SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets

Charul Daudkhane charul.daudkhane@campus.tu-berlin.de
BDASEM 2018
DIMA – TU BERLIN Student Report on the Paper
"SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets" [1]

ABSTRACT

This paper introduces SCOPE (Structured Computations Optimized for Parallel Execution), a language designed for internal use at Microsoft targeted for massive data analysis. Companies providing cloud-scale services such as Microsoft, AWS, Adobe, and VM Ware have an increasing need to store and analyze massive data sets. Such an analysis is needed to improve quality of service and support novel features, to detect fraudulent activity or changes in usage pattern. In order to do so, a programming model independent of the underlining system is needed which allows users to extend functionality to meet a variety of requirements.
SCOPE is an easy to use language with no explicit parallelism while being amenable to efficient parallel execution on large clusters.

1. Introduction

The size of the data set analyzed by Internet companies is massive. Due to the size, traditional parallel database solutions can be expensive. In order to perform this analysis in a cost-effective manner, companies have developed distributed data storage and processing systems like: Google's File System [4], Bigtable [3], Map- Reduce [5], Hadoop [1], Yahoo!'s Pig system [2] etc. It is challenging to design a programming model that allows efficient utilization of all resources of a cluster.
The paper states that even Google's Map-Reduce programming model is not simple to implement since the users are forced to map their applications to the map-reduce model in order to achieve parallelism. However, SCOPE, a declarative scripting language developed at Microsoft, allows users to specify the data transformations required to solve the problem via SQL like scripts. Data is modeled as rows composed of typed columns. SCOPE's support to user defined version of operators: extractors, processors, combiners, reducers allows better problem depiction than that with traditional SQL.

Figure 1 shows a sample SCOPE script which finds from the search log the popular queries that have been requested at least 1000 times.

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

OUTPUT TO "qcount.result";
```

Figure 1: A sample SCOPE script for counting popular queries

The select command uses a Log Extractor, to parse each log record and extract the requested columns. By default, output of a previous command is used as input for the next command. The result is written back to the file "qcount.result".

2. PLATFORM OVERVIEW

Cosmos is Microsoft's developed distributed computing platform, for storing and analyzing massive data sets. It runs on large clusters consisting of thousands of commodity servers with distributed disk storage. Some high-level objectives of Cosmos include: reliability, availability, scalability, high performance, low cost.

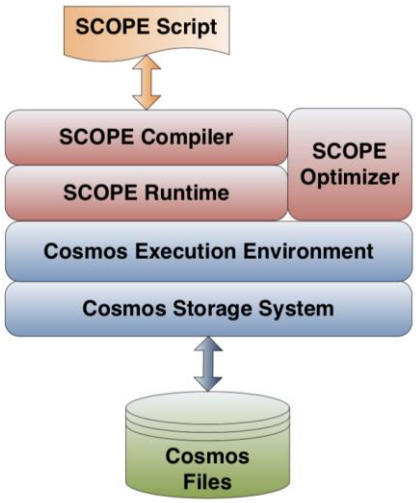

Figure 2: Cosmos Software Layers

Figure 2 shows the main components of the Cosmos platform, which mainly includes:

2.1. Cosmos Storage

Cosmos storage is an append only file system which stores petabytes of data. The data on the storage is replicated for fault tolerance and compressed to save storage. The Cosmos file is divided into sequence of **extents** (refer figure 3) which are a unit of space allocation and can be few 100 Megabytes in size.

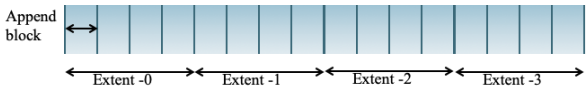

Figure 3: Cosmos file is physically divided into units of extents.

2.2. Cosmos Execution Environment

Cosmos execution environment is responsible for deploying, executing, and debugging distributed applications. The runtime component responsible of execution is called as Job Manager. The jobs to the Job Manager are submitted via a Directed Acyclic Graph

(DAG) where the vertices represent the processes and the edge represents data flow between the processes. Job Manager is responsible for scheduling and coordinating the vertex execution along with resource management and fault tolerance.

2.3. SCOPE

It is a high-level scripting language built on top of Cosmos for writing data analysis jobs. The SCOPE scripting language resembles SQL and supports a variety of data types, including int, long, double, float, Date Time, string, and bool.
The SCOPE execution environment consists of a SCOPE compiler and SCOPE optimizer which translates scripts to efficient parallel execution plans.
The ***SCOPE compiler*** parses the script, and performs metadata operations (syntax check, and name resolution). The result of the compilation is an *internal parse tree*.
The ***SCOPE optimize***r is a transformation-based optimizer. The internal parse tree from the compiler is parsed bottom-up and a physical execution plan is created. In order to obtain an efficient execution plan, traditional optimization rules from database systems such as removing unnecessary columns, pushing down selection predicates, and pre-aggregating the results are applied.

SCOPE'S resemblance to SQL reduces the learning curve for users and allows easy porting of existing SQL scripts into SCOPE script. The most common operators for a SCOPE script include Input & Output, Select & Join and some user defined operators (Produce, Reduce and Combine). The following sections describes them in detail.

3. Input and Output

The Scope script provides two customizable commands for reading data from a data source and writing data to a data sink. These are EXTRACT and OUTPUT.

The extract command extracts data from a Cosmos file and outputs sequence of rows in the format specified in the extract clause. Custom Extractors can be specified by extending the C# class 'Extractor'. Users can provide custom schema information by overwriting the function "Produce", which is called at compile time. The output command is used to write data to a Cosmos file, a regular file, or any other data sink.

4. Select and Join

SCOPE supports select, join statements, and allows nesting in from clause similar to SQL. It also supports different aggregate functions like: COUNT, COUNTIF, MIN, MAX, SUM, AVG, STDEV, VAR, FIRST, LAST. However, it does not allow subqueries.

The subquery in figure 4 can be executed in SCOPE as shown in figure 5

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

Figure 4: A simple subquery in SQL

```
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;
```

Figure 5: An equivalent subquery in SCOPE.

5. User Defined Operators

SCOPE provides three highly extensible commands that manipulate rowsets: PROCESS, REDUCE and COMBINE.
The extensible commands provide the same functionality as the map-reduce model described in [5] and the operations map, reduce, and merge described in [6].

5.1. Process

The process command takes a rowset as input, processes each row in turn, and outputs a sequence of rows. The main work of a process command is done by a user-written processor figure 6 shows an example of the process command to extract BIGRAMS from the sample string. The general syntax of the process command includes a clause USING which specifies the user defined process. figure 7

Figure 6: An example showing possible BIGRAMS for 'ABC'

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

Figure 7: A General Process Command

The process command is a very flexible command that enables users to implement processing that is difficult or impossible to express in SQL. A Custom Processor can be implemented by overriding the Process method and specifying the output Schema via the Produce method as shown in figure 8.

```
public class MyProcessor: Processor {

public override Schema Produce ( string[ ] requestedColumns ,string [ ] args, Schema
inputSchema)

{ ..... }

public override IEnumerable<Row> Process (RowSet input,Row outRow, string[ ] args)

{ ..... }

}
```

Figure 8: A sample custom Processor

5.2. Reduce

The reduce command takes in input a grouped row set based on the grouping attribute specified in the ON clause and produces zero or multiple rows per group. Figure 9 shows a sample reduce command with this grouping clause.

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

Figure 9: A sample Reduce command

The actual work of the reduce function is done via a custom reducer by overriding the reduce method. The example in figure 10 shows a reducer for counting the rows.

5.3. Combine

COMBINE is a binary operator that takes two input row sets, combines them in some way, and outputs a sequence of rows. Inputs are grouped and only rows from matching groups can be combined. This leads to partitioning and distributed processing of the inputs. Figure 11 shows a custom combiner 'Multiset Difference' which calculates the difference between two multisets.

```
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[0].String);
     count++;
    }

 outputRow[1].Set(count.toString());
 yield return outputRow
 }
}
```

Figure 10: A custom Reducer for counting the number of rows

```
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 leftSchem
string leftTable, Schema rightSchema, string rightTable) {

         return new Schema(requestedColumns) };
}
}
```

Figure 11: Example Implementation of a Custom Combiner (computes the difference of two multisets)

6. Example Query Plan

This section explains query execution plan used by SCOPE for the QCount query of figure1. The input file is divided into extents which 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 4.

1. **Extract**: As shown in the figure 12, the first stage involves running multiple extractors in parallel, each one reading part of the file.
2. **Partial aggregation**: In the second stage, data from extractors running on machines within the same rack is pre-aggregated to reduce data volume. In order to do so, knowledge about network topology of the cluster is required. Partial aggregation is achieved by sorting or hashing and can be applied multiple times on groups of extents.
3. **Distribute**: The result from Partial Aggregation stage is partitioned on the grouping column "query". This leads to mapping of all (partially aggregated) rows with the same query string into the same partition.
4. **Full aggregation & Filter**: Each partition calculates the final aggregation in parallel and the fully aggregated rows are filtered based of the count of 1000
5. **Sort& Merge**: The remaining rows are sorted by count and merged together on a single machine, producing the final result.
6. **Output**: The final result is written to a Cosmos file 'qcount.result'.

The execution plan is submitted to Cosmos execution environment which insures all necessary resources are available and schedules its execution. Job Manager monitors progress of all executing vertices and re-executes the failing vertices.

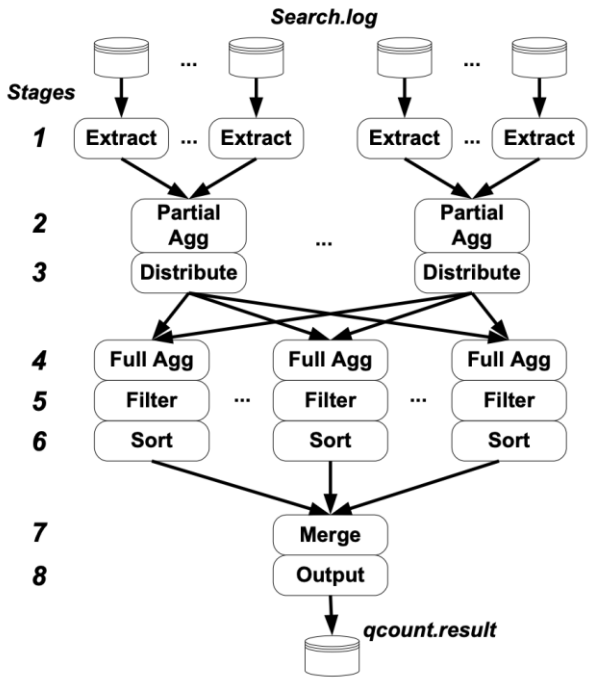

Figure 12: Execution Plan for QCount Query

7. EXPERIMENTAL EVALUATION

This section records the performance and scalability of complex OLAP queries executed using SCOPE on machines with varying cluster sizes. The experimental setup included-

1. Test cluster of 84 machines with two dual-core processors and four 500GB SATA disks.
2. All machines installed with Windows Server 2003 Enterprise X64 Edition SP1

The performance was recorded via TPC-H [7], a well-known data warehouse benchmark. In the experiment, three TPC-H databases with scale factors 10 (10GB), 100 (100GB), and 1000 (1TB) were generated and the raw database files were stored as Cosmos files in the cluster.

7.1. TPC-H-Queries

All of the 22 queries from TPC-H benchmark were executed using SCOPE. For some complex queries, SCOPE generated fairly sophisticated parallel execution plans. This paper shows one of the queries in TPC-H benchmark.

7.1.1. 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. The SCOPE script is listed in figure 13.

The final execution plan for this Cosmos script is similar to the one for QCount query in section 6. The plan applies partial aggregation as early as possible to reduce the data size. The 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.

```
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";
```

Figure 13: SCOPE script for TPC-H Query 1

7.1.2. TPC-H Query 2

The TPC-H Query 2 finds the appropriate suppliers to place an order to for a given part in a given region. The query contains multi-way joins, aggregation, and a subquery.

7.2. Scalability

In the first experiment, Q1 and Q2 queries were run against the 1TB TPC-H. The query elapsed times of Q1 and Q2 on a cluster of 20 machines was used as base lines and performance ratio (elapsed time / base line) for different cluster configurations was recorded.
As shown in figure 14 query performance for both queries scales linearly with cluster size. An increase in cluster size results in high availability of resources to run 1TB TPC-H data. As a result, there is a linear growth of the performance.

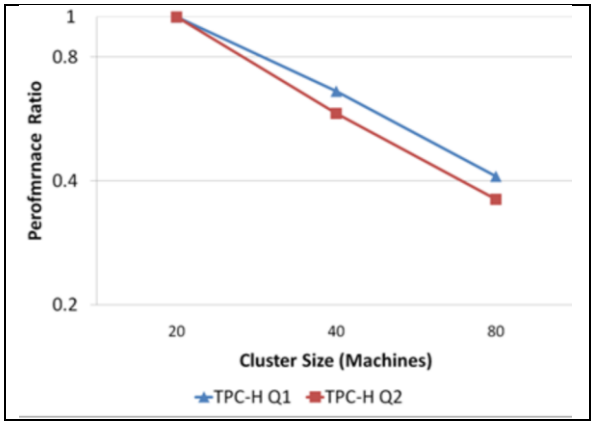

Figure 14: Scalability result recorded for experiment 1

In the second experiment, both queries were run on the full cluster but against databases of different sizes. The elapsed times of Q1 and Q2 against the 10GB TPC-H data was used as the base line. As shown in figure 15, query performance for both queries scales linearly with input size. It can be observed that database size of 10GB run against full cluster leads to underutilization of the cluster and performance drops. As the database size increases, the performance increases as the complete cluster is utilized.

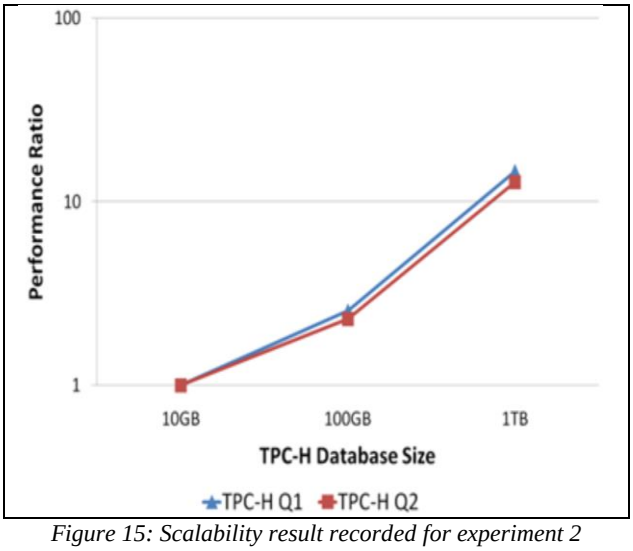

Figure 15: Scalability result recorded for experiment 2

8. Strengths and Limitations

The paper introduces strong concepts about Microsoft's scripting language, SCOPE, for efficient parallel processing of massive data sets.

In my opinion, the most interesting aspect of the paper is the similarity of SCOPE script with SQL and the ease with which parallel processing of data is achieved via user defined operators- Process, Reduce and Combine. The way in which query execution plans are created from SCOPE scripts (through sequence of aggregations and distributions) and sent to cosmos execution environment to be run as DAG job is also very interesting.

The paper claims that writing SCOPE scripts is simpler than programming map-reduce jobs and hence suggests that processing data via SCOPE is a better than with map-reduce. However, this conclusion is made only on the basis of 'ease of programming' and does not take into consideration, scalability and performance. Also, the paper does not discuss the possibility of applying multi-query optimizations parallelly which might result in faster execution plan creation.

9. Recent Developments

In 2016, Microsoft released Data Lake Store and Data Lake Analytics as Azure services. Both, Data Lake Store and Data Lake Analytics are derived from Cosmos which offers storage and analytics as single service. Data Lake offers storage and analytics as two separate services Data Lake is open to all and is not specific for only Microsoft's use.

10. Conclusion

This paper presents a summary of a scripting language called SCOPE, introduced in the original paper [1], which was developed for internal use at Microsoft. SCOPE was developed for web-scale data analytics for large clusters.
This scripting language has resemblance to SQL. It is a high-level declarative language with implementation details transparent to user. It also supports C# expressions and built-in .NET functions/library. The execution environment, Cosmos, for running the SCOPE script is available externally as Azure Data Analytics and Storage. Experiments confirm that query performance scales linearly with cluster and data sizes.

11. References

[1]. Ronnie Chaiken, SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets.

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

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

[6]. 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

[7] TPC-H: Decision Support Benchmark http://www.tpc.org/tpch/