



Qizmt SQL Quick Start Guide

Version: Alpha 1.2
Revision: 2010-02-02

For Mapreduce Developers

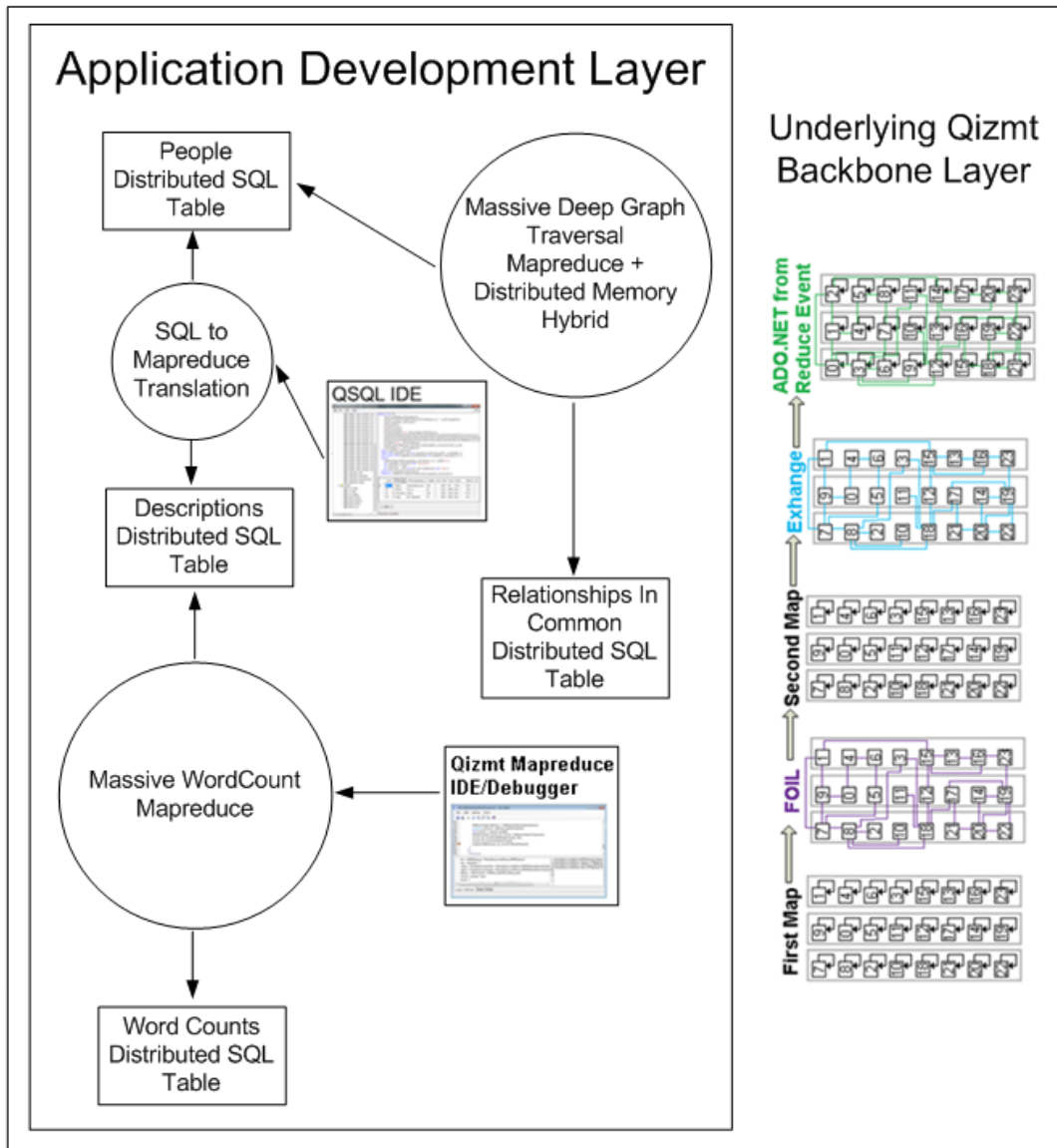
Qizmt C# Mapreduce/ADO.NET/Qizmt SQL Interoperability

Contents

Overview	3
Qizmt SQL Focus.....	3
Qizmt Mapreduce Prerequisite.....	4
Qizmt SQL.....	4
Qizmt SQL Extension Package	4
Qizmt ADO.NET Data Provider Package.....	4
Walkthrough: Qizmt SQL Extension	6
Maximizing Bandwidth Usage between Data Centers with ADO.NET.....	18
System Tables	19
Qizmt SQL INDEXs/Distributed Memory.....	20
Massive Deep Graph Traversal Performance Test.....	20
Distributed Memory Example	21
Qizmt Modeling.....	26
Qizmt Modeling Syntax	26
Qizmt Modeling Example.....	28
Qizmt SQL Administration.....	29

Overview

Qizmt SQL extends the functionality of Qizmt to include SQL operations which speed up the development process for application developers to write comprehensive business logic for large amounts of data in Qizmt clusters. With Qizmt alone, the developer is limited to the mapreduce paradigm and problems that fit into that paradigm. With the Qizmt SQL extension application developers may integrate other commonly useful tasks into their mapreduce pipelines such as massive deep graph processing; delta-only index, and index subset calculations and even distributed memory access from within map and reduce functions.



Qizmt SQL Focus

- Massively distributed applications with easy development with SQL & C# mapreduce
- Leverages the stability of Windows Servers, lower overall development costs than POSIX centric solutions
- Leverages thousands of servers with virtually 0 development cost. For the most part, application developers can abstract the cluster as a single machine with many cores, disk space and memory
- No special tools or skills required other than C# and SQL knowledge

Qizmt Mapreduce Prerequisite

Mapreduce commonly refers to an easy way to optimally process very large amounts of data which is split across a cluster of many machines. Qizmt is a mapreduce framework developed specifically for large clusters of Windows Servers with non-blocking dedicated bandwidth between each machine. Unlike other available mapreducer solutions, Qizmt has a built-in IDE/Debugger and functions as a single solution for rapid mapreduce development/deployment/maintenance. The concepts in this document assume basic understanding of Qizmt mapreducer development.

Here are some guides which may be used to master basic Qizmt mapreduce development:

Qizmt Quick Start Guide	So you are ready to install Qizmt on your development machine? This guide will walk you through every step of installing Qizmt on your development machine, creating and executing mapreducer jobs, as well as specifications on the tools used to develop Qizmt mapreducers, and strategies for optimizing mapreducer pipelines for common data processing tasks.
Qizmt Reference Guide	This guide is a listing of all the objects available to mapreducer jobs which make writing highly efficient mapreducers easy.

Qizmt SQL

The Qizmt SQL or (QSQL) is an infrastructure built on top of Qizmt which translates Standard Query Language (SQL) into set-based mapreducer jobs, MR.DFS commands and directly accessing/muting MR.DFS data. There are two packages available for installation: *Qizmt SQL Extension*, *Qizmt ADO.NET Data Provider*.

Qizmt SQL Extension Package

http://code.google.com/p/qizmt/source/browse/#svn/trunk/Extensions/QueryAnalyzer/QueryAnalyzer_Setup

This package opens up ADO.NET connectivity into a Qizmt cluster. Once installed, any machine of the cluster may be used to launch the Query Analyzer, a windows application for editing and executing SQL commands. This install also includes a suite administration commands and examples. Once installed, mapreducer jobs within the cluster have access to the Qizmt ADO.NET Data Provider and may perform SQL queries during map() and reduce() operations.

Qizmt ADO.NET Data Provider Package

http://code.google.com/p/qizmt/source/browse/#svn/trunk/Extensions/QueryAnalyzer/Qizmt_ADONET_DataProvider

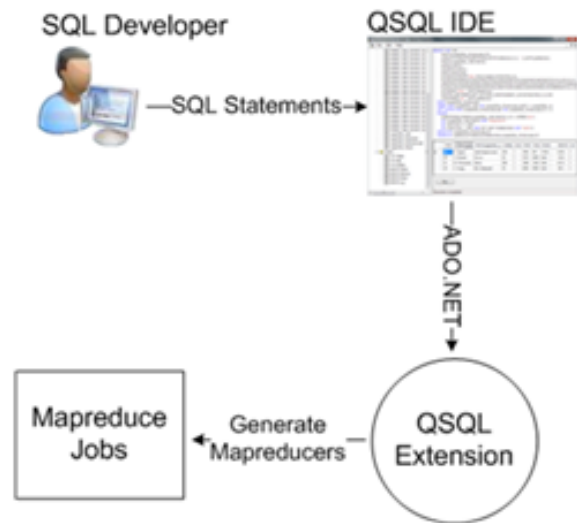
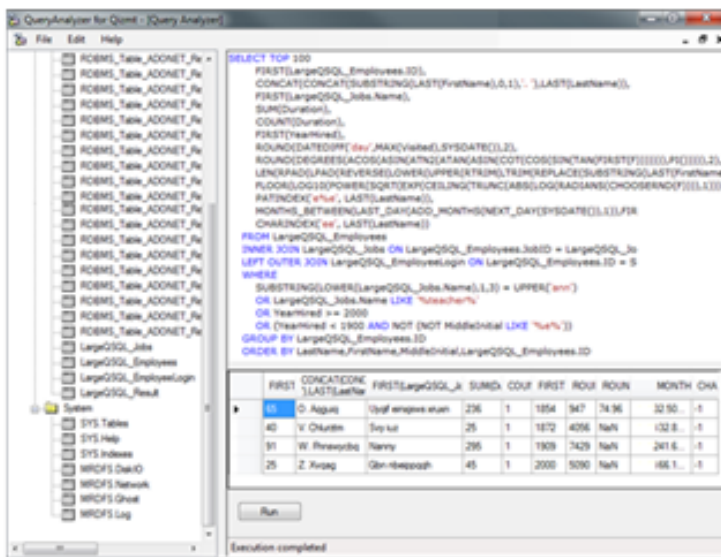
This package should be installed on machines outside of the Qizmt cluster which need ADO.NET connectivity into the cluster. This is used for easily binding applications outside of the cluster to Qizmt data within a cluster. Such applications often include: .NET Web Applications, .NET Forms Applications, .NET Windows Services and MSSQL CLR functions/stored procedures.

Qizmt SQL Translation

There are three forms of SQL translation which occur. These include: translation to mapreduce, translation to MR.DFS administration and translation to direct data access. Much of the SQL92 standard is available for translation to mapreducer and MR.DFS administration, however there is no query optimizer or selection of algorithms for processing SQL. SQL statements are directly mapped to a series of mapreducer jobs which leverage the resources of the cluster. However there are a few SQL-like commands as well which do not translate to mapreducer jobs and may only be executed on sorted tables. Such jobs allow for very low latency queries and updates on sorted (indexed) tables and are often used from within map() and reduce() functions to query SQL tables in distributed memory.



SQL to Mapreduce Translation



When SQL is executed in the Query Analyzer, it is sent via ADO.NET to the Qizmt SQL Extension which runs as a service on all the machines of the cluster. From there it is translated into a series of mapreducer jobs which are executed in the Qizmt cluster. Any resulting record sets are then published back to the ADO.NET Data Provider and displayed in the Query Analyzer's list view. Note however that if the SQL Operation is selecting the Top 10 tuples from an inner join which produces 20TB of data, the mapreducers which do the inner join will produce the entire 20 TB of resulting data before returning the top 10 tuples. This is because SQL is simply translated to a set of mapreducers without any form of query optimizer logic.

In order to facilitate low latency SQL operations, there are also a few SQL-like commands which allow for muting and querying large sorted tables in distributed memory. These operations are described later on in this document.

Walkthrough: Qizmt SQL Extension

In this walk through will demonstrate the interoperability of mapreduce and Qizmt SQL, how tables may be input or output of mapreducer jobs as well as SQL commands.

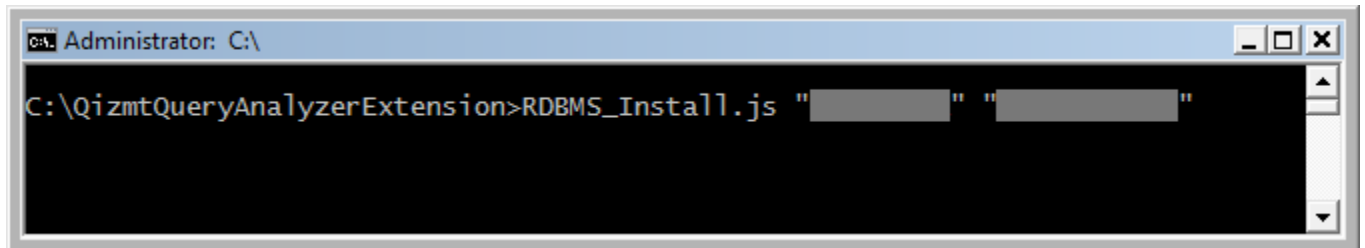
1. Install the latest Qizmt on every machine in the cluster. (Note there is an install.js which may be used to avoid having to run the .msi installer on every machine of the cluster)

The latest mainline source may be found under:
/trunk

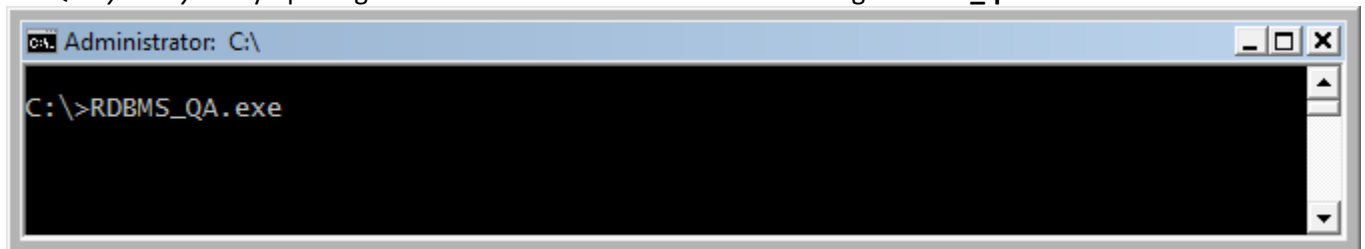
2. Build the Qizmt SQL Extension solution then zip the QueryAnalyzer_Setup folder as QSQLExtension.zip
 - This folder contains all of the Qizmt Jobs, assemblies, services, etc. of the Qizmt SQL Extension.

The latest mainline source may be found under:
/trunk/Extensions/QueryAnalyzer

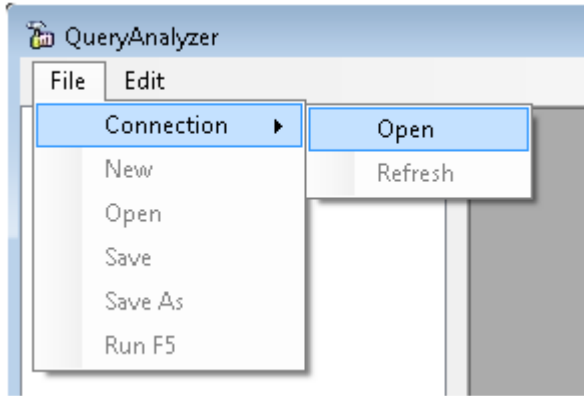
3. Unzip QSQLExtension.zip on any machine in the cluster and navigate to the unzipped folder at the command line. It does not matter which machine in the cluster you use to install, once it is installed it will be available to the entire cluster.
4. Issue the command: **RDBMS_Install.js "<username>" "<password>"**
 - (use the same windows credentials which Qizmt was installed with)



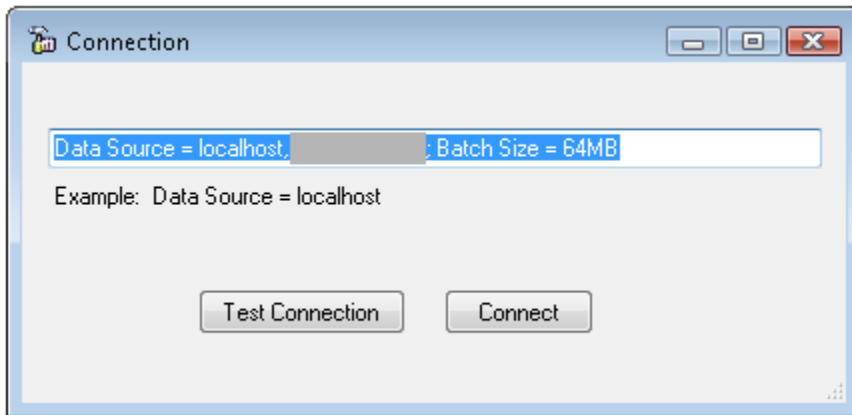
5. Once installation is complete, log off the machine and log back in to any machine of the cluster and connect with the *Query Analyzer* by opening the windows command-line and entering: **RDBMS_qa.exe**



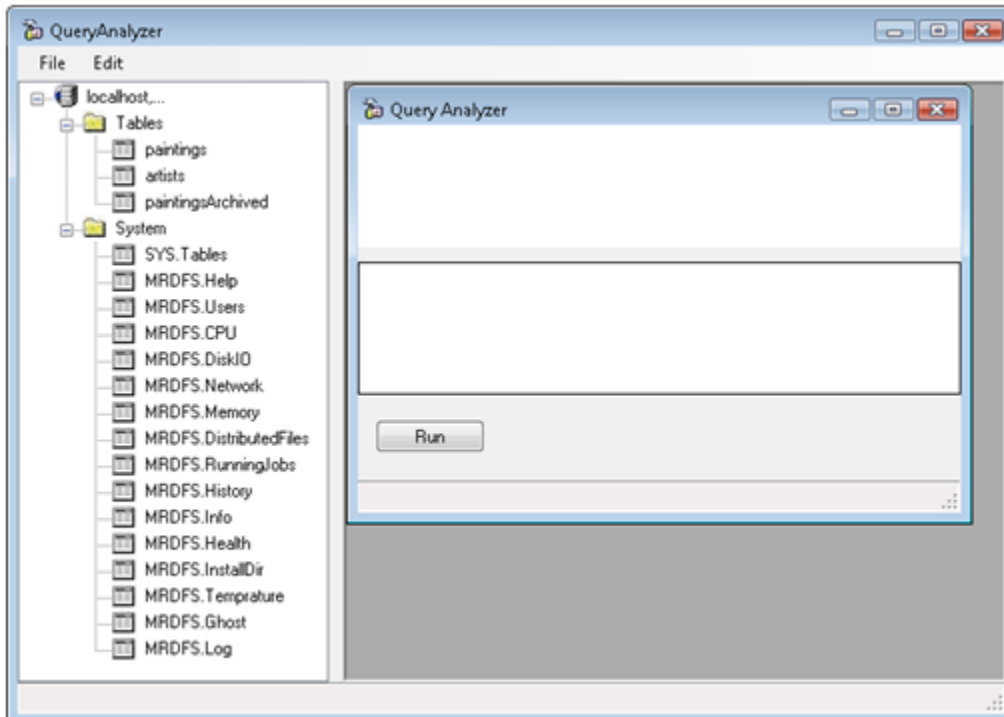
6. Select File->Connection->Open



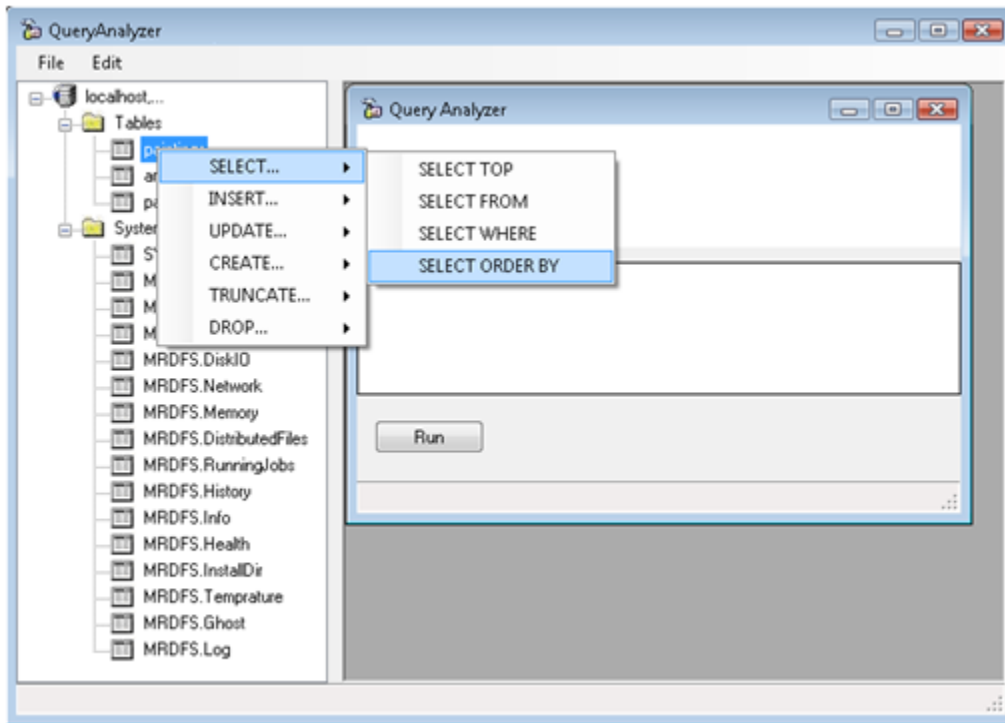
7. The connection window will default to your current computer, listing in the Data Source both localhost and the name of your computer's hostname. Test the connection by clicking **Test Connection** then click **Connect** to connect to the single-machine Qizmt cluster on your desktop.



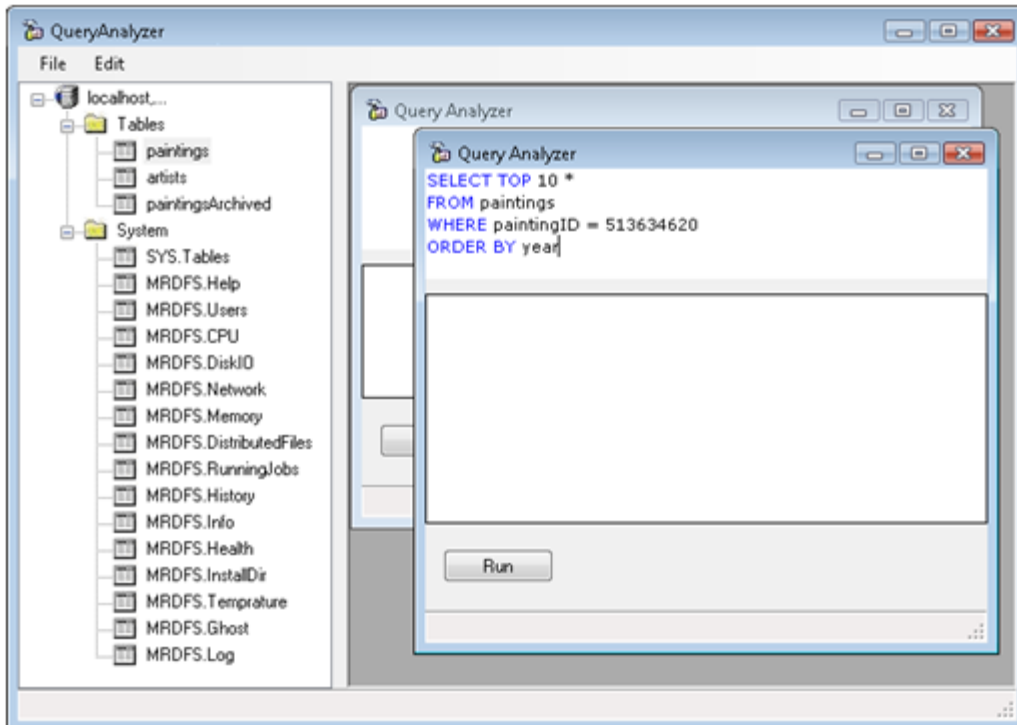
8. Once connected, you will see **System** tables and some sample user tables; **paintings**, **artists** and **paintingsArchive** on the left, and on the right you will see an empty Query Window.

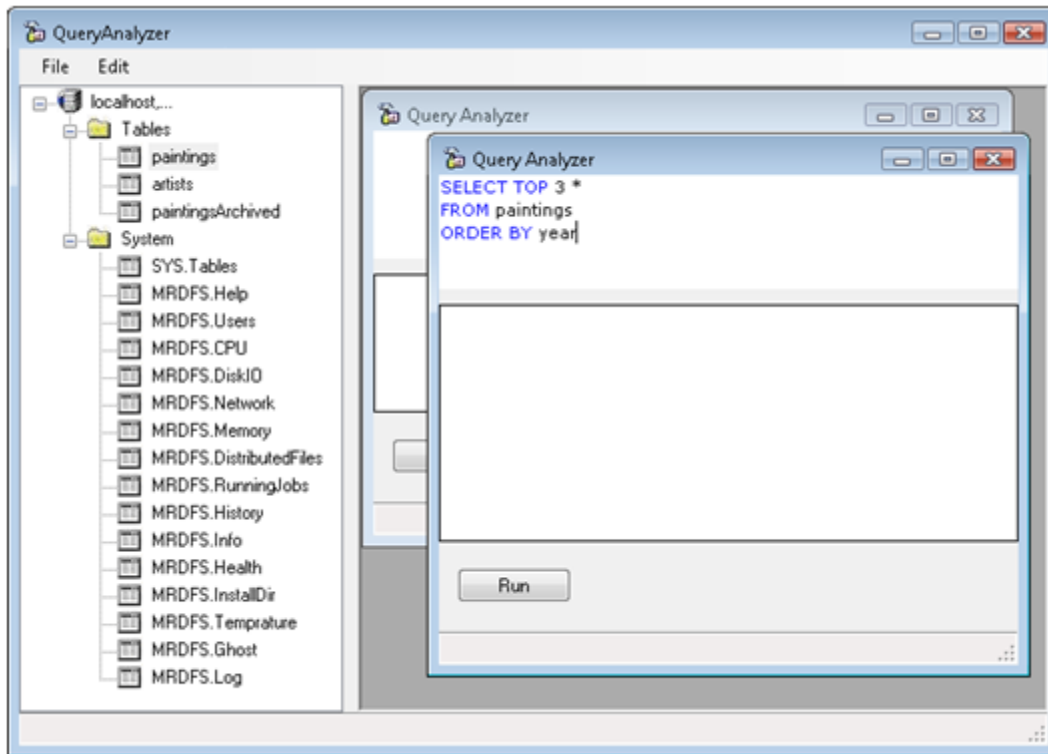


9. View the first three entries in the **paintings** table by right clicking it and navigating **SELECT...->SELECT ORDER BY**

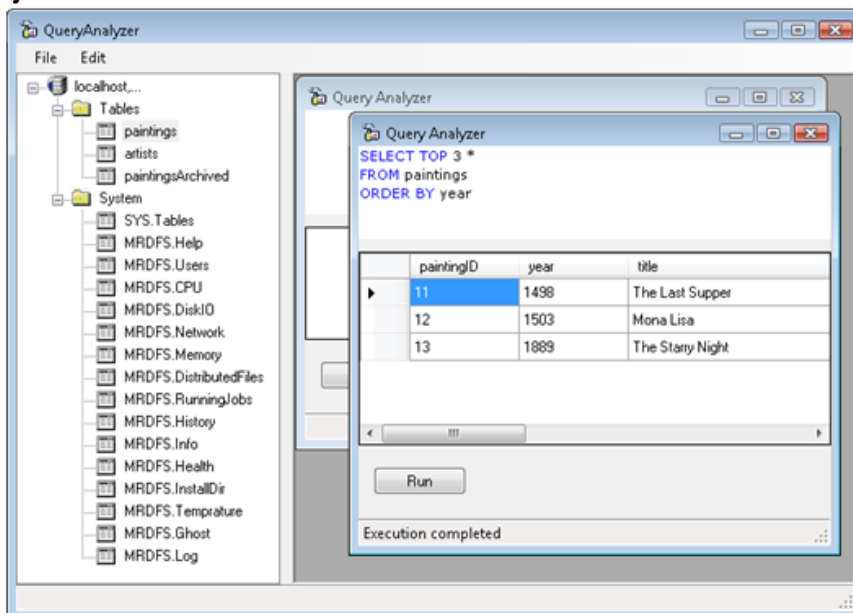


10. This will create an example SQL statement in a new sub-window on the right with a randomly generated **WHERE** clause.



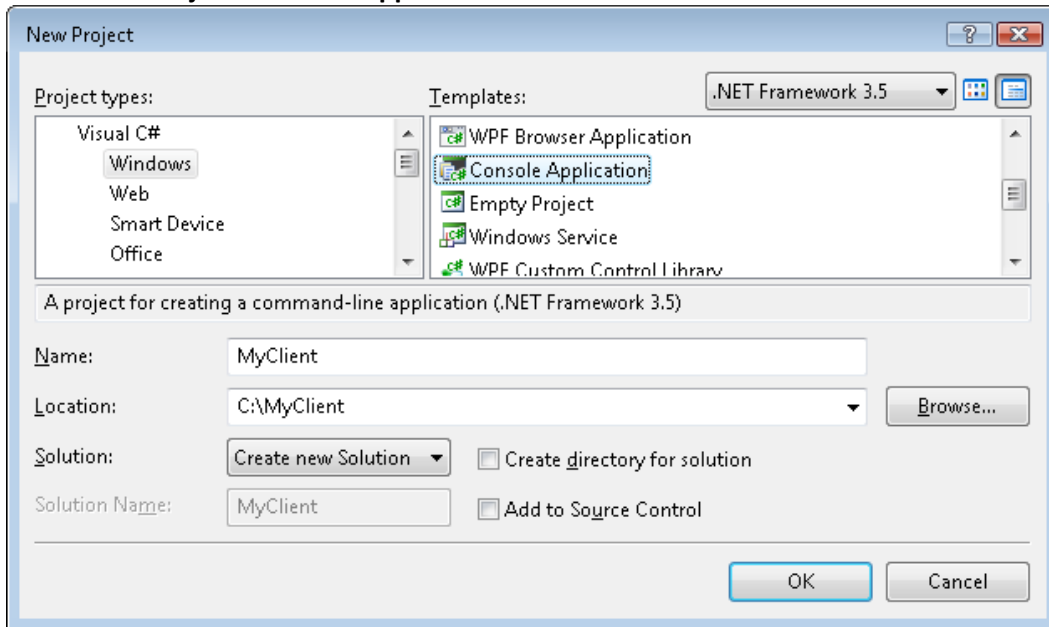
11. Remove the **WHERE** clause and set the **TOP** to 3.

12. Click the **Run** button or hit **F5** to execute the query. Note that these queries have a fairly heavy amount of fixed overhead time, even if working with very small tables as this RDBMS is designed for executing queries over very large clusters of computers. Because the overhead time is fixed, large amounts of data can be streamed in and out via the *Qizmt ADO.NET Data Provider* maximizing the bandwidth between data centers, but for small tasks like this, the overhead is not so negligible. This is because the SQL is translated to mapreduce and a number of network connections are made optimized to large-scale distributed data. While your job is running you can, however, view the progress of the underlying mapreducer jobs by discovering the jobID of the underlying mapreducer jobs **qizmt history -j** and attaching to the standard-out of a particular job with **qizmt -a viewjob <jobID>**



13. Next, let's connect via ADO.NET in Visual Studio.Net. Open a new Visual Studio.Net from a machine within the LAN or VLAN and select

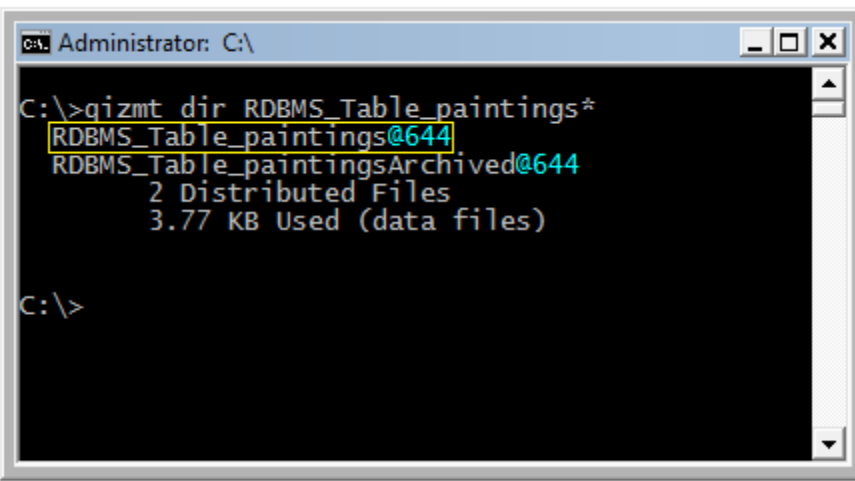
File->New->Project->Console Application



14. Use regular ADO.NET to connect to the cluster and execute SQL from your Main() function.

```
public static void Main(string[] args)
{
    System.Data.Common.DbProviderFactory fact =
        System.Data.Common.DbProviderFactories
            .GetFactory("Qizmt_DataProvider");
    using (System.Data.Common.DbConnection conn = fact.CreateConnection())
    {
        conn.ConnectionString = "Data Source = localhost; Batch Size = 64MB";
        conn.Open();
        System.Data.Common.DbCommand cmd = conn.CreateCommand();
        cmd.CommandText = "SELECT TOP 3 * from paintings ORDER BY year";
        System.Data.Common.DbDataReader reader = cmd.ExecuteReader();
        while (reader.Read())
        {
            Console.WriteLine("-----");
            Console.WriteLine("num = {0}", (int) reader["paintingID"]);
            Console.WriteLine("num = {0}", (int) reader["year"]);
            Console.WriteLine("num = {0}", (string) reader["title"]);
            Console.WriteLine("num = {0}", (double) reader["size"]);
            Console.WriteLine("num = {0}", (long) reader["pixel"]);
            Console.WriteLine("num = {0}", (int) reader["artistID"]);
        }
        Console.WriteLine("-----");
        reader.Close();
        conn.Close();
    }
    System.Console.ReadLine();
}
```

15. Next, to write a *Qizmt mapreducer* to count the words in all the *title* field of every tuple of the *paintings* table. Although mapreduce development is outside the scope of this Guide, it is included in this tutorial for completeness and the steps can be followed as an introduction to mapreduce development. To start, find the *underlying rectangular binary MR.DFS file* for the *paintings* table. Do this by issuing the command:
qizmt dir RDBMS_Table_paintings*



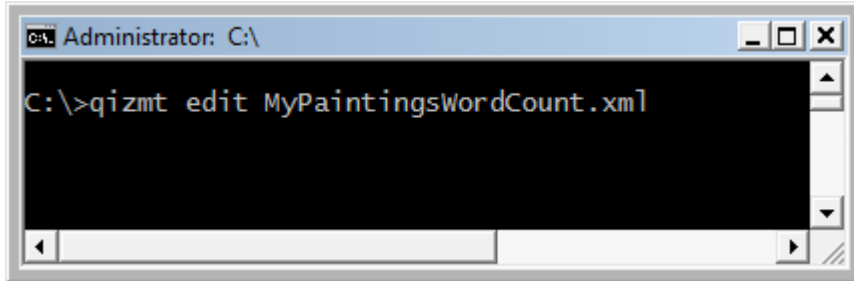
```
C:\>qizmt dir RDBMS_Table_paintings*
RDBMS_Table_paintings@644
RDBMS_Table_paintingsArchived@644
  2 Distributed Files
  3.77 KB Used (data files)

C:\>
```

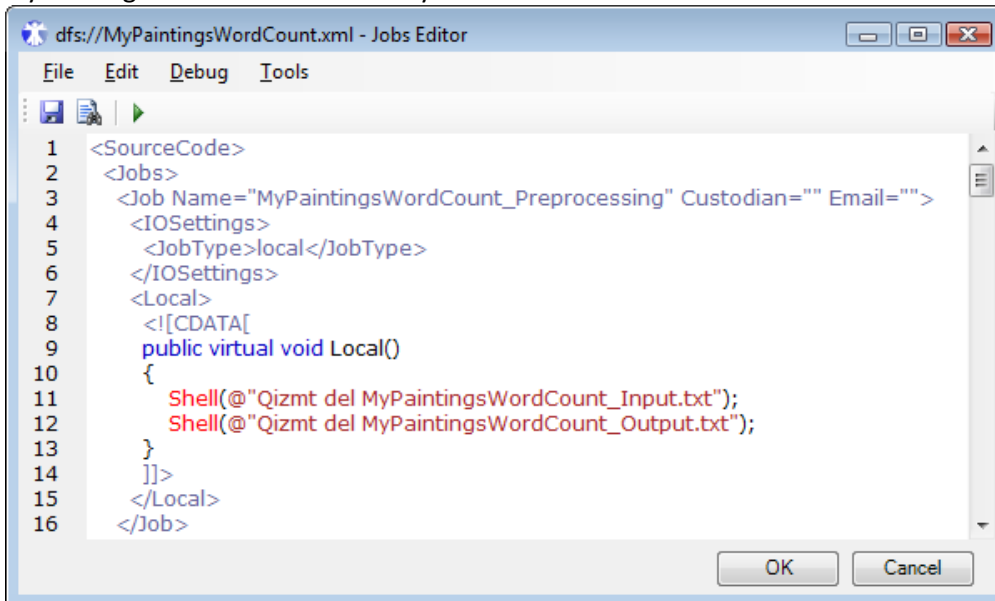
Although, two MR.DFS files matched the wild card, the file to use is in the pattern:
RDBMS_TABLE_<tablename>@<tuplesize>

16. Now that the name of the *underlying rectangular binary MR.DFS file* is known to be *dfs://RDBMS_Table_paintings@644*, a mapreducer can be written to produce another MR.DFS file containing the count of each unique word in the *title* field of every tuple of the *paintings* table.

17. Launch the *Qizmt IDE/Debugger* with a new job file called *MyPaintingsWordCount.xml* by issuing the following command at the windows command prompt: **qizmt edit MyPaintingsWordCount.xml**



18. This will bring up the *Qizmt IDE/Debugger* with a default mapreduce example, unless *MyPaintingsWordCount.xml* already exists in the *MR.DFS*.



19. Replace the entire contents of the mapreducer with the following code:
(Note: key length may contain any combination of comma separated null-able types
These include: *nInt,nLong,nDouble,nDateTime,nChar(m)*)

```

<SourceCode>
<Jobs>
  <Job Name="MyPaintingsWordCount" Custodian="" Email="">
    <IOSettings>
      <JobType>mapreduce</JobType>
      <KeyLength>nChar(25)</KeyLength>
      <DFSInput>dfs://RDBMS_Table_paintings@nInt,nInt,nChar(300),nDouble,nLong,nInt,nDateTime</DFSInput>
      <DFSOutput>dfs://MyPaintingsWordCount_Output@nChar(25),nInt</DFSOutput>
      <OutputMethod>sorted</OutputMethod>
    </IOSettings>
    <Add Reference="RDBMS_DBCORE.dll" Type="dfs"/>
    <Using>RDBMS_DBCORE</Using>
    <MapReduce>
      <Map>
        <![CDATA[
          public virtual void Map(ByteSlice line, MapOutput output)
          {
            DbRecordset dbr = DbRecordset.Prepare(line);
            dbr.GetInt();
            dbr.GetInt();

```

(CONTINUED ON NEXT PAGE)

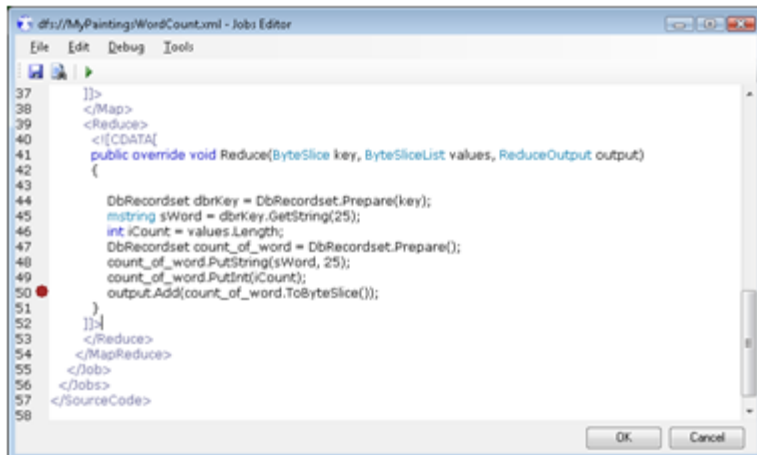
(CONTINUED FROM PREVIOUS PAGE)


```

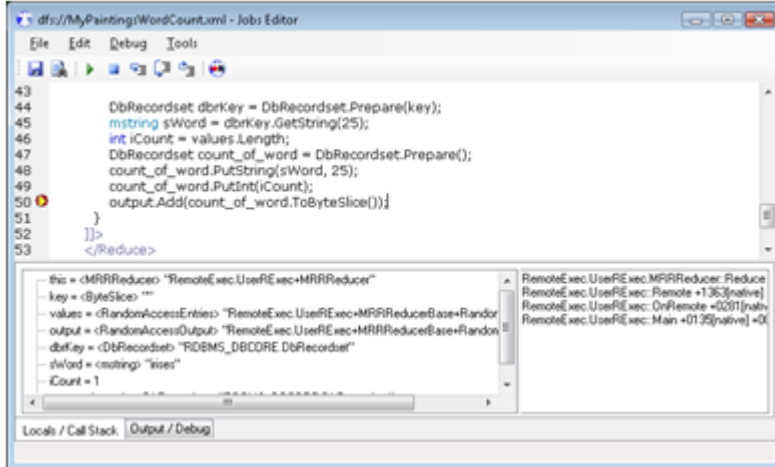
    mstring sTitle = dbr.GetString(300);
    sTitle = sTitle.TrimM('\0');//trim off padding
    mstringarray parts = sTitle.SplitM(' ');
    for(int i=0; i < parts.Length; i++)
    {
        mstring word = parts[i];
        if(word.Length > 0)
        {
            DbRecordset dbrKey = DbRecordset.Prepare();
            dbrKey.PutString(word.ToLowerM(), 25);
            DbRecordset dbrValue = DbRecordset.Prepare();
            dbrValue.PutInt(1);
            output.Add(dbrKey.ToByteSlice(), dbrValue.ToByteSlice());
        }
    }
}
]]>
</Map>
<Reduce>
<![CDATA[
public override void Reduce(ByteSlice key, ByteSliceList values, ReduceOutput output)
{
    DbRecordset dbrKey = DbRecordset.Prepare(key);
    mstring sWord = dbrKey.GetString(25);
    int iCount = values.Length;
    DbRecordset count_of_word = DbRecordset.Prepare();
    count_of_word.PutString(sWord, 25);
    count_of_word.PutInt(iCount);
    output.Add(count_of_word.ToByteSlice());
}
]]>
</Reduce>
</MapReduce>
</Job>
</Jobs>
</SourceCode>


```

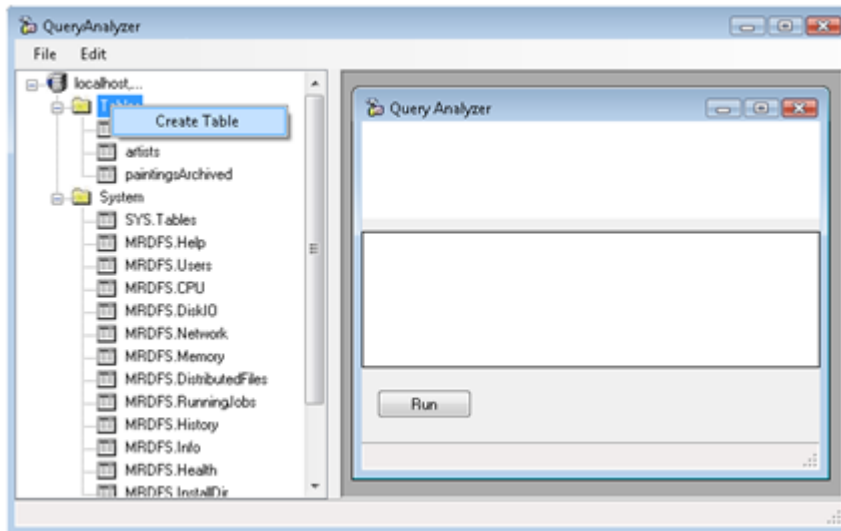
20. Add a breakpoint to the last line of the Reducer, by placing a cursor on the line containing *output.Add(count_of_word);* and then press **F9**.



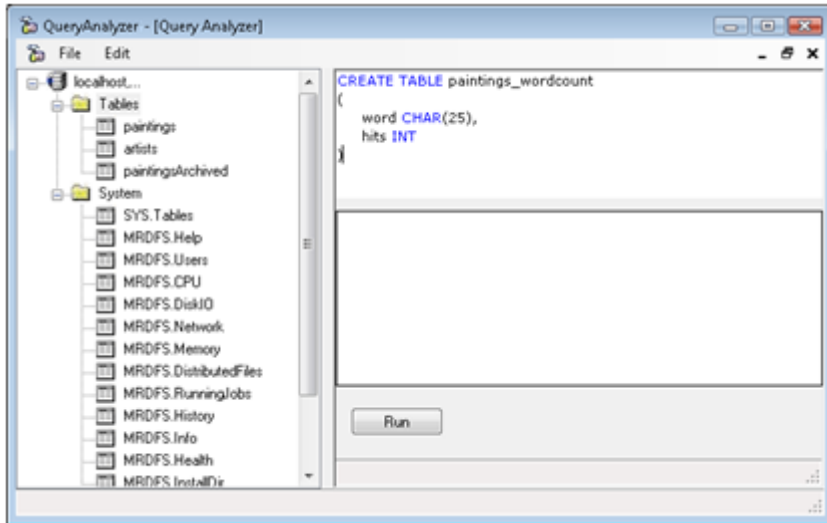
- Run the mapreducer in debug mode by pressing **F5** or by clicking on the  button.



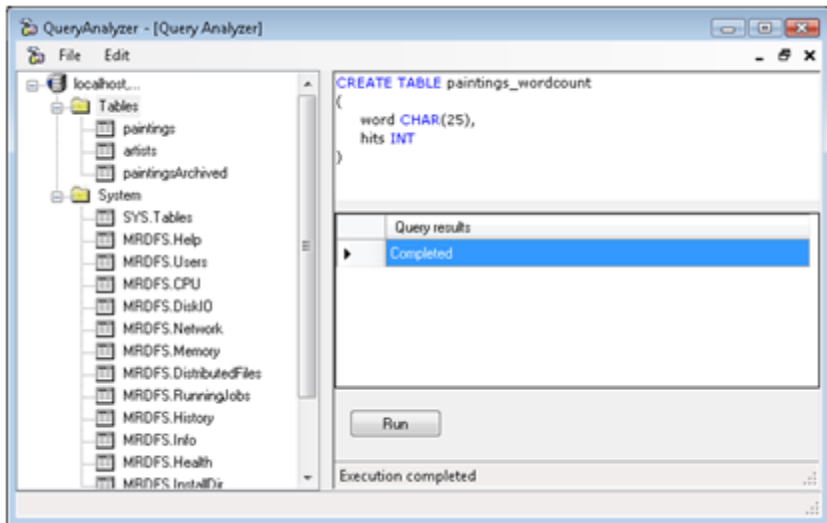
- In *Locals / Call Stack* box you can see the values of the current reducer iteration each time you press **F5**.
- Click on  to remove the break point and then press **F5** again to let the mapreducer finish executing in debug mode.
- Once the mapreducer finishes executing in debug mode, the resulting *MR.DFS* file *dfs://MyPaintingsWordCount_Output* is created containing the resulting words and number of occurrences for each word.
- In the *Query Analyzer*, create a table to store the words and corresponding counts by right clicking on the **Tables** folder and selecting **Create Table**.



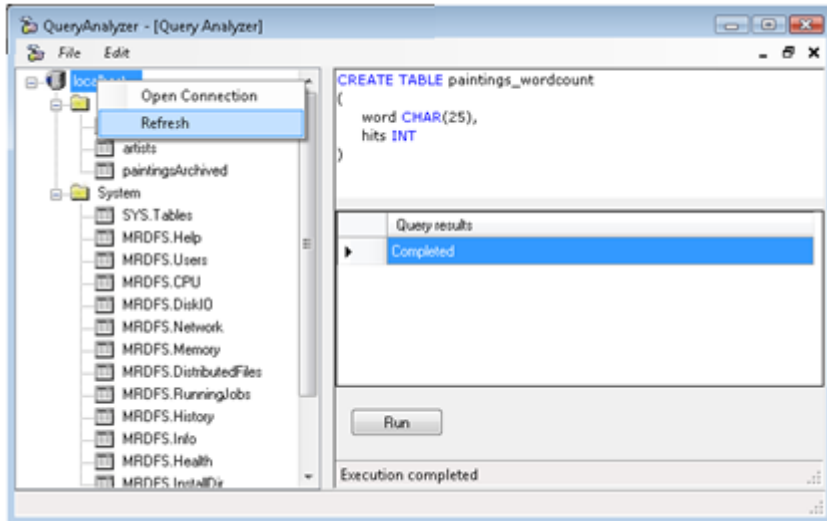
Modify the SQL to create a table with two columns, **word CHAR(25)** to hold the word and **hits INT** to hold the number of occurrences of each word.



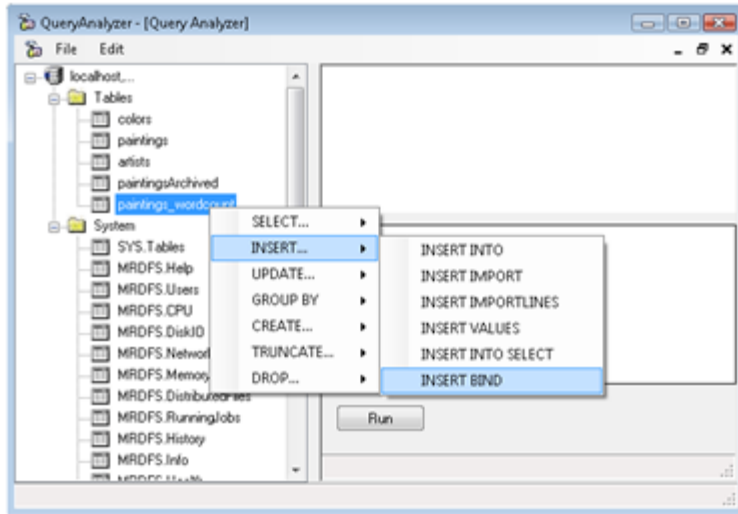
26. Press F5 or click the **Run** button to create the table.



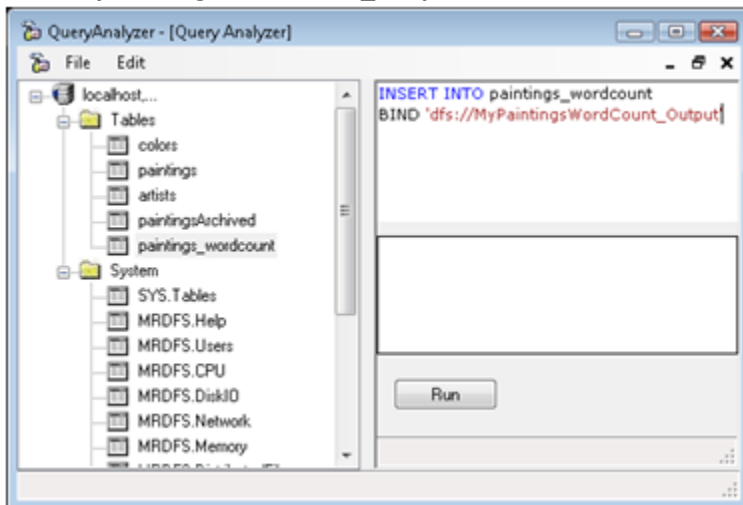
27. Refresh the *Database Objects Tree* by right clicking on localhost,... and selecting **Refresh**.



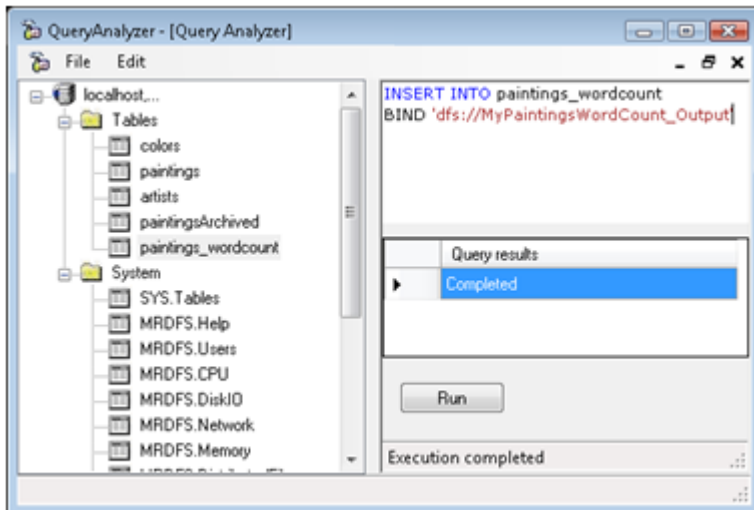
28. The new table **paintings_wordcount** will now show in the *Database Objects Tree*. Right click on **paintings_wordcount** and navigate to **INSERT...** then navigate to **INSERT BIND**.



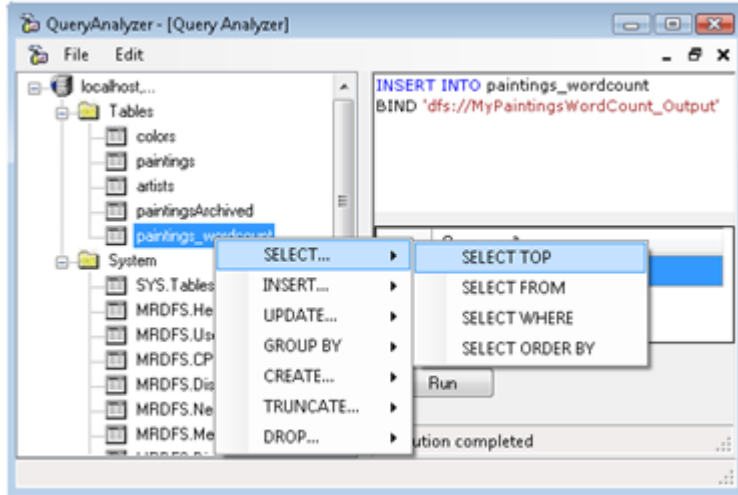
29. Modify the SQL to import the words and numbers of occurrences of each word from the **dfs://MyPaintingsWordCount_Output** file in MR.DFS into the newly created SQL table **paintings_wordcount**.



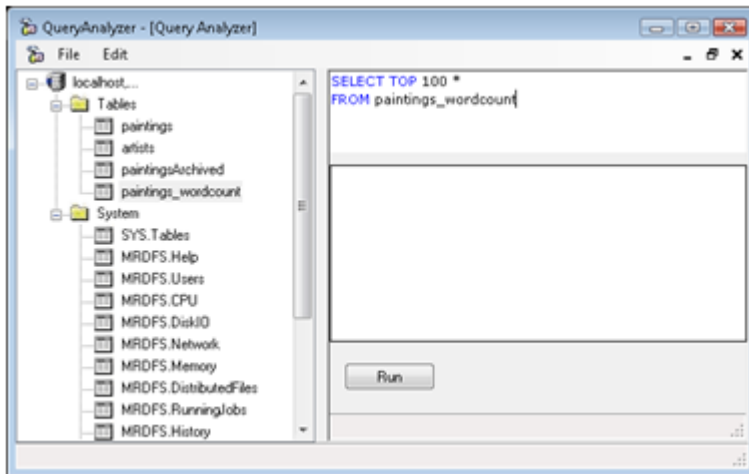
30. Press **F5** or click on the **Run** button to execute the import.



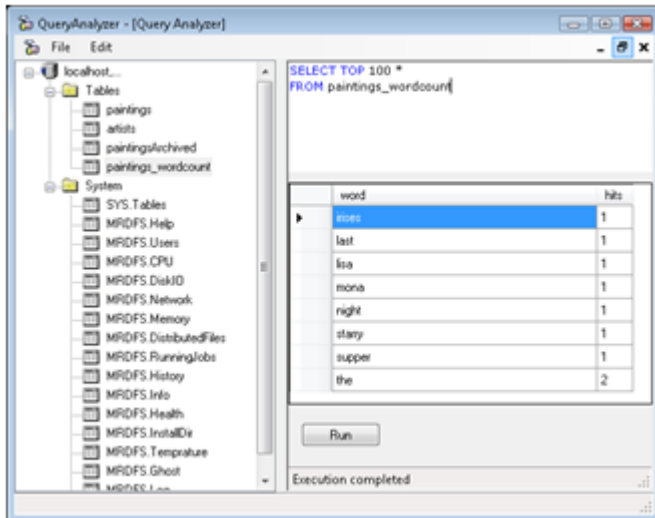
- View the new contents of the **paintings_wordcount** table by right clicking on it in the *Database Objects Tree* and navigating to **SELECT...** then to **SELECT TOP**.



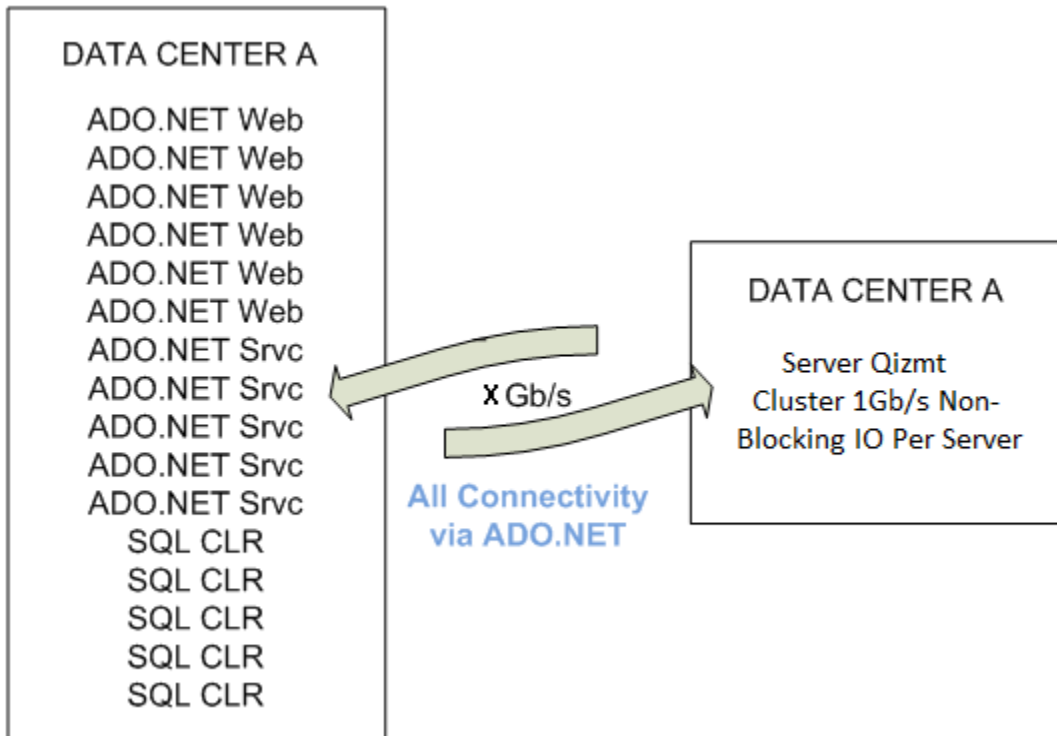
- Modify the SQL to select the top 100 rows from the **paintings_wordcount** table. The data was already sorted by the word column in the mapreducer so you do not need an **ORDER BY** clause unless you want to resort rows before displaying the results.



- Press **F5** or click the **Run** button to execute the query.



Maximizing Bandwidth Usage between Data Centers with ADO.NET



Given each server in a data center has a 1, 2 or 5 Gb/s network card, it is generally not possible for a one server on each data center to max out the bandwidth between the data centers. This problem is easily solvable by having every ADO.NET client list most or all of the host names of the target cluster. The *Qizmt ADO.NET Data Provider* will automatically orchestrate the connectivity for optimal bandwidth use. In this way, the clients writing ADO.NET applications to talk to Qizmt Clusters with SQL do not have to worry about which servers in the remote cluster are up, or which servers in the remote cluster are busy. Rather, they may specify most or all of the hostnames in the remote cluster and let the *Qizmt ADO.NET Data Provider* manage failover and bandwidth maximization and to avoid bottlenecks.

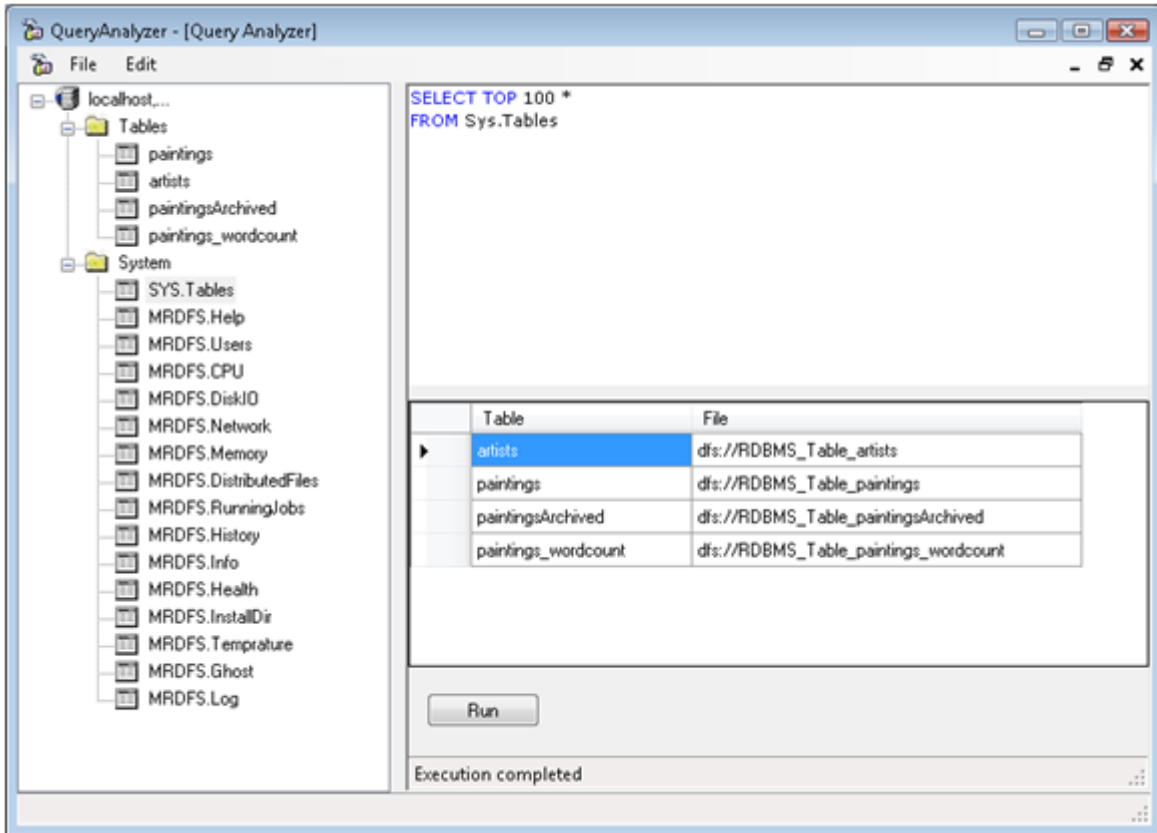
```
System.Data.Common.DbProviderFactory fact =
    System.Data.Common.DbProviderFactories
        .GetFactory("QueryAnalyzer_DataProvider");
using (System.Data.Common.DbConnection conn = fact.CreateConnection())
{
    conn.ConnectionString = "Data Source = MACHINE0,MACHINE1,MACHINE2,MACHINE3,
        MACHINE4,MACHINE5,MACHINE6,MACHINE7,MACHINE8,
        MACHINE9,MACHINE10,MACHINE11,MACHINE12,MACHINE13,
        MACHINE14,MACHINE15,MACHINE16,MACHINE17,MACHINE18,
        .
        .
        .
        MACHINE99; Batch Size = 64MB";

    conn.Open();
}
```

In this example code, all of the machines in a Qizmt Cluster are listed in the connect string. The ADO.NET Data Provider will select one of these machines on each connect, and if that machine is unavailable, it will failover to another one to use as an entry point in the cluster. Outside of the Qizmt cluster, MSSQL CLR's, Windows Services, or Web Applications connecting to the same Qizmt Cluster listing the same hostnames in their connection string will also exhibit the same benefits. Connecting to a Qizmt Cluster with ADO.NET, of course, requires that the cluster be installed with the *Qizmt SQL Extension* first. The ADO.NET client on the other hand needs only install the *Qizmt ADO.NET Data Provider*.

System Tables

Sys.Tables contains a list of all of the user tables along with the underlying *rectangular binary file* in *MR.DFS*. When data developers are communicating with mapreduce developers, this can be used to locate the underlying data of SQL tables for mapreduce processing.



The rest of the System Tables may be used to perform diagnostics on the cluster.

MRDFS.Help	Information and usage on all Qizmt commands.
MRDFS.Users	Users currently logged into the machines of the cluster.
MRDFS.CPU	Perfmon statistics about the CPU across the cluster.
MRDFS.DiskIO	Perfmon statistics about the disk IO across the cluster.
MRDFS.Network	Perfmon statistics about the network bandwidth use across the cluster.
MRDFS.Memory	Perfmon statistics about the available memory across the cluster.
MRDFS.DistributedFiles	List of files in the underlying MR.DFS
MRDFS.RunningJobs	List of jobs currently running on the cluster.
MRDFS.History	History of commands executed on the cluster.
MRDFS.Info	Hard disk usage / free space per machine in the cluster.
MRDFS.Health	Percent health of the cluster and listing of all machines in cluster with disk failures.
MRDFS.InstallDir	Installation directory of Qizmt for every machine in the cluster.
MRDFS.Temperature	Physical temperature of every machine in the cluster.
MRDFS.Ghost	List intermediate cache data in the cluster which is leaked from a job being aborted early.
MRDFS.Log	Concatenation of error logs from every machine in the cluster.

Qizmt SQL INDEXs/Distributed Memory

It is often useful to perform low latency queries and updates of large tables in distributed memory. In order to facilitate this, a few SQL-like commands are available. This is often useful for performing deep graph traversals, large look-up tables available to mapreducer jobs and integrating periodic updates into large distributed tables in near-real-time pipelines.

This next example generates random data and creates an indexed table. Every core in the cluster is given an even subset of the nodes in the graph; from there they traverse into distributed memory bringing 2 levels of related nodes into the local memory of each process.

Massive Deep Graph Traversal Performance Test

This example was used to perform 2-level graph traversals on various data sets. Here is a summary of the findings for running this test on a 512 core cluster.

The diagram illustrates a four-stage workflow for a SQL Distributed Memory Performance Test:

- First Map:** Shows a grid of nodes being processed.
- FOIL:** A transition step between the first and second maps.
- Second Map:** Shows a second grid of nodes being processed.
- Exchange:** A transition step between the second and third maps.
- ADO.NET from Reduce Event:** The final stage where data is processed by ADO.NET.

Accompanying SQL code snippets include:

```
CREATE INDEX indPaintings
FROM tblPaintings ON artistID

RSELECT * FROM indTest WHERE KEY = 1 OR KEY = 3 ...

BEGIN BULKINSERT
RDELETE FROM indPaintings WHERE KEY = 1
RINSERT INTO indPaintings VALUES (2, 'D') WHERE KEY = 2
RDELETE FROM indPaintings WHERE KEY = 3 AND name = 'Da Vinci'
.
END BULKUPDATE
```

RSELECT stress test results on 512 cores with 64Gb/s memory bandwidth

Maximum Rels per node	Average Rels per node	Nodes In Graph	Runtime	OR Statements per second	Graph Traversals per second
5	2.5	2,000,000,000	2 hours	751,314	2,930,127
5	2.5	4,000,000,000	7 hours	396,825	1,547,619
6,400	3,200	10,000	17 minutes	31,158	99,739,045
300	150	200,000,000	47 hours	176,825	26,701,795

- Each node consisted of an integer with a set of related integers.
- Each node is visited, and a full traversal is made to all related nodes as well as all relationships of relationships.
- In addition to the 2-level traversal, these tests also include the time it takes to evenly partition, index and load the graph into distributed memory from a randomly ordered set of relationships in a cluster with redundancy 2.
- Although these stress tests used random data, we have verified that similar performance holds for highly skewed user data as the physical location of any node in the distributed graph is entirely random.
- Both RSELECT and BULKINSERT failover during execution if a machine is lost during operation.

Distributed Memory Example

This example can be obtained

<SourceCode>

```

<Jobs>
  <Job Name="Cleanup_Previous_Data" Custodian="" Email="">
    <IOSettings>
      <JobType>local</JobType>
    </IOSettings>
    <Add Reference="System.Data.dll" Type="system"/>
    <Using>System.Data</Using>
    <Using>System.Data.Common</Using>
    <Local>
      <![CDATA[
        public virtual void Local()
        {
          //Clean up data from previous run.
          Shell(@"Qizmt del QizmtSQL-RIndexBasicStressTest_Input*.txt");
          Shell(@"Qizmt del QizmtSQL-RIndexBasicStressTest_Output.txt");

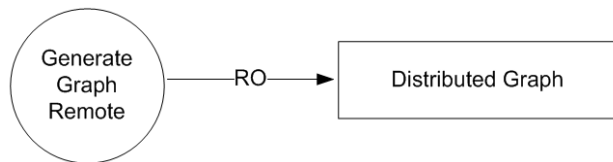
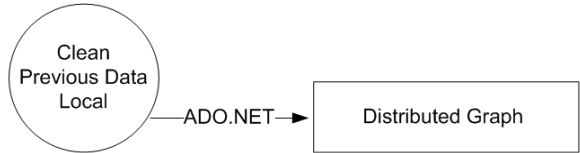
          System.Data.Common.DbProviderFactory fact = DbProviderFactories.GetFactory("Qizmt_DataProvider");

          using (DbConnection conn = fact.CreateConnection())
          {
            try
            {
              conn.ConnectionString = "Data Source = localhost";
              conn.Open();
              DbCommand cmd = conn.CreateCommand();
              cmd.CommandText = "drop index tblRIndexBasicStressTestIndex";
              cmd.ExecuteNonQuery();
              cmd.CommandText = "drop table tblRIndexBasicStressTest";
              cmd.ExecuteNonQuery();
              conn.Close();
            }
            catch
            {
            }
          }
        }
      ]]>
    </Local>
  </Job>
  <Job Name="Generate_Data_Remote" Custodian="" Email="" Description="Generate_Data_Remote">
    <IOSettings>
      <JobType>remote</JobType>
      <DFS_IO_Multi>
        <DFSReader></DFSReader>
        <DFSWriter>dfs://QizmtSQL-RIndexBasicStressTest_Input####.txt</DFSWriter>
        <Mode>ALL MACHINES</Mode>
      </DFS_IO_Multi>
    </IOSettings>
    <Remote>
      <![CDATA[
        public virtual void Remote(RemoteInputStream dfsinput, RemoteOutputStream dfsoutput)
        {
          //Create random graph.

          int max = 10000000;
          int maxasso = 10;
          int maxassoasso = 3; /*3

          if (Qizmt_ExecArgs.Length == 4)
          {
            max = Int32.Parse(Qizmt_ExecArgs[0]);
            maxasso = Int32.Parse(Qizmt_ExecArgs[1]);
            maxassoasso = Int32.Parse(Qizmt_ExecArgs[2]);
          }
      ]]>
    </Remote>
  </Job>

```



Qizmt SQL Quick Guide for Mapreduce Developers

```

/ 2);

List<int> assoasso = new List<int>();
for (int i = 0; i < max; i++)
{
    if (i % StaticGlobals.Qizmt_BlocksTotalCount == StaticGlobals.Qizmt_BlockID)
    {
        int self = i;

        int numberofasso = rnd.Next(1, maxasso + 1);
        for (int ai = 0; ai < numberofasso; ai++)
        {
            int asso = rnd.Next(0, max);
            dfsoutput.WriteLine(self.ToString() + ";" + asso.ToString());
        }
    }
}
]]>
</Remote>
</Job>
<Job Name="Generate_Data_MR" Custodian="" Email="">
<IOSettings>
<JobType>mapreduce</JobType>
<KeyLength>int</KeyLength>
<DFSInput>dfs://QizmtSQL-RIndexBasicStressTest_Input*.txt</DFSInput>
<DFSOutput>dfs://QizmtSQL-RIndexBasicStressTest_InputTable.bin@nInt,nInt</DFSOutput>
<OutputMethod>sorted</OutputMethod>
</IOSettings>
<Add Reference="RDBMS_DBCORE.dll" Type="dfs"/>
<Using>RDBMS_DBCORE</Using>
<MapReduce>
<Map>
<![CDATA[
public virtual void Map(ByteSlice line, MapOutput output)
{
    //Sort graph by left key. This prepares the underlying binary table used for creating the table.
    mstring sLine = mstring.Prepare(line);
    int self = sLine.NextItemToInt(',');
    int asso = sLine.NextItemToInt(',');

    recordset rKey = recordset.Prepare();
    rKey.PutInt(self);

    recordset rValue = recordset.Prepare();
    rValue.PutInt(asso);

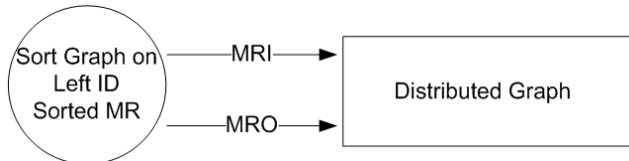
    output.Add(rKey, rValue);
}
]]>
</Map>
<Reduce>
<![CDATA[
public override void Reduce(ByteSlice key, ByteSliceList values, ReduceOutput output)
{
    recordset rKey = recordset.Prepare(key);
    int self = rKey.GetInt();

    for (int i = 0; i < values.Length; i++)
    {
        recordset rValue = recordset.Prepare(values.Items[i]);
        int asso = rValue.GetInt();

        DbRecordset row = DbRecordset.Prepare();
        row.PutInt(self);
        row.PutInt(asso);

        output.Add(row.ToByteSlice());
    }
}
]]>

```



Qizmt SQL Quick Guide for Mapreduce Developers

```

]]>
</Reduce>
</MapReduce>
</Job>
<Job Name="Cleanup_Binary_Data" Custodian="" Email="">
  <IOSettings>
    <JobType>local</JobType>
  </IOSettings>
  <Local>
    <![CDATA[
      public virtual void Local()
      {
        Shell(@"Qizmt del QizmtSQL-RIndexBasicStressTest_Input*.txt");
      }
    ]]>
  </Local>
</Job>
<Job Name="Create_RIndex" Custodian="" Email="">
  <IOSettings>
    <JobType>local</JobType>
  </IOSettings>
  <Add Reference="RDBMS_DBCORE.dll" Type="dfs"/>
  <Add Reference="System.Data.dll" Type="system"/>
  <Using>RDBMS_DBCORE</Using>
  <Using>System.Data</Using>
  <Using>System.Data.Common</Using>
  <Local>
    <![CDATA[
      public virtual void Local()
      {
        //If highly skewed patterns in data, e.g. user generated data,
        //shuffle the physical location of data chunks while keeping the data sorted:
        //Shell(@"Qizmt shuffle <source> <target>");

        System.Data.Common.DbProviderFactory fact = DbProviderFactories.GetFactory("Qizmt_DataProvider");

        using (DbConnection conn = fact.CreateConnection())
        {
          //Batch SQL: Cast sorted rectangular binary data as a SQL table (cheap operation)
          conn.ConnectionString = "Data Source = localhost";
          conn.Open();
          DbCommand cmd = conn.CreateCommand();
          cmd.CommandText = "CREATE TABLE tbIRIndexBasicStressTest (id INT, rid INT)";
          cmd.ExecuteNonQuery();
          cmd.CommandText = "INSERT INTO tbIRIndexBasicStressTest bind 'dfs://QizmtSQL-
RIndexBasicStressTest_InputTable.bin'";
          cmd.ExecuteNonQuery();
          conn.Close();
        }

        using (DbConnection conn = fact.CreateConnection())
        {
          //Batch SQL: Create rindex on already sorted SQL table (cheap operation)
          conn.ConnectionString = "Data Source = localhost";
          conn.Open();
          DbCommand cmd = conn.CreateCommand();
          cmd.CommandText = "CREATE RINDEX tbIRIndexBasicStressTestIndex FROM tbIRIndexBasicStressTest
PINMEMORYHASH ON id";
          cmd.ExecuteNonQuery();
          conn.Close();
        }
      }
    ]]>
  </Local>
</Job>
<Job Name="rindexexample_processdata" Custodian="" Email="">
  <IOSettings>
    <JobType>mapreduce</JobType>
    <KeyLength>int</KeyLength>
    <DFSInput>dfs://RDBMS_Table_tbIRIndexBasicStressTest@nInt,nInt</DFSInput>

```

Qizmt SQL Quick Guide for Mapreduce Developers

```

<DFSOutput>dfs://QizmtSQL-RIndexBasicStressTest_Output.txt</DFSOutput>
<OutputMethod>grouped</OutputMethod>
</IOSettings>
<Add Reference="RDBMS_DBCORE.dll" Type="dfs"/>
<Add Reference="System.Data.dll" Type="system"/>
<Using>RDBMS_DBCORE</Using>
<Using>System.Data</Using>
<Using>System.Data.Common</Using>
<MapReduce>
  <Map>
    <![CDATA[
      public virtual void Map(ByteSlice line, MapOutput output)
      {
        DbRecordset rline = DbRecordset.Prepare(line);
        int id = rline.GetInt();
        int rid = rline.GetInt();

        recordset rkey = recordset.Prepare();
        rkey.PutInt(id);

        recordset rval = recordset.Prepare();
        rval.PutInt(rid);

        output.Add(rkey, rval);
      }
    ]]>
  </Map>
  <ReduceInitialize>
    <![CDATA[
      StringBuilder whrsb = new StringBuilder();

      System.Data.Common.DbProviderFactory fact = DbProviderFactories.GetFactory("Qizmt_DataProvider");
      DbConnection conn = null;
      DbCommand cmd = null;

      int batchsize = 1900;

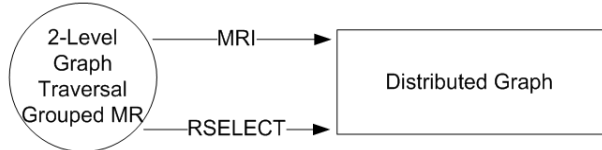
      int reduceittrs = 0;
      public void ReduceInitialize()
      {
        whrsb.Append("RSELECT * FROM tblRIndexBasicStressTestIndex WHERE KEY = -1");

        if (Qizmt_ExecArgs.Length == 4)
        {
          batchsize = Int32.Parse(Qizmt_ExecArgs[3]);
        }

        conn = fact.CreateConnection();
        conn.ConnectionString = "Data Source = " + string.Join(",", StaticGlobals.Qizmt_Hosts) + "; rindex=pooled;
        retrymaxcount = 100; retriesleep = 5000";
        conn.Open();
        cmd = conn.CreateCommand();
      }
    ]]>
  </ReduceInitialize>
  <Reduce>
    <![CDATA[
      public override void Reduce(ByteSlice key, ByteSliceList values, ReduceOutput output)
      {
        {
          recordset rkey = recordset.Prepare(key);
          int self = rkey.GetInt();

          for (int i = 1; i < values.Length; i++)
          {
            recordset rval = recordset.Prepare(values[i].Value);
            int asso = rval.GetInt();
            //Append another OR statement to SQL string
            whrsb.Append(" OR KEY = ").Append(asso.ToString());
          }
        }
      }
    ]]>
  </Reduce>

```



Qizmt SQL Quick Guide for Mapreduce Developers

```
++reduceittrs;
if (reduceittrs >= batchsize || StaticGlobals.Qizmt_Last == true)
{
    string whr = whrsb.ToString();
    if (whr.Length > 0)
    {
        cmd.CommandText = whr;
        //Download batch of tuples from distributed memory into local memory for this reducer
        using (DbDataReader reader = cmd.ExecuteReader())
        {
            while (reader.Read())
            {
                int asso = reader.GetInt32(0);
                int assoasso = reader.GetInt32(1);
            }
            reader.Close();
        }
    }
    whrsb.Length = 0;
    //Start new SQL string for selecting the next batch into local memory from distributed memory
    whrsb.Append("RSELECT * FROM tblRIndexBasicStressTestIndex WHERE KEY = -1");
    reduceittrs = 0;
}
}
]]>
</Reduce>
<ReduceFinalize>
<![CDATA[
    public void ReduceFinalize()
    {
        conn.Close();
    }
]]>
</ReduceFinalize>
</MapReduce>
</Job>
<Job Name="Cleanup_Data" Custodian="" Email="">
<IOSettings>
    <JobType>local</JobType>
</IOSettings>
<Add Reference="System.Data.dll" Type="system"/>
<Using>System.Data</Using>
<Using>System.Data.Common</Using>
<Local>
<![CDATA[
    public virtual void Local()
    {
        Shell(@"Qizmt del QizmtSQL-RIndexBasicStressTest_Input*.txt");
        Shell(@"Qizmt del QizmtSQL-RIndexBasicStressTest_Output.txt");

        System.Data.Common.DbProviderFactory fact = DbProviderFactories.GetFactory("Qizmt_DataProvider");

        using (DbConnection conn = fact.CreateConnection())
        {
            try
            {
                conn.ConnectionString = "Data Source = localhost";
                conn.Open();
                DbCommand cmd = conn.CreateCommand();
                cmd.CommandText = "drop table tblRIndexBasicStressTest";
                cmd.ExecuteNonQuery();
                cmd.CommandText = "drop rindex tblRIndexBasicStressTestIndex";
                cmd.ExecuteNonQuery();
                conn.Close();
            }
            catch
            {
            }
        }
    }
}
]]>
</Local>
</Job>
</IOSettings>
</Add Reference>
</Using>
</Using>
</Job>
</IOSettings>
</Job>
</MapReduce>
</ReduceFinalize>
</Reduce>
]]>
```

```

    }
  ]]>
</Local>
</Job>
</Jobs>
</SourceCode>

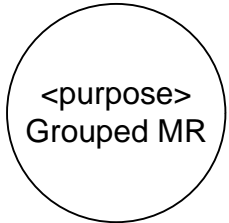
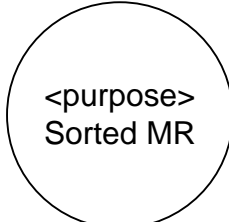
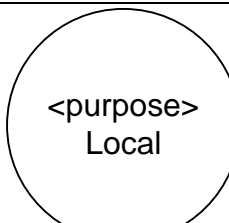
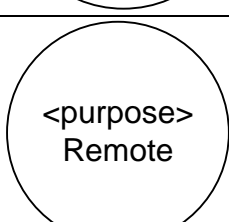

```

Qizmt Modeling

Often when applications developers are designing pipelines, it is helpful to draw up a design for the pipeline. In order to facilitate this we have settled on a simple modeling technique for this. These models are often hand-drawn or created with Microsoft Visio. Qizmt modeling is often used to:

- Design new pipelines
- Communicate design of pipelines already created
- Help achieve capability maturity certifications

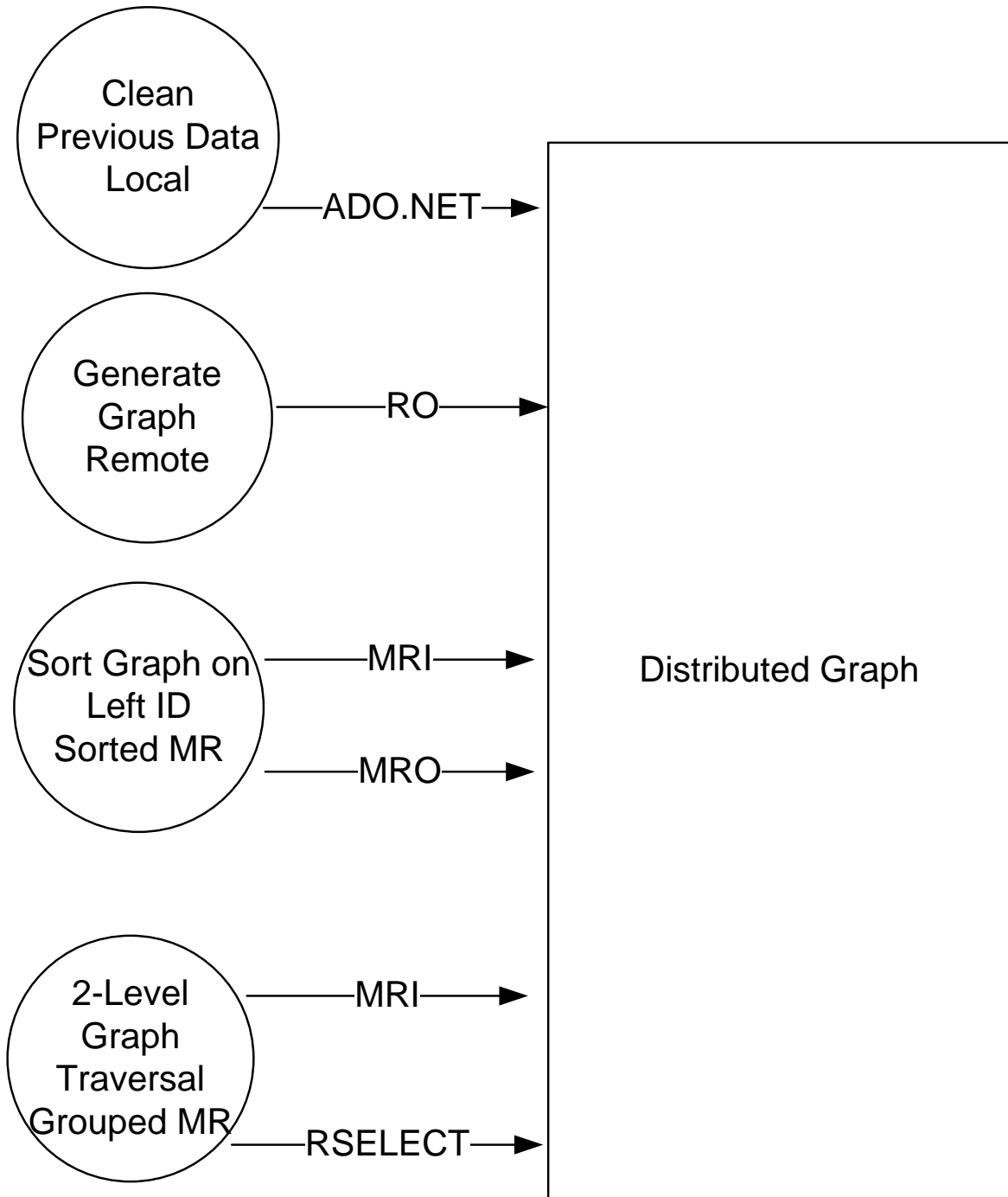
Qizmt Modeling Syntax

Concept	Narrative
	Grouped mapreduce, often used to re-pivot a large amount of data on some column of its tuples. Applying logic both before and after the data is pivoted on a field in its tuples. (Note: arrows always point to data in this methodology)
	Like grouped mapreducer, but data is fully sorted instead of hashed on the keys such that each reducer gets an even range of sorted data. Often used to sort data into an index for RINDEX creation, however this may also be done with INSERT INTO SELECT which will translate to a sorted mapreducer execution.
	Function that executes on a single core in the cluster.
	Function that executes on a single or multiple cores in the cluster and has a stream both into and out of and MR.DFS file.
	An MR.DFS file or SQL table that has an underlying MR.DFS file.
	Mapreduce Input

<p>————MRI————></p>	
<p>————MRO————></p>	Mapreduce Output.
<p>————RI————></p>	Remote Input
<p>————RO————></p>	Remote Output
<p>————RSELECT————></p>	ADO.NET RSELECT which returns tuples from distributed memory.
<p>————BULK UPDATE————></p>	ADO.NET BULK UPDATE consisting of an ordered series of INSERTs and DELETES to apply to and index in distributed memory.
<p>————ADO.NET————></p>	All other ADO.NET operations besides RSELECT and BULK UPDATE.

Qizmt Modeling Example

A simplified model to quickly describe the business logic of the Distributed Memory Performance Test might look something like this:



Qizmt SQL Administration

The Qizmt SQL administration command is available to every machine in the cluster. If you need to run a killall on qizmt cluster, the Qizmt SQL Extension should be stopped first then restarted afterward. This will prevent ADO.NET request from being served while the underlying Qizmt cluster is being restarted.

Usage:

```
RDBMS_admin <action> [<arguments>]
```

Actions:

```
killall          kill all protocol services
stopall         stop all protocol services
startall        start all protocol services
version         get version of protocol service
viewlog        view log entries
clearlog        clear logs entries
examples        generate built-in examples
rindexfilteringstressstest
                [maxPrimary] [maxAssociations]
                [maxSharedAssociations] [batchSize]
                [-v verbose]
                generate and run rindex filtering stress test
rindexbasicstressstest generate a basic rindex stress test
health          check health of protocol services
```