MapReduce.

```
ColumnFinder cf = new ColumnFinder(lambdaExpression.Body);
List<string> columns = cf.Columns;

AccessRecord[] records;
if (expression.ToString().Contains("SelectMany")
    && expression.ToString().Contains("HBase.frequency"))
{
    List<AccessRecord> aList = new List<AccessRecord>();
    aList.Add(new AccessRecord(columns.First<string>(), (cf.IsIP ? "ip" : "page")));
    records = aList.ToArray<AccessRecord>();
}
else
{
    // Call HBase and get the results.
    records = HBaseHelper.GetColumnsFromHBase(columns.First<string>(), cf.IsIP, cf.IsPage);
}
```

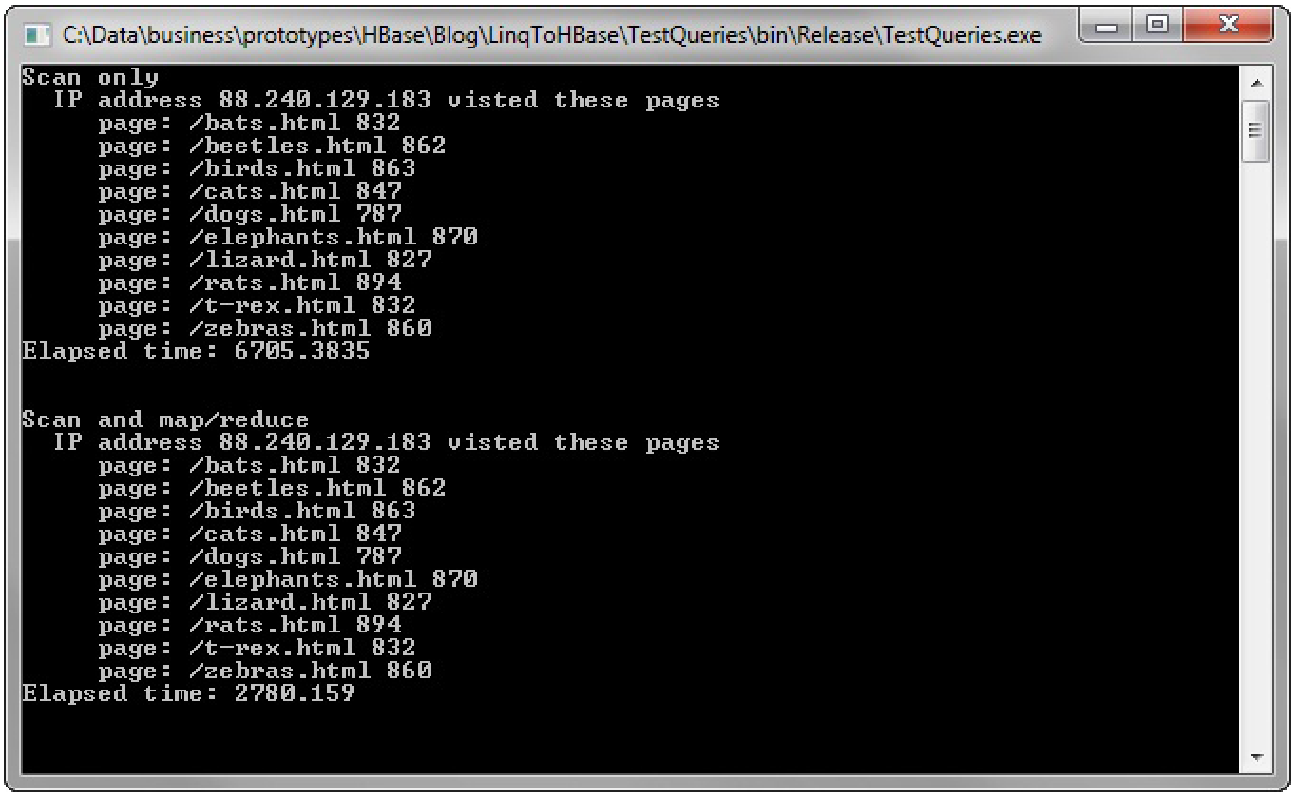
Testing the modified LINQ provider

The test code has two LINQ queries. The first uses the innermost-where strategy, the second uses the SelectMany to invoke the frequency delegate that results in calling the Java MapReduce implementation. The code also provides some bench-marking times on the two queries.

```
static void Main(string[] args)
{

    DateTime startTime;
    DateTime endTime;
    TimeSpan ts;

    QueryableHBaseData<AccessRecord> records =
        new QueryableHBaseData<AccessRecord>();

    Console.WriteLine("Scan only");
    startTime = DateTime.Now;
    var query1 = from record in records
        where record.ipAddress == "88.240.129.183"
        orderby record.page
        group record by record.page into grps
        select new { ip = grps.Key, count = grps.Count() };

    Console.WriteLine(" IP address 88.240.129.183 visted these pages");
    foreach (var pg in query1)
    {
```

```
                Console.WriteLine(“ page: “ + pg.ip + “ “ + pg.count);
            }
            endTime = DateTime.Now;
            ts = endTime - startTime;
            Console.WriteLine(“Elapsed time: “ + ts.TotalMilliseconds);

            Console.WriteLine(“Scan and map/reduce”);
            startTime = DateTime.Now;
            var query2 = from record in records
                where record.ipAddress == “88.240.129.183”
                from count in HBase.frequency(record)
                select count;

            Console.WriteLine(“ IP address 88.240.129.183 visted these pages”);
            foreach (KeyValuePair<string, int> kvp in query2)
            {
                Console.WriteLine(“ page: “ + kvp.Key + “ “ + kvp.Value);
            }
            endTime = DateTime.Now;
            ts = endTime - startTime;
            Console.WriteLine(“Elapsed time: “ + ts.TotalMilliseconds);
        }
```

Here’s the output to the console. It’s obvious that the MapReduce LINQ provider runs in less than half the time of the innermost-where LINQ provider. That’s because the *OrderBy* and *GroupBy* query operators are required to process 8,474 AccessRecord objects.


C:\Data\business\prototypes\HBase\Blog\LinqToHBase\TestQueries\bin\Release\TestQueries.exe

```
Scan only
  IP address 88.240.129.183 visted these pages
     page: /bats.html 832
     page: /beetles.html 862
     page: /birds.html 863
     page: /cats.html 847
     page: /dogs.html 787
     page: /elephants.html 870
     page: /lizard.html 827
     page: /rats.html 894
     page: /t-rex.html 832
     page: /zebras.html 860
Elapsed time: 6705.3835


Scan and map/reduce
  IP address 88.240.129.183 visted these pages
     page: /bats.html 832
     page: /beetles.html 862
     page: /birds.html 863
     page: /cats.html 847
     page: /dogs.html 787
     page: /elephants.html 870
     page: /lizard.html 827
     page: /rats.html 894
     page: /t-rex.html 832
     page: /zebras.html 860
Elapsed time: 2780.159
```

Conclusion

Building a LINQ provider for HBase is relatively straight forward if a table scan using the HBase client API is mapped directly to a LINQ innermost *Where* query operator. However, the Where operation can result in a very large collection of objects residing in memory, therefore subsequent query operators like *OrderBy* and *GroupBy* are inefficient. Using HBase MapReduce to provide the equivalent functionality of the Where, OrderBy and GroupBy query operators in a single distributed, parallel computation is much faster, but it's difficult to parse an expression tree of arbitrary query operators to determine if the query can be optimized to a set of MapReduce jobs. The simple solution is to add a MapReduce operator called *frequency*.

Acknowledgements and resources

Thanks to Steve Willsens and MSDN for the LINQ example, LINQ to TerraServer Provider Sample, which supplied the classes that implement a LINQ expression parser. For an excellent discussion of the LINQ extensions to IEnumerable<T>, visit Raj Kaimal's How LINQ to Object statements work. For a look at a HBase MapReduce implementation, visit Sujee Maniyam's Hbase Map Reduce Example-Frequency Counter.

The source for this example can be downloaded http://www.jnbridge.com/labs/LinqToHBase.zip.