SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets

Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey,
Darren Shakib, Simon Weaver, Jingren Zhou
Microsoft Corporation
{rchaiken, bobjen, palarson, brams, darrens, sweaver, jrzhou}@microsoft.com

ABSTRACT

Companies providing cloud-scale services have an increasing need to store and analyze massive data sets such as search logs and click streams. For cost and performance reasons, processing is typically done on large clusters of shared-nothing commodity machines. It is imperative to develop a programming model that hides the complexity of the underlying system but provides flexibility by allowing users to extend functionality to meet a variety of requirements.

In this paper, we present a new declarative and extensible scripting language, SCOPE (Structured Computations Optimized for Parallel Execution), targeted for this type of massive data analysis. The language is designed for ease of use with no explicit parallelism, while being amenable to efficient parallel execution on large clusters. SCOPE borrows several features from SQL. Data is modeled as sets of rows composed of typed columns. The select statement is retained with inner joins, outer joins, and aggregation allowed. Users can easily define their own functions and implement their own versions of operators: extractors (parsing and constructing rows from a file), processors (row-wise processing), reducers (group-wise processing), and combiners (combining rows from two inputs). SCOPE supports nesting of expressions but also allows a computation to be specified as a series of steps, in a manner often preferred by programmers. We also describe how scripts are compiled into efficient, parallel execution plans and executed on large clusters.

1. INTRODUCTION

Internet companies store and analyze massive data sets, such as search logs, web content collected by crawlers, and click streams collected from a variety of web services. Such analysis is becoming increasingly valuable for business in a variety of ways, for example, to improve service quality and support novel features, to detect changes in patterns over time, and to detect fraudulent activity.

Due to the size of these data sets, traditional parallel database solutions can be prohibitively expensive. To be able to perform this type of web-scale analysis in a cost-effective manner, several companies have developed distributed data storage and processing systems on large clusters of shared-nothing commodity servers, including Google's File System [8] , Bigtable [3], Map-Reduce [5], Hadoop [1], Yahoo!'s Pig system [2], Ask.com's Neptune [4], and Microsoft's Dryad [6]. A typical cluster consists of hundreds or thousands of commodity machines connected via a high-bandwidth network. It is challenging to design a programming model that enables users to easily write programs that can efficiently and effectively utilize all resources in such a cluster and achieve maximum degree of parallelism.

The *Map-Reduce* programming model provides a good abstraction of group-by-aggregation operations over a cluster of machines. The programmer provides a map function that performs grouping and a reduce function that performs aggregation. The underlying run-time system achieves parallelism by partitioning the data and processing different partitions concurrently using multiple machines.

However, this model has its own set of limitations. Users are forced to map their applications to the map-reduce model in order to achieve parallelism. For some applications this mapping is very unnatural. Users have to provide implementations for the map and reduce functions, even for simple operations like projection and selection. Such custom code is error-prone and hardly reusable. Moreover, for complex applications that require multiple stages of map-reduce, there are often many valid evaluation strategies and execution orders. Having users implement (potentially multiple) map and reduce functions is equivalent to asking users specify physical execution plans directly in database systems. The user plans may be suboptimal and lead to performance degradation by orders of magnitude.

In this paper, we present a new scripting language, SCOPE (Structured Computations Optimized for Parallel Execution), targeted for large-scale data analysis that is under development at Microsoft. Many users are familiar with relational data and SQL. SCOPE intentionally builds on this knowledge but with simplifications suited for the new execution environment. Users familiar with SQL require little or no training to use SCOPE. Like SQL, data is modeled as sets of rows composed of typed columns. Every rowset has a well-defined schema. The SCOPE runtime provides implementations of many standard physical operators, saving users from implementing similar functionality repetitively. SCOPE is being used daily for a variety of data analysis and data mining applications inside Microsoft.

SCOPE is a declarative language. It allows users to focus on the data transformations required to solve the problem at hand and hides the complexity of the underlying platform and implementation details. The SCOPE compiler and optimizer are responsible for generating an efficient execution plan and the runtime for executing the plan with minimal overhead.

VLDB '08, August 24-30, 2008, Auckland, New Zealand.

SCOPE is highly extensible. Users can easily define their own functions and implement their own versions of operators: extractors (parsing and constructing rows from a file), processors (row-wise processing), reducers (group-wise processing), and combiners (combining rows from two inputs). This flexibility greatly extends the scope of the language and allows users to solve problems that cannot be easily expressed in traditional SQL.

SCOPE provides functionality similar to views in SQL. This feature greatly enhances modularity and code reusability. It can also be used to restrict access to sensitive data.

SCOPE supports writing a program using traditional nested SQL expressions or as a series of simple data transformations. The latter style is often preferred by programmers who are used to thinking of a computation as a series of steps. We illustrate the usage of SCOPE by the following example.

Example 1: A QCount query computes search query frequencies: how many times different query strings have occurred. There are several variants of QCount queries, for example, a QCount query may return only the top *N* most frequent queries or it may return queries that have occurred more than *M* times. Nevertheless, all QCount queries involve simple aggregation over a large data set, followed by some filtering conditions.

In this example, we want to find from the search log the popular queries that have been requested at least 1000 times. Expressing this in SCOPE is very easy.

```
SELECT query, COUNT(*) AS count
FROM "search.log" USING LogExtractor
GROUP BY query
HAVING count > 1000
ORDER BY count DESC;

OUTPUT TO "qcount.result";
```

The select command is similar to SQL's select command except that it uses a built-in extractor, LogExtractor, which parses each log record and extracts the requested columns. By default, a command takes the output of the previous command as its input. In this case, the output command writes the result of the select to the file "qcount.result".

The same query can also be written in SCOPE as a step-by-step computation.

```
e = EXTRACT query
    FROM "search.log"
    USING LogExtractor;

s1 = SELECT query, COUNT(*) as count
     FROM e
     GROUP BY query;

s2 = SELECT query, count
     FROM s1
     WHERE count > 1000;

s3 = SELECT query, count
     FROM s2
     ORDER BY count DESC;

OUTPUT s3 TO "qcount.result";
```

The script is also easy to understand. The extract command extracts all query string from the log file. The first select command counts the number of occurrences of each query string. The second select command retains only rows with a count greater than 1000. The third select command sorts the rows on count. Finally, the output command writes the result to the file "qcount.result".

In either case, users do not need to implement any operators or wonder how to efficiently execute the query on a large cluster. The SCOPE compiler and optimizer are responsible for translating a script into an efficient, parallel execution plan.

The rest of the paper is organized as follows. We first give an brief overview of the software platform developed at Microsoft for storing and analyzing massive data sets in Section 2. We present the SCOPE scripting language in more detail in Section 3. In Section 4, we describe other SCOPE components and show how a SCOPE script is compiled, optimized, and executed. Experimental evaluation using TPC-H queries is provided in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2. PLATFORM OVERVIEW

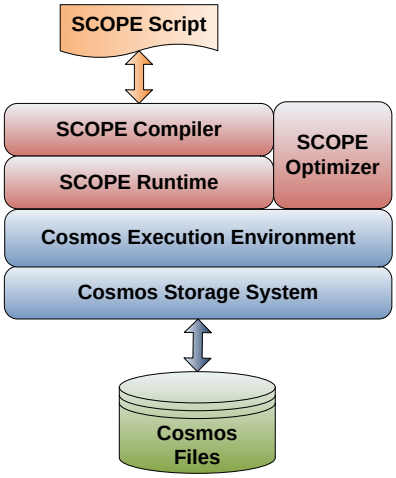

Figure 1: Cosmos Software Layers

Microsoft has developed a distributed computing platform, called Cosmos, for storing and analyzing massive data sets. Cosmos is designed to run on large clusters consisting of thousands of commodity servers. Disk storage is distributed with each server having one or more direct-attached disks.

High-level design objectives for the Cosmos platform include:

1. **Availability**: Cosmos is resilient to multiple hardware failures to avoid whole system outages. File data is replicated many times throughout the system and file meta-data is managed by a quorum group of $2f+1$ servers so as to tolerate f failures.
2. **Reliability**: Cosmos is architected to recognize transient hardware conditions to avoid corrupting the system. System components are check-summed end-to-end and apply mechanisms to crash faulty components. The on-disk data is periodically scrubbed to detect corrupt or bit rot data before it is used by the system.
3. **Scalability**: Cosmos is designed from the ground up to be a scalable system, capable of storing and processing petabytes of data. Storage and compute capacity is easily increased by adding more servers to the cluster.
4. **Performance**: Cosmos runs on clusters comprised of thousands of individual servers. Data is distributed among the

servers. A job is broken down into small units of computation and distributed across a large number of CPUs and storage devices, significantly reducing job completion times.

5. **Cost**: Cosmos is cheaper to build, operate and expand, per gigabyte, than traditional approaches to the same problem. It is far more cost-effective to utilize large numbers of low-cost servers to tackle these types of massive compute problems, as opposed to buying a smaller number of expensive large-scale servers.

Figure 1 shows the main components of the Cosmos platform.

1. *Cosmos storage*: A distributed storage subsystem designed to reliably and efficiently store extremely large sequential files.
2. *Cosmos execution environment*: An environment for deploying, executing, and debugging distributed applications.
3. *SCOPE*: A high-level scripting language for writing data analysis jobs. The SCOPE compiler and optimizer translate scripts to efficient parallel execution plans.

In this paper, we focus on the SCOPE and its components. We briefly describe other Cosmos components in this section. A detailed description of the Cosmos platform is out of the scope of this paper.

2.1 Cosmos Storage System

The Cosmos Storage System is an append-only file system that reliably stores petabytes of data. The system is optimized for large sequential I/O. All writes are append-only and concurrent writers are serialized by the system. Data is distributed and replicated for fault tolerance and compressed to save storage and increase I/O throughput.

A Cosmos Store provides a directory with a hierarchical namespace and stores sequential files of unlimited size. A file is physically composed of a sequence of extents. Extents are the unit of space allocation and are typically a few hundred megabytes in size. A unit of computation generally consumes a small number of collocated extents. Extents are replicated for reliability and also regularly scrubbed to protect against bit rot.

The data within an extent consist of a sequence of append blocks. The block boundaries are defined by application appends. Append blocks are typically a few megabytes in size and contain a collection of application-defined records. Append blocks are stored in compressed form with compression and decompression done transparently at the client.

2.2 Cosmos Execution Environment

The lowest level primitives of the Cosmos execution environment provide only the ability to run arbitrary executable code on a server. Clients upload application code and resources onto the system via a Cosmos execution protocol. A recipient server assigns the task a priority and executes it at an appropriate time. It is difficult, tedious, error prone, and time consuming to program at this lowest level to build an efficient and fault tolerant application.

In Cosmos, applications are programmed against the execution engine that provides a higher-level programming interface and a runtime system that automatically handles the details of optimization, fault tolerance, data partitioning, resource management, and parallelism.

An application is modeled as a dataflow graph: a directed acyclic graph (DAG) with vertices representing processes and edges representing data flows. The runtime component of the execution engine is called the *Job Manager*. The JM is the central and coordinating process for all processing vertices within an application. The primary function of the JM is to construct the runtime DAG from the compile time representation of a DAG and execute over it. The JM schedules a DAG vertex onto the system processing nodes when all the inputs are ready, monitors progress, and, on failure, re-executes part of the DAG.

We refer readers to the Dryad paper [6] for details of an execution engine that is built over the basic primitives of Cosmos. Dryad implements a job manager and a graph building language for composing vertices of computation and edges of communication channels between the vertices.

3. SCOPE Scripting Language

The SCOPE scripting language resembles SQL but with C# expressions. This design choice offers several advantages. Its resemblance to SQL reduces the learning curve for users and eases porting of existing SQL scripts into SCOPE. SCOPE expressions can use C# libraries. Custom C# classes can compute functions of scalar values, or manipulate whole rowsets.

A SCOPE script consists of a sequence of commands. Except for a few auxiliary commands, commands are data transformation operators that take one or more rowsets as input, perform some operation on the data, and output a rowset. Every rowset has a well-defined schema that all its rows must adhere to.

By default, a command takes the result rowset of the previous command as input. As shown in Example 1, the output command in the first SCOPE script takes the result of the previous select command as its input. SCOPE commands can also take named inputs and users can name the output of a command using assignment. The output of a command can be used by subsequent commands one or more times. The second script in Example 1 shows an example of named inputs where e, s1, s2, s3 represent the result of the corresponding command. Named inputs/outputs enable users to write scripts in multiple (small) steps, a style preferred by some programmers.

SCOPE supports a variety of data types, including int, long, double, float, DateTime, string, bool and their nullable counterparts. SCOPE uses C# semantics for nulls, which differs from SQL null semantics. Null compares equal to null. Null compared to a non-null value is always false. Null sorts high. The aggregates ignore nulls in SCOPE.

A script writer can view operators as being entirely serial; mapping the script to an efficient parallel execution plan is handled completely by the SCOPE compiler and optimizer.

3.1 Input and Output

As described earlier, Cosmos provides distributed storage for massive data sets, such as site usage and click streams. While SCOPE is mostly used to analyze data stored in Cosmos files, any type of data store can be used as a data source or data sink.

Input data for a SCOPE script is obtained by means of built-in or user-written extractors. SCOPE provides many standard extractors such as a generic extractor for text files and more specific ones for frequently used log files. However, input data does not have to originate from files; a user could, for example, write an extractor that pulls data from a database system. Similarly, SCOPE outputs data by means of built-in or user-written outputters, regardless of the type of data sink.

```
public class LineitemExtractor : Extractor
{
    enum Cols{l_orderkey=0, l_partkey, l_suppkey, l_linenumber, l_quantity,
              l_extendedprice, l_discount, l_tax, l_returnflag, l_linestatus,
              l_shipdate, l_commitdate, l_receiptdate, l_shipinstruct, l_shipmode, l_comment};

    public override Schema Produce(string[] requestedColumns, string[] args)
    { return new Schema(requestedColumns); }

    public override IEnumerable<Row> Extract(StreamReader reader, Row outputRow, string[] args)
    {   string line;
        int[] requestCol = new int[]{(int)Cols.l_quantity, (int)Cols.l_extendedprice,
                                     (int)Cols.l_discount, (int)Cols.l_tax,(int)Cols.l_returnflag,
                                     (int)Cols.l_linestatus, (int)Cols.l_shipdate};
        while ((line = reader.ReadLine()) != null) {
           string[] tokens = line.Split('|');
           if (tokens[(int)Cols.l_shipdate].Substring(0,10) > "1998-10-01") // filter on ship date
              continue;
           for (int i=0; i < requestCol.Length; i++) {
              if (requestCol [i] <= (int)Cols.l_tax)
                  outputRow[i].Set(double.Parse(tokens[requestCol[i]]));
              else
                  outputRow[i].Set(tokens[requestCol[i]]);
           }
           yield return outputRow;
        }
    }
}
```

Figure 2: Example Implementation of a Custom Extractor

SCOPE provides two customizable commands, EXTRACT and OUTPUT, for users to easily read in data from a data source and write out data to a data sink.

```
EXTRACT column[:<type>] [, …]
FROM < input_stream(s) >
USING <Extractor> [(args)]
[HAVING <predicate>]
```

The extract command extracts data from some data source, usually a Cosmos file or a regular file, and outputs a sequence of rows with the schema specified in the extract clause. Parsing the raw input data and constructing output rows is most often done using one of the standard extractors provided by SCOPE. The optional having clause can be used to quickly filter the output; rows not satisfying the predicate are immediately discarded.

Users can provide custom extractors by extending the C# class "Extractor". Figure 2 shows an example implementation of a custom extractor that extracts some columns from the lineitem file. The "Extract" function is an iterator over output rows, that is, a call returns the next output row. (The C# statement `yield return outputRow` returns the current row and the next call resumes execution from there, not from the beginning of the function.) Users can also provides custom schema information by overwriting the function "Produce", which is called at compile time. In this example, we assumes that records in the lineitem file correspond to rows and columns are separated by '|'. It extracts the columns indicated in "requestCol" from records that have a ship date later than '1998-10-01'.

```
OUTPUT [<input>
   [PRESORT column [ASC | DESC] [, …]]]
TO <output_stream>
[USING <Outputter> [(args)]]
```

Similarly, the output command is used to write data to a Cosmos file, a regular file, or any other data sink. This is the only way that data can exit the system. Formatting a row for output is done by calling the specified outputter, which can be one supplied by the system or by the user through extending the C# class "Outputter". If no outputter is specified a default text outputter is used. The optional presort clause specifies that the input stream is to be sorted before rows are formatted and output.

3.2 Select and Join

SCOPE includes a select command that is patterned on SQL's select statement.

```
SELECT [DISTINCT] [TOP count]
        select_expression [AS <name>] [, …]
FROM {  <input stream(s)> USING <Extractor> |
        {<input> [<joined input> […]]} [, …]
     }
[WHERE  <predicate>]
[GROUP BY <grouping_columns> [, …] ]
[HAVING <predicate>]
[ORDER BY <select_list_item> [ASC | DESC] [, …]]

joined input:
  <join_type> JOIN <input> [ON <equijoin>]

join_type:
  [INNER | {LEFT | RIGHT | FULL} OUTER]
```

Nesting of commands in the from clause is allowed but subqueries are not allowed. The select command can join multiple inputs using inner or outer joins. Join order selection is currently heuristic, preferring equijoins and then joins with other predicates. Predicates are pushed down before joins when possible.

Ten different aggregation functions are currently supported: COUNT, COUNTIF, MIN, MAX, SUM, AVG, STDEV, VAR, FIRST, and LAST. COUNTIF takes a predicate and counts only

```
public class TrimProcessor : Processor
{
   // This method is called at compile time to get column names and types of the output rows
   public override Schema Produce(string[] requestedColumns, string[] args, Schema inputSchema)
   { return new Schema(requestedColumns);  }
   // This function trims all string valued columns and leaves others unchanged.
   public override IEnumerable<Row> Process(RowSet input, Row outRow, string[] args)
   {
     foreach (Row row in input.Rows) {
        row.Copy(outRow);
        for (int i=0; i < row.Count; i++) {
          if(outRow.Schema[i].Type == ColumnDataType.String){
             outRow[i].Set(outRow[i].String.Trim());
          }
        }
        yield return outRow;
     }
   }
}
```

Figure 3: Example Implementation of a Custom Processor

the rows that satisfy the predicate. FIRST (LAST) returns the first (last) row in the group. FIRST and LAST are nondeterministic if the rows within a group are not sorted.

Disallowing subqueries does not reduce the expressive power of the language because outer join is supported. Any subquery can be handled by first computing the result of the main query block and of the subquery block, then outer-joining the subquery result with the main query result (using the predicates correlating the main query to the subquery as join predicates), and finally filtering the result using the predicate referencing the subquery.

Example 2: We rewrite the following SQL query so the subquery is eliminated. The correlation predicate is the equality predicate Sc = Rc in the subquery.

```
SELECT Ra, Rb
FROM R
WHERE Rb < 100
  AND (Ra > 5 OR EXISTS(SELECT * FROM S
                        WHERE Sa < 20
                          AND Sc = Rc))
```

Here is an equivalent script in SCOPE.

```
SQ = SELECT DISTINCT Sc FROM S WHERE Sa < 20;
M1 = SELECT Ra, Rb, Rc FROM R WHERE Rb < 100;
M2 = SELECT Ra, Rb, Rc, Sc
     FROM M1 LEFT OUTER JOIN SQ ON Rc == Sc;
Q  = SELECT Ra, Rb FROM M2
     WHERE Ra > 5 OR Rc != Sc;
```

The first select (SQ) finds the S rows that satisfy the non-correlation predicates, project them onto the S columns used in the correlation predicate (Sc), and eliminates duplicates. The second select (M1) begins processing of the main query by applying predicates that do not involve the subquery. The third select (M2) outer-joins the previous result with the subquery result using the correlation predicate. Outer join is used to guarantee that every row from the main query is retained in the output. M1 rows that do not join with any rows from SQ are null-extended on Sc. The fourth select (Q) computes the final result by applying the predicate referencing the subquery. Note that the predicate Rc != Sc is satisfied only for rows that are null-extended on Sc.

3.3 Expressions and Functions

Scalar expressions and predicates in SCOPE are translated into C# expressions and predicates. This means that all functions and operators available in C# are available in SCOPE. SCOPE also makes it easy for users to write their own functions. The definition of a user-defined function can be included right in the script file.

Example 3: The following script illustrates the use of C# string functions and shows how to write a user-defined function. Columns A, B and C are all of type string and, consequently, any of the C# string functions can be used. The expression A+C denotes string concatenation because both operands are strings. The C# function "Trim" strips white space from the beginning and the end of a string. The user-defined function "StringOccurs" counts the number of occurrences of a given pattern string in an input string. The function is written in C# and included in the script file in a section delimited by #CS and #ENDCS.

```
R1 = SELECT A+C AS ac, B.Trim() AS B1
     FROM R
     WHERE StringOccurs(C, "xyz") > 2

#CS
public static
int StringOccurs(string str, string ptrn)
{
   int cnt=0; int pos=-1;
   while (pos+1 < str.Length) {
     pos = str.IndexOf(ptrn, pos+1) ;
     if (pos < 0) break;
     cnt++;
   }
   return cnt;
}
#ENDCS
```

3.4 User-Defined Operators

For complex data mining and analysis applications, it may sometimes be complicated or impossible to express a desired operation with SQL-like commands alone. Examples include complex data manipulation, customized aggregates, etc.

SCOPE provides three highly extensible commands that manipulate rowsets: PROCESS, REDUCE and COMBINE. Users can write customized operations by extending built-in C# components. The code can also be easily reused in other SCOPE scripts.

The extensible commands provides the same functionality as the *map-reduce* model described in [5] and the operations *map*, *reduce*, and *merge* described in [12]. These extensible commands

```
public class CountReducer : Reducer
{
   public override Schema Produce(string[] requestedColumns, string[] args, Schema upstreamSchema)
   { return new Schema(requestedColumns); }

   public override IEnumerable<Row> Reduce(RowSet input, Row outputRow, string[] args)
   {
      int count = 0;
      foreach (Row row in input.Rows) {
        if (count == 0)
           outputRow[0].Set(row[o].String); // copy over first column
        count++;
      }
      outputRow[1].Set(count.ToString());  // convert to string and return in second col
      yield return outputRow;
   }
}
```

Figure 4: Example Implementation of a Simple Count Reducer

```
public class MultiSetDifference : Combiner
{
   public override IEnumerable<Row> Combine(RowSet left, RowSet right, Row outputRow, string[] args)
   {
      int rightcount = 0;
      Foreach (Row row in right.Rows) {
         rightcount++;
      }
      foreach (Row row in left.Rows) {
         rightcount--;
         if (rightcount < 0) {
           row.Copy(outputRow);
           yield return outputRow;
         }
      }
   }
   public override Schema Produce(string[] requestedColumns, string[] args,
               Schema leftSchema,  string leftTable, Schema rightSchema, string rightTable)
   {
      return new Schema(requestedColumns);
   }
}
```

Figure 5: Example Implementation of a Custom Combiner (computes the difference of two multisets)

complement SELECT, which offers easy declarative filtering, joining, arithmetic, and aggregation. We now describe the three commands and illustrate their usage.

3.4.1 Process

```
PROCESS [<input>]
USING <Processor> [ (args) ]
[PRODUCE column [, …]]
[WHERE <predicate> ]
[HAVING <predicate> ]
```

The process command takes a rowset as input, processes each row in turn, and outputs a sequence of rows. The schema of the output rowset is specified in the produce clause. The where and having clauses are convenient shorthands – they can be used to pre-filter the input rows and post-filter the output rows, respectively, without the need for separate select commands.

The actual work of a process command is done by a user-written processor. The processor receives one input row at a time, performs some computation on the row, and outputs zero, one, or multiple rows. Figure 3 shows a simple example processor which trims all string valued columns and leaves others unchanged.

The process command is a very flexible command that enables users to implement processing that is difficult or impossible to express in SQL. An interesting feature is that the process command can return *multiple* rows per input row, which is highly desirable in some applications and can be used to provide unnesting capabilities. For instance, a process command could be used to break an input search string into a series of words and return one row for each of these words, possibly with some additional information encoded in other columns. Subsequent commands can then perform further analysis of individual words.

3.4.2 Reduce

```
REDUCE [<input> [PRESORT column [ASC|DESC] [, …]]]
ON grouping_column [, …]
USING <Reducer>  [ (args) ]
[PRODUCE column [, …]]
[WHERE <predicate> ]
[HAVING <predicate> ]
```

The reduce command takes as input a rowset that has been grouped on the grouping columns specified in the ON clause, processes each group, and outputs zero, one or multiple rows per group. Input groups are guaranteed to be complete, that is, contain all rows of the group. The Reduce function is called once per group.

Some reducers may require the rows within each group to be sorted on specific columns. This can be achieved with the presort clause. The execution plan generated by SCOPE ensures that the input is sorted as requested, possibly by sorting the input explicitly if not done before. It saves users from having to sort the input

inside the reducer. The produce, where, and having clauses have the same function as in the process command.

Figure 4 shows an example reducer that simply counts the number of rows. The reducer returns rows with a string value in the first column and its count in the second column. One could implement a more sophisticated reducer that sums up counts and returns the percentage of occurrences of each distinct word in the group. The reducer framework is flexible enough to handle such complex aggregates.

3.4.3 *Combine*

```
COMBINE <input1> [AS <alias1>] [PRESORT …]
   WITH <input2> [AS <alias2>] [PRESORT …]
ON <equality_predicate>
USING <Combiner>  [ (args) ]
PRODUCE column [, …]
[HAVING <expression> ]
```

COMBINE is a binary operator that takes two input rowsets, combines them in some way, and outputs a sequence of rows. The two inputs must be grouped in the same way and the user-written combiner receives matching groups as input. The combiner then processes the rows within each matching group in some way to produce output rows. Requiring that inputs be grouped and only allowing rows from matching groups to be combined enables partitioning and distributed processing of the inputs.

Figure 5 shows an implementation of a combiner MultiSetDifference that computes the difference between two multisets (using SQL semantics). Suppose we have two multisets S1 and S2 with columns A, B, and C. To compute the difference between S1 and S2, invoke the combiner as follows:

```
COMBINE S1 WITH S2
ON S1.A==S2.A AND S1.B==S2.B AND S1.C==S2.C
USING MultiSetDifference
PRODUCE A, B, C
```

3.5 Importing Scripts

As described earlier, SCOPE allows the output of a command to be assigned a name. Named outputs can be referenced, possibly multiple times, by other commands within the script. This is equivalent to the concept of "virtual" views in SQL where a view is a named SQL expression that can be referenced in the same way as a table and a reference to a view is resolved by substituting the reference with the view definition (similar to macro expansion).

SCOPE introduces an IMPORT command to extend view functionality across scripts.

```
IMPORT <script_file>
[PARAMS <par_name> = <value> [,…]]
```

The import command reads in the contents of the named script file (at compile time). In the process, parameter references are replaced by the values provided. This is actually more powerful than SQL's view mechanism because SQL views cannot be parameterized.

Example 4: Suppose the file MyView.script contains the following script that extract query strings from a log file, computes the frequency of each query, and retains those with a frequency greater than a specified limit. Parameters are identified by being enclosed by "@@". The keyword EXPORT identifies the result returned by the script.

```
E = EXTRACT query
    FROM @@logfile@@
    USING LogExtractor ;

EXPORT
R = SELECT query, COUNT() AS count
    FROM E
    GROUP BY query
    HAVING count > @@mincount@@;
```

This script invokes the MyView script twice: once to extract data from a query log for January and once for a February query log. It then computes how the frequency of popular queries has changed from January to February.

```
Q1 = IMPORT "MyView.script"
     PARAMS logfile="Queries_Jan.log",
            limit=1000 ;

Q2 = IMPORT "MyView.script"
     PARAMS logfile="Queries_Feb.log",
             limit=1000 ;

JQ = SELECT Q1.query, Q2.count-Q1.count AS diff,
            Q1.count AS jan_cnt,
            Q2.count AS feb_count,
     FROM Q1 LEFT OUTER JOIN Q2
          ON Q1.query == Q2.query
     ORDER BY diff DESC;
```

The innocent-looking import command is an important and distinguishing feature of SCOPE. It provides several important benefits: it enables modularity and information hiding; it provides a mechanism for users to share reusable code; and it can be used to restrict access to sensitive data by only allowing access through provided scripts.

4. SCOPE Execution

In this section, we describe how a SCOPE script is compiled, optimized, and executed and show interactions among different components.

We use the QCount query from Section 1 as a running example-the script is repeated below. The query first extracts the search query string from each log record by using one of the standard SCOPE extractors. It then counts the number of occurrences of each query and returns frequently used keywords (occurs more than 1000 times) in descending order on the count. The result is finally output as a Cosmos file.

```
SELECT query, COUNT() AS count
FROM "search.log"
      USING LogExtractor
GROUP BY query
HAVING count > 1000
ORDER BY count DESC;

OUTPUT TO "qcount.result";
```

The SCOPE script for this query is quite simple. The script goes through the SCOPE compiler and optimizer to generate a parallel execution plan which is then executed on the cluster.

4.1 SCOPE Compilation

The SCOPE compiler parses the script, checks the syntax, and resolves names. It tracks all column definitions and renaming. For each command in the script, the compiler checks that all the columns have been properly defined by the inputs. The result of the compilation is an internal parse tree. SCOPE has an option to translate the parsed tree directly to a physical execution plan using default plans for each command.

A physical execution plan is, in essence, a specification of Cosmos job. The job describes a data flow DAG where each vertex is a program and each edge represents a data channel. A vertex program is a serial program composed from SCOPE runtime physical operators, which may in turn call user-defined functions. All operators within a vertex program are executed in a pipelined fashion, much like the query execution in a traditional database system.

The job manager constructs the specified graph and schedules the execution. A vertex becomes runnable when its inputs are ready. The execution environment keeps track of the state of vertices and channels, schedules runnable vertices for execution, decides where to run a vertex, sets up the resources needed to run a vertex, and finally starts the vertex program.

The translation into an execution plan is performed by traversing the parse tree bottom-up. For each operator, SCOPE has default implementation rules. For example, implementation of a simple filtering operation is a vertex program using SCOPE's built-in physical operator "filter" provided with a function that implements the filtering predicate.

Following the translation, the SCOPE compiler combines adjacent vertices with physical operators that can be easily pipelined into (super) vertices. There are four relationships between any two adjacent vertices: *1:1*, *1:n*, *n:1*, and *n:m*. One of the heuristics used by SCOPE is to combine two vertices with *1:1* relationship. For example, if a "filter" is followed by a "sort", SCOPE combines the two operators into a single (super) vertex and executes "filter"+"sort" in a pipelined fashion.

4.2 SCOPE Optimization

The SCOPE compiler may invoke the optimizer to find a better plan for complex queries. We give a high level overview of the optimizer – further details will be reported in a separate paper.

The SCOPE optimizer is a transformation-based optimizer based on the Cascades framework [7]. Conceptually, the optimizer generates all possible rewritings of a query expression and chooses the one with the lowest estimated cost. Rewritings are generated by applying local transformation rules on query subexpressions, producing substitute expressions logically equivalent to the original subexpression.

Many of the traditional optimization rules from database systems are clearly applicable also in this new context, for example, removing unnecessary columns, pushing down selection predicates, and pre-aggregating when possible. However, the highly distributed execution environment offers new opportunities and challenges, making it necessary to explicitly consider the effects of large-scale parallelism during optimization. For example, choosing the right partition scheme and deciding when to partition are crucial for finding an optimal plan. It is also important to correctly reason about partitioning, grouping, and sorting properties, and their interaction, to avoid unnecessary computations.

4.3 Example Query Plan

We now show the query execution plan used by SCOPE for the QCount query. The extents of the input file are distributed over many machines. For this query, a good strategy is to split the aggregation into multiple layers of partial (local) aggregation followed by a full (global) aggregation.

The plan consists of eight stages, as shown in Figure 6.

1. *Extract*: The input file consists of multiple file extents, distributed across many machines in the cluster. Multiple extractors run in parallel, each one reading part of the file.
2. *Partial aggregation*: In this stage, partial aggregation is applied at the rack level. That is, data from extractors running on machines within the same rack is pre-aggregated to reduce data volume. This exploits knowledge about network topology of the cluster. Partial aggregation can be done either using sorting or hashing and, in this case, it can be applied multiple times, either on a single extent or on groups of extents.
3. *Distribute*: The result from the previous stage is partitioned on the grouping column "query". This brings all (partially aggregated) rows with the same query string into the same partition.
4. *Full aggregation*: Each partition can safely calculate the final aggregation in parallel, again either by sorting or hashing.
5. *Filter*: The fully aggregated rows are then filtered in parallel and any row with a count less than 1000 is discarded.
6. *Sort*: The remaining rows are sorted by count in parallel.
7. *Merge*: The sorted results from all partitions are merged together on a single machine, producing the final result.
8. *Output*: The final result is output as a Cosmos file.

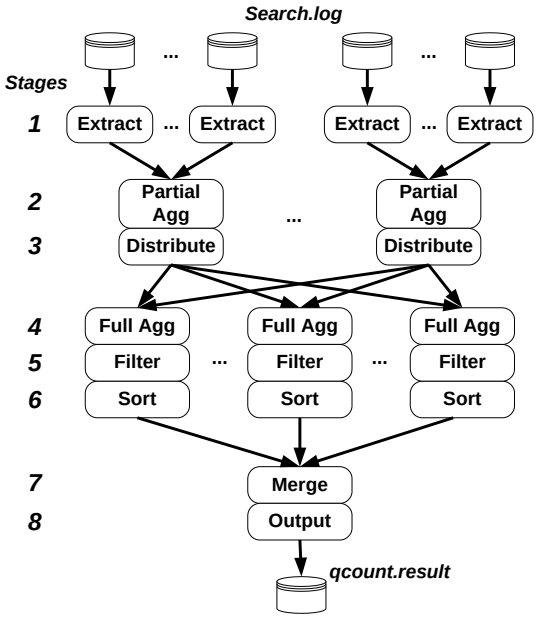

Figure 6: Execution Plan for QCount Query

The execution plan is submitted to Cosmos execution environment which prepares all necessary resources and schedules its execution. As mentioned earlier, the Job Manager monitors progress of all executing vertices. Failing vertices are re-executed a limited number of times and if there are too many failures, the job is terminated.

4.4 Runtime Optimization

Accurate information is not always available at compile time. Therefore, some decisions are better left to run time when additional information is available. We briefly describe some optimizations applied at run time.

A large cluster typically has hierarchically structured network. For example, each rack of machines may have its own dedicated switch and the per-rack switches are then connected to a single common switch. In this architecture, it is important to not overload the common switch and use the per-rack switches as much as possible. In the Cosmos execution environment, the scheduler tries hard to schedule vertices to execute on the same machine as their input data or at least within the same rack as the data. Making such scheduling decisions at optimization time is difficult because completion times for different vertices are hard to predict.

As mentioned earlier, partial aggregation in Stage 3 can be applied multiple times at different levels without changing the correctness of the program. Given that partial aggregation reduces the input data size, it makes sense to aggregate the inputs within the same rack before sending them out, thereby reducing the overall network traffic between racks. The scheduler also has grouping heuristics to ensure that each vertex has not more than a set number of inputs, or a set volume of input data, in order to avoid overloading the I/O system or the vertex.

5. EXPERIMENTAL EVALUATION

SCOPE is used for a wide variety of applications inside Microsoft, including complex relational queries and large-scale data mining applications. SCOPE is highly extensible in that users can easily create customized extractors, processors, reducers, and combiners by extending corresponding built-in components. This provides powerful extensions to the scripting language.

In this section, we show that some complex database OLAP queries can be executed on a large using SCOPE. It illustrates flexibility of the SCOPE language and some fairly complex execution plans. We also run the experiments on clusters with different sizes and demonstrate the scalability of the system.

5.1 Experimental Setup

All experiments were run on a small test cluster of 84 machines. Each machine has two dual-core Xeon processors running at 2GHz, 8 GB of DRAM, and four 500GB SATA disks. All machines run Windows Server 2003 Enterprise X64 Edition SP1.

TPC-H is a well-known data warehouse benchmark. It consists of a suite of business oriented ad-hoc queries. We generated three TPC-H databases with scale factors 10 (10GB), 100 (100GB), and 1000 (1TB). The raw database files were stored as Cosmos files in the cluster. Data is stored as text; each line contains a single row with columns separated by the delimiter '|'. We created custom extractors in C# for different database files in order to extract necessary columns.

5.2 TPC-H Queries

All of the 22 queries can be executed using SCOPE. For some complex queries, SCOPE generates fairly sophisticated parallel execution plans. Due to space limitation, we use Query 1 and 2 as examples to illustrate the implementation details. We focus on the flexibility of the SCOPE language and demonstrate complex but efficient execution plans.

5.2.1 TPC-H Query 1

Query 1 reports the amount of business that was billed, shipped, and returned. It provides multiple aggregated results over the lineitem table. We list the SCOPE script below.

```
// Extract lineitem
//   (The local filter has been pushed into
//    LineitemExtractor)
LINEITEM =
  EXTRACT l_quantity:double,
          l_extendedprice:double,
          l_discount:double,
          l_tax:double, l_returnflag,
          l_linestatus, l_shipdate
  FROM "filesystem://lineitem.tbl"
  USING LineitemExtractor;


// Main query
RESULT =
  SELECT l_returnflag, l_linestatus,
    SUM(l_quantity) AS sum_qty,
    SUM(l_extendedprice) AS sum_base_price,
    SUM(l_extendedprice*(1.0-l_discount)) AS
       sum_disc_price,
    SUM(l_extendedprice*(1.0-l_discount)*
       (1.0+l_tax)) AS sum_charge,
    AVG(l_quantity) AS avg_qty,
    AVG(l_extendedprice) AS avg_price,
    AVG(l_discount) AS avg_disc,
    COUNT(*) AS count_order
  FROM lineitem
  GROUP BY l_returnflag, l_linestatus
  ORDER BY l_returnflag, l_linestatus;


// output result
OUTPUT RESULT TO "tpchQ1.tbl";
```

The script looks very much like a SQL query except that both the input and the output are Cosmos files. As described in Figure 2, the function "LineitemExtractor" extracts from the input table file all necessary columns and convert them to the desired type. For better efficiency, the local filter on l_shipdate has been pushed into the extractor.

The final execution plan is similar to the one for QCount query in Section 4. The plan exploits both partial and full aggregation, and applies partial aggregation as early as possible to reduce the data size. All the machines in the cluster participate in extracting data from the lineitem file. During execution, the intermediate result is partitioned into many small partitions so that each machine is busy working on some partitions. The final results are merged and output as a Cosmos file.

5.2.2 TPC-H Query 2

The previous example showed a SCOPE script written as a traditional SQL block. SCOPE also accepts scripts written in a stepwise fashion where each step performs one or a few small operations like filter, join, group-by, etc. We can implement TPC-H Query 2 in such a way.

Query 2 finds which supplier should be selected to place an order for a given part in a given region. It contains multi-way joins, an aggregation, and a subquery. The script is listed below.

```
// Here are all the extracts that we need.
//   (Local filters have been pushed into
//    the extractors)
REGION =
  EXTRACT r_regionkey, r_name
  FROM "region.tbl"
  USING RegionExtractor;
NATION =
  EXTRACT n_nationkey, n_name, n_regionkey
  FROM "nation.tbl"
  USING NationExtractor;
```

```
SUPPLIER =
  EXTRACT s_suppkey,s_name, s_address,
    s_nationkey, s_phone, s_acctbal, s_commen
  FROM "supplier.tbl"
  USING SupplierExtractor;

PARTSUPP =
  EXTRACT ps_partkey, ps_suppkey, ps_supplycost
  FROM "partsupp.tbl"
  USING PartSuppExtractor;

PART =
  EXTRACT p_partkey, p_mfgr
  FROM "part.tbl"
  USING PartExtractor;


// Join region, nation, and, supplier
//   (Retain only the key of supplier)
RNS_JOIN =
  SELECT s_suppkey, n_name
  FROM region, nation, supplier
  WHERE r_regionkey == n_regionkey
    AND n_nationkey == s_nationkey;


// Now join in part and partsupp

RNSPS_JOIN =
  SELECT p_partkey, ps_supplycost,
         ps_suppkey, p_mfgr, n_name
  FROM part, partsupp, rns_join
  WHERE p_partkey == ps_partkey
    AND s_suppkey == ps_suppkey;


// Finish subquery so we get the min costs
SUBQ =
  SELECT p_partkey AS subq_partkey,
         MIN(ps_supplycost) AS min_cost
  FROM rnsps_join
  GROUP BY p_partkey;


// Finish computation of main query
//   (Join with subquery and join with supplier
//    again to get the required output columns)
RESULT =
  SELECT s_acctbal, s_name, p_partkey,
         p_mfgr, s_address, s_phone, s_comment
  FROM rnsps_join AS lo, subq AS sq, supplier AS s
  WHERE lo.p_partkey == sq.subq_partkey
    AND lo.ps_supplycost == min_cost
    AND lo.ps_suppkey == s.s_suppkey
  ORDER BY acctbal DESC, n_name, s_name, partkey;


// output result
OUTPUT RESULT TO "tpchQ2.tbl";
```

For this complex query, the SCOPE implementation is quite simple, requiring only a few dozens of lines of code. The subquery of the original SQL query is rewritten as an equi-join in the script. We extract all necessary columns from five source table files, using customized extractors (not shown, due to space limitation). All local filters have been pushed into the corresponding extractors. The join result of the five tables, ***RNSPS_JOIN***, is first used to calculate the minimal supply cost per part, which is then joined with the join result ***RNSPS_JOIN*** and supplier table to retrieve all output columns.

The complete execution plan generated by SCOPE is fairly sophisticated. Instead of showing every detail, Figure 7 shows an overview of the execution plan. Both the join result, shown as ***RNSPS_JOIN*** in the figure and the result of the supplier extractor are used twice.

The plan achieves maximum degree of parallelism by utilizing all the machines to extract source table files and partitioning large input data sets so that each machine is busy with computation.

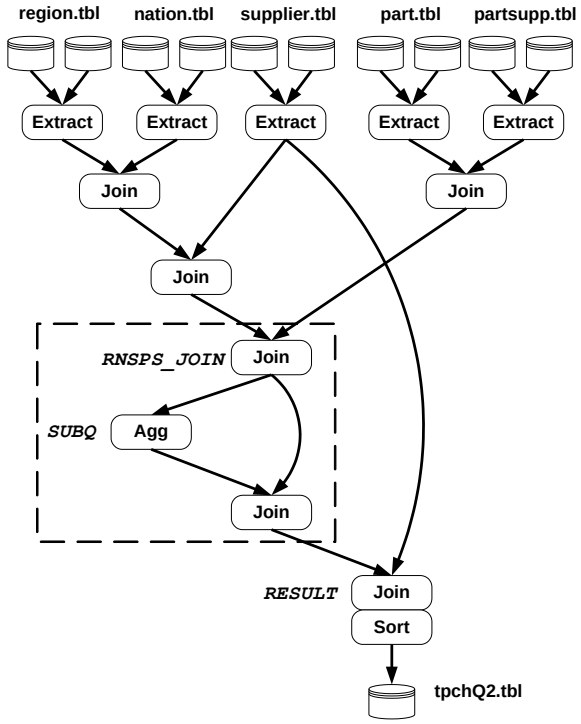

Figure 7: Overall Execution Plan for TPC-H Query 2

We drill into the dashed part in Figure 7 and show the details of the subplan in Figure 8. For presentation purposes, the plan shown uses a degree of parallelism of three. The actual plan's degree of parallelism depends on several factors including the number of machines available, the amount of data processed, etc.

We work through the subplan by stages.

1. *Join*: The join predicate of this stage is s_suppkey == ps_suppkey. Before this stage, both inputs of the join have been partitioned by s_suppkey and ps_suppkey, respectively. Each join vertex takes two matching partitions and generates the local join result.
2. *Partial aggregation*: In this stage, partial aggregation is applied to the local join results at the rack level.
3. *Distribute*: Each local aggregated result is partitioned on the group-by column, p_partkey.
4. *Full aggregation*: Each partition can safely calculate the final aggregation in parallel. This groups all (partially aggregated) rows with the same p_partkey into the same partition.
5. *Distribute*: The fully aggregated result is partitioned by the next join column, subq_partkey, in parallel.
6. *Merge*: Each vertex merges corresponding partitions in parallel in order to prepare a join partition for one join input.
7. *Distribute*: In this stage, the same join result of ***RNSPS_JOIN*** is partitioned by the next join column, p_partkey, in parallel.
8. *Merge*: Each vertex merges corresponding partitions in parallel in order to prepare a join partition for the other join input.

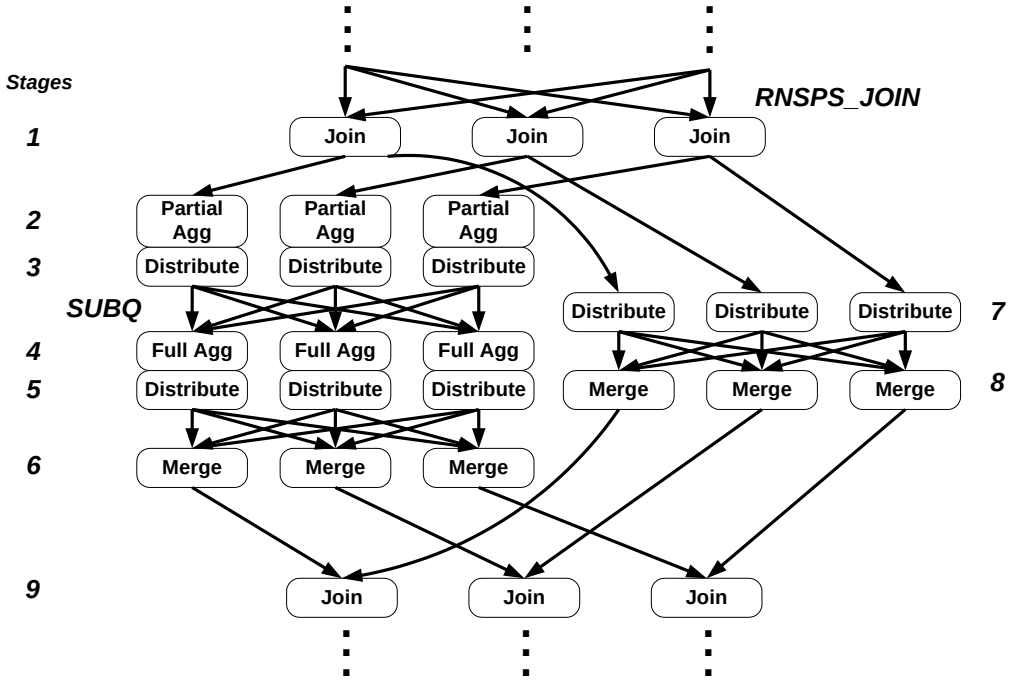

Figure 8: Sub Execution Plan for TPC-H Query 2

9. *Join*: The matching partitions from both inputs are joined in parallel. The join results are consumed by the following stages, also in parallel.

5.3 Scalability

In this section, we show how query performance scales with clusters with different sizes and databases with different scale factors, respectively. We report performance trends rather than actual elapsed times.

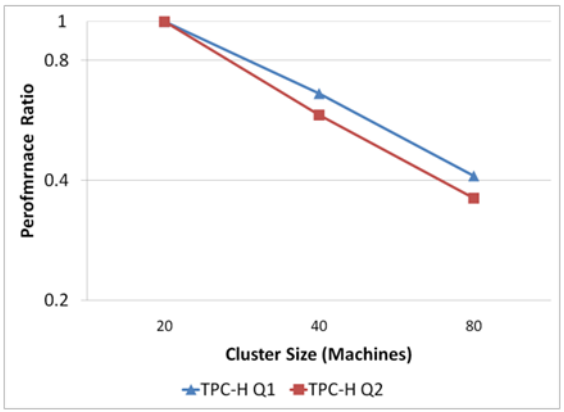

Figure 9: Query Performance with Different Cluster Sizes

In the first experiment, we ran both Q1 and Q2 against the 1TB TPC-H database. We changed the cluster size by limiting the number of machines used for query execution. We use query elapsed times of Q1 and Q2 on a cluster of 20 machines as base lines, respectively, and show performance ratio (elapsed time / base line) for different cluster configurations. As shown in Figure 9 which uses a log scale on the axis of performance ratio, query performance for both queries scales linearly with cluster size.

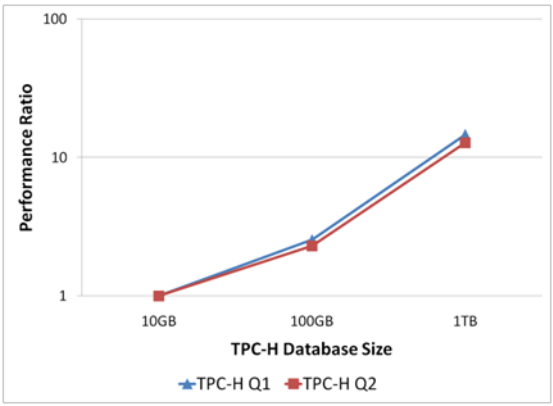

Figure 10: Query Performance with Different Database Sizes

In the second experiment, we ran both queries on the full cluster but against databases of different sizes. We use the elapsed times of Q1 and Q2 against the 10GB TPC-H database as the base lines. In fact, when querying against the 10GB database, since the data file is relatively small, not all the machines on the cluster are utilized. Nevertheless, as shown in Figure 10, query performance for both queries scales linearly with input size.

6. RELATED WORK

SCOPE is heavily influenced by SQL but its target applications and execution environment differ from traditional database systems. SCOPE is designed for easy and efficient processing of massive amounts of data stored in distributed, sequential files. It

provides efficient query processing functionality. The execution strategies used owe much to earlier work on query processing in parallel and distributed database systems [9].

All companies operating internet-scale services have the need to store and process massive data sets and have developed their own system for this purpose. Google popularized the map-reduce programming model. Based on what has been published in the open literature, their software stack consists of Google File System [8] and Bigtable [3] for storage, the MapReduce execution environment [5] with users writing MapReduce applications in C++ or Sawzall [11]. A MapReduce application written in C++ takes many more lines of code than the corresponding application expressed in SCOPE. For example, the word count application used as an example in [5] requires about 70 lines of C++ code but only five or six lines of SCOPE code.

Yahoo! also has a software stack designed for distributed processing of massive data sets. Users write applications in a language called Pig Latin [10] [1]. A Pig Latin program is compiled by the Pig system into a sequence of MapReduce operators that are executed using Hadoop [1], an open-source implementation of MapReduce. Pig Latin is a dataflow language using a nested data model. Each step in a program specifies a single, high-level data transformation. A complex computation is expressed as a series of such transformations. Yahoo! also has a more powerful Map-Reduce-Merge execution environment but it is apparently not the execution environment used by the Pig system.

Both Google and Yahoo! use a MapReduce execution environment. MapReduce is very rigid, forcing every computation to be structured as a sequence of map-reduce pairs. The Cosmos execution environment is significantly more flexible, handling execution of any computation that can be expressed as an acyclic dataflow graph.

7. CONCLUSION

In this paper, we present a new scripting language SCOPE for web-scale data analysis on large clusters of hundreds or thousands of machines. SCOPE has a strong resemblance to SQL – an intentional design choice. The language is high-level and declarative so that the SCOPE compiler and optimizer can optimize SCOPE scripts and improve them over time. All the hardware and implementation details are transparent to users. SCOPE is also highly extensible. Users can easily create customized extractors, processors, aggregators, and combiners by extending corresponding built-in C# components. Such extensions allow users to efficiently solve problems that are otherwise difficult to express in SQL. The parallel execution plans generated by the SCOPE compiler and optimizer fully utilize the cluster. Experiments confirm that query performance scales linearly with cluster and data sizes.

8. ACKNOWLEDGEMENTS

We would like to thank the following people for their contributions to the SCOPE system: Robert Ragno and Mike Schultz for their many contributions to an earlier prototype; Fritz Behr for bravely suffering through early versions of SCOPE; Grace Zhang for regression testing; Achint Srivastava for contributions to the runtime; Daniel Dedu-Constantin for contributions to the design; Andrew Kadatch, Sam McKelvie and the entire Cosmos team for providing the reliable and available substrate on which SCOPE is built; and Nat Ballou for facilitating the technology transfer from Microsoft Research and building the SCOPE team.

9. REFERENCES

[1] Apache. Hadoop. *http://lucene.apache.org/hadoop/*, 2008.

[2] Apache. Pig. *http://incubator.apache.org/pig/*, 2008.

[3] Fay Chang et al, Bigtable: a distributed storage system for structured data, *OSDI* 2006, 205-218.

[4] Chu, L., Tang, H., Yang, T., and Shen, K. 2003. Optimizing data aggregation for cluster-based internet services. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.

[5] Jeffrey Dean, Sanjay Ghemawat: MapReduce: simplified data processing on large clusters. *OSDI* 2004: 137-149.

[6] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, Dennis Fetterly: Dryad: distributed data-parallel programs from sequential building blocks. *EuroSys* 2007: 59-72.

[7] G. Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3): 1995.

[8] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: The Google file system. *SOSP* 2003: 29-43.

[9] Lu, Ooi, Tan, Query Processing in Parallel Relational Database Systems, *IEEE Computer Society Press*, 1994.

[10] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig Latin: A Not-So-Foreign Language for Data Processing, *SIGMOD* 2008.

[11] Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan: Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming* 13(4): 277-298 (2005)

[12] Hung-Chih Yang, Ali Dadsdan, Ruey-Lung Hsiao, D. Stott Parker, Map-Reduce-Merge: simplified relational data processing on large clusters, *SIGMOD* 2007, 1029-1040.

[13] Yahoo! Research, PNUTS - Platform for Nimble Universal Table Storage, *http://research.yahoo.com/node/212*, 2008