SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets

Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. @ {Microsoft Corporation}

PVLDB, 2008

Presented by: Güneş Aluç
Problem

(a) accumulation of massive data sets → search logs, web content collected by crawlers, ad-click streams, etc.

necessitates the development of cost-efficient distributed storage solutions: GFS, BigTable, ... (i.e. exploit large clusters of commodity hardware)

(b) business value in analyzing massive data sets → better ad-placement, improved service (e.g. web search), data-mining opportunities, fraudulent activity detection, etc.

necessitates the development of distributed computing frameworks: MapReduce, Hadoop, ...

(c) the need to describe and execute ad-hoc large-scale data analysis tasks → in-house experiments

necessitates the development of high-level distributed dataflow languages: PigLatin, Dryad, SCOPE
Focus

- a **declarative** and **extensible** scripting language: SCOPE → “(S)tructured (C)omputations (O)ptimized for (P)arallel (E)xecution”
  - **Declarative**: users describe *large-scale data analysis tasks* as a flow of *data transformations*, w/o worrying about how they are parallelized on the underlying platform
  - **Extensible**: user-defined functions and operators
  - **Structured Computations**: data transformations consume and produce “rowsets” that conform to a *schema*
  - **Optimized for Parallel Execution**: ??? plan optimization not explicitly discussed in this paper
Yet Another High-Level Language for Large-Scale Data Analysis?

- A hybrid scripting language supporting not only user-defined map-reduce-merge operations, but also SQL-flavored constructs to define large-scale data analysis tasks

- How about PigLatin?
  - Somewhere in between SQL and MapReduce
    
    ```
    good_urls = FILTER urls BY pagerank > 0.2;
    groups = GROUP good_urls BY category;
    big_groups = FILTER groups BY COUNT(good_urls)>10^6;
    output = FOREACH big_groups GENERATE
               category, AVG(good_urls.pagerank);
    ```
  - Has support for a nested data model
    
    ```
    t = (‘alice’,
         {‘lakers’, 1},
         {‘iPod’, 2},
         [‘age’ – 20])
    ```
Overview

COSMOS execution environment

SCOPE Scripts

Dataflow

- Extract
- Process
- Reduce
- Combine

User-defined functions

User-defined operators

Built-in/custom extractors

Built-in/custom outputters

Input schemas:
- \( \text{IN}_1 \)
- \( \cdot \cdot \cdot \)
- \( \text{IN}_K \)

Output schemas:
- \( \text{OUT}_1 \)

{int/long/double/float/datetime/string/bool/...}

Data sources:
- Regular files
- External storage

Data sinks:
- Regular files
- External storage
Background on Cosmos

- **Cosmos Storage System**: a distributed storage platform, sharing ~ to GFS:
  - high availability, reliability, scalability and performance
  - compression/decompression
  - only supports append-style updates

- **Cosmos Execution Environment**: provides a high-level programming interface to execute parallel programs expressed as dataflow graphs, ~ to MapReduce:
  - parallelism, fault tolerance, data partitioning and resource management
At its core, SCOPE provides **SQL-flavored constructs** to describe large-scale data analysis tasks.

The language can be extended with **user defined functions** and **operators** (i.e. expressed in C#)

- Why? Literally speaking: “Its resemblance to SQL reduces the learning curve for users.”
- Why? Personal opinion: easier to translate extensible SCOPE scripts into Dryad Runtime DAGs
SCOPE Scripting Language

- SCOPE scripts consist of a sequence of commands. Sometimes, it is possible to break a single SCOPE command into a series of smaller commands which are tied together by named inputs (i.e. placeholders or variables):

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

```

```sql
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 dataflow is essentially made up of a sequence of commands each of which consumes a set of rowsets and produces a single rowset as output.
**Input & Output**

1. `EXTRACT column[:<type>] [, ...]`
2. `FROM <input_stream(s)>`
3. `USING <Extractor> [(args)]`
4. `[HAVING <predicate>]`

(1) schema of the rowset to be produced
(2) i.e. Cosmos files, regular files, external storage
(3) built-in or custom extractor
(4) [optional] filter
**Input & Output**

1. OUTPUT [<input>]
2. [PRESORT column]
   
   [ASC | DESC] [, …]]
3. TO <output_stream>
4. [USING <Outputter> [(args)]]

(1) rowset to export
(2) [optional] provide sort-order by columns
(3) i.e. Cosmos files, regular files, external storage
(4) built-in or custom outputter
Select & Join

SELECT
    [DISTINCT]
    [TOP count]
    select_expression [AS <name>] [, ...]

FROM
    <input stream(s)> USING <extractor> |
    <input> INNER JOIN <input> [ON <equijoin>] [, ...]
    LEFT OUTER
    RIGHT OUTER
    FULL OUTER

Aggregation functions: COUNT, COUNTIF, MIN, MAX, SUM, AVG, STDEV, VAR, FIRST, LAST

i.e. to rename columns in the intermediate rowset(s) produced

either an intermediate rowset is used or it is EXTRACTED from data source(s)

Multiple joins are allowed; equijoins have higher priority.

Employees

<table>
<thead>
<tr>
<th>Emp. ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
</tr>
</tbody>
</table>

Departments

<table>
<thead>
<tr>
<th>Emp. ID</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sales</td>
</tr>
</tbody>
</table>

inner join?
left outer join?
etc.
Select & Join

[ WHERE <predicate> ]

[ GROUP BY <grouping_columns> [, ...] ]

[ HAVING <predicate> ]

[ ORDER BY <select_list_item> [ASC | DESC] [, ...] ]

Subqueries are not allowed! However, the same functionality can be achieved by using OUTER joins within a sequence of commands.
Subqueries as Outer Joins

Let's work the following example out:

<table>
<thead>
<tr>
<th>Ra</th>
<th>Rb</th>
<th>Rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>50</td>
<td>Alice</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>Bob</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
<td>Clarice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sa</th>
<th>Sc</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Bob</td>
</tr>
<tr>
<td>15</td>
<td>Bob</td>
</tr>
<tr>
<td>25</td>
<td>Clarice</td>
</tr>
</tbody>
</table>

Question: Do both approaches share the same cost?
Expressions and Functions

• Where are these scalar expressions and functions used?
  - SELECT … select_expression …
  - [ WHERE <predicate> ]
  - [ HAVING <predicate> ]

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

- We have studied:
  - the syntax of SCOPE
  - how its SQL-flavored constructs can be extended with user defined expressions and functions

- However, we have not yet really discussed:
  (i) how the tasks described by SCOPE scripts would benefit from parallelization
  (ii) if we were to omit user defined expressions and functions, what value-added features would this system bring on top of traditional parallel database solutions

Well, intuitively, any aggregation operation (e.g. COUNT, SUM, AVG, etc. will benefit from parallelization. In this regard, we may assume that SCOPE compiler & optimizer will take care of it.
**Process – Reduce – Combine**

- Analogous to the map-reduce-merge model\(^1\)
- Let's work with the example in [1], Section 3.1

---

Process – **Reduce** – Combine

<table>
<thead>
<tr>
<th>emp-id</th>
<th>dept-id</th>
<th>Bonus ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>150</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>100</td>
</tr>
</tbody>
</table>

**REDUCE**

REDUCE [<input>
PRESORT column [ASC|DESC] [...]]
on grouping_column [...] using <Reducer> [(args)]
[PRODUCE column [...] [HAVING <predicate>]]

For each (grouped) rowset produce 0, 1, or multiple rows

<table>
<thead>
<tr>
<th>emp-id</th>
<th>dept-id</th>
<th>Bonus Sum ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>150</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>250</td>
</tr>
</tbody>
</table>

Venue for parallelization
Process – Reduce – Combine

- **presort dept-id, emp-id**
  - emp-id | dept-id | bonus-sum
  - 2      | A       | 0
  - 3      | A       | 250
  - 1      | B       | 150

- **presort dept-id**
  - dept-id | bonus adjustment
  - A       | 0.95
  - B       | 1.15

**COMBINE**

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

- **rowset**
  - emp-id | bonus
  - 2      | 0
  - 3      | 237.5
  - 1      | 172.5

**venue for parallelization:**

(!) requirement: inputs are grouped, and only rows from matching groups can be combined
So far...

Extract ...
Select ...
Process ...
Reduce ...
Combine ...
Select ...
Output ...

Extract ...
Process ...
Reduce ...
Combine ...
Combine ...
Reduce ...
Process ...

**SCOPE Compilation & Optimization**

- **Compilation**: SCOPE script $\rightarrow$ internal parse tree $\rightarrow$ [optimization] $\rightarrow$ Dryad DAG execution plan

- “SCOPE compiler combines adjacent vertices with physical operators that can be easily pipelined into (super) vertices (1:1)”

- **Optimization (traditional sense):**
  - remove unnecessary columns
  - pushing down selection predicates
  - Pre-aggregation, etc.

- **Optimization (distributed):**
  - when and what to partition
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 interesting part is that, even though the script does not explicitly contain any PROCESS-REDUCE-MERGE commands, a distributed plan is produced.
SCOPE Compilation & Optimization

- **Run-time optimization:**
  - make optimization decisions based on network topology (~static)
    - racks of commodity machines
    - per-rack switch
    - common switch
      - reduce workload on common switch
  - why not as well optimize based on how the data is actually stored in the Cosmos Storage System?
Experimental Results

- TPC-H / $Q_1$, $Q_2$

These experiments are somewhat amateur:
- 2 of the 22 queries in TPC-H
- only 3 clusters
- only 3 different sized dBs
- performance experiment → not linear in log scale
- how about performance of Map-Reduce-Merge tasks?
Discussion

- (Q1) How exactly is SCOPE different from Pig-Latin?

- (Q2) In SCOPE, queries can be written in a single SQL-block or in a pipelined sequence of commands. What advantages does each have? If we were to combine SQL-flavored commands with map-reduce-merge jobs, don't we essentially have to stick with the second option? Would it make a difference in terms of optimization?

- (Q3) SCOPE is built on top of the Dryad framework in which execution plans are expressed as directed-acyclic graphs. Is this restrictive? Can we actually benefit from cycles?

- (Q4) Plan optimization is quite vaguely discussed in this paper. To the best of our knowledge, the authors have not yet published a follow-up work, either. Do you believe this area is subject to improvement? Do you think optimization strategies that exploit the run-time distribution of data need to be developed?

- (Q5) What difficulties do we face in trying to compile user-defined functions into parallel execution plans? i.e. we know the semantics of SUM, COUNT, etc., but this is not true for user defined functions.
Thank you...

Any questions?
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.CopyTo(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
Additional Slides

// Join region, nation, and, supplier
// (Retain only the key of supplier)
\[RNS\_JOIN = \]
\[\begin{align*}
  & \text{SELECT } s\_suppkey, n\_name \\
  & \text{FROM region, nation, supplier} \\
  & \text{WHERE } r\_regionkey == n\_regionkey \\
  & \quad \text{AND } n\_nationkey == s\_nationkey; \\
\end{align*}\]

// Now join in part and partsupp
\[RNSPS\_JOIN = \]
\[\begin{align*}
  & \text{SELECT } p\_partkey, ps\_supplycost, \\
  & \quad ps\_suppkey, p\_mfr, n\_name \\
  & \text{FROM part, partsupp, rns_join} \\
  & \text{WHERE } p\_partkey == ps\_partkey \\
  & \quad \text{AND } s\_suppkey == ps\_suppkey; \\
\end{align*}\]

// Finish subquery so we get the min costs
\[SUBQ = \]
\[\begin{align*}
  & \text{SELECT } p\_partkey \text{ AS subq}\_partkey, \\
  & \quad \text{MIN(ps\_supplycost) AS min}\_cost \\
  & \text{FROM rnsps\_join} \\
  & \text{GROUP BY p\_partkey; } \\
\end{align*}\]

// Finish computation of main query
// (Join with subquery and join with supplier
// again to get the required output columns)
\[RESULT = \]
\[\begin{align*}
  & \text{SELECT } s\_acctbal, s\_name, p\_partkey, \\
  & \quad p\_mfr, s\_address, s\_phone, s\_comment \\
  & \text{FROM rnsps_join AS l0, subq AS sq, supplier AS s} \\
  & \text{WHERE l0.p\_partkey == sq.subq}\_partkey \\
  & \quad \text{AND l0.ps\_supplycost == min}\_cost \\
  & \quad \text{AND l0.ps\_suppkey == s.s\_suppkey} \\
  & \text{ORDER BY acctbal DESC, n\_name, s\_name, partkey; } \\
\end{align*}\]
Additional Slides

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