SCOPE: Easy and Efficient Parallel Processing of Massive Datasets

Appeared in VLDB 2008
Spring'10, CPS 296.1
Vamsidhar Thummala

Slides adapted from author's VLDB presentation

Distributed Computing Paradigms

<table>
<thead>
<tr>
<th></th>
<th>Google</th>
<th>Yahoo!</th>
<th>Microsoft</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>GFS/BigTable</td>
<td>HDFS</td>
<td>Cosmos</td>
</tr>
<tr>
<td>(Files: Chunk)</td>
<td></td>
<td></td>
<td>(Files: Extent)</td>
</tr>
<tr>
<td>Computation</td>
<td>MR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interface</td>
<td>Sawzall/MR</td>
<td>PigLatin</td>
<td>SCOPE/DryadLINQ</td>
</tr>
</tbody>
</table>

SCOPE Introduction (1/2)

• Used in Live Search (informal)
  – Web data analysis, user log analysis, relevance studies
• Infrastructure
  – Large shared nothing commodity hardware
• Programming goals similar to DryadLINQ
  – Sequential, single machine programming abstraction
  – SQL emphasis
    • MR is too rigid
    • Writing MR programs is like writing physical execution plans in DBMS

SCOPE Introduction (2/2)

• Structured Computations Optimized for Parallel Execution
  – A declarative scripting language
  – Easy to use: SQL-like syntax plus MapReduce-like extensions
  – Modular: provides a rich class of runtime operators
  – Highly extensible:
    • Fully integrated with .NET framework
    • Provides interfaces for customized operations
  – Flexible programming style: nested expressions or a series of simple transformations

Architecture

• Cosmos Storage system
  – Append-only distributed file system for storing petabytes of data
  – Optimized for sequential I/O
  – Data is compressed and replicated
• Cosmos Execution Environment
  – Dryad

SCOPE – An example

• Compute the popular queries that has been requested at least 1000 times

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Scenario 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT count(*) FROM sample_log TIMES LogReader GROUP BY query</td>
<td></td>
</tr>
<tr>
<td>= SELECT query, count(*) FROM a GROUP BY query</td>
<td></td>
</tr>
<tr>
<td>= SELECT query, count(*) FROM a LOGREADER GROUP BY DESC</td>
<td></td>
</tr>
<tr>
<td>OUTPUT TO &quot;popular.count&quot;</td>
<td></td>
</tr>
<tr>
<td>= SELECT query, count(*) FROM a LOGREADER GROUP BY DESC</td>
<td></td>
</tr>
<tr>
<td>OUTPUT TO &quot;popular.count&quot;</td>
<td></td>
</tr>
</tbody>
</table>
Data Model, Input, Output

- Data model
  - Relation row set with typed columns

- Input, Output
  - Relational, non-relational sources

  - EXTRACT, OUTPUT commands are provided

    | EXTRACT column{type} [ ... ] | OUTPUT {input} |
    | FROM <input stream(s)> > | TO <output stream(s)> |
    | USING <Extractor> [ ... ] | [USING <Outputter> [ ... ]]
    | [HAVING <predicate>] | |

  - USING clause allows customization (C#)

Select and Join

- 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 <sort_list> [ASC | DESC] [ ... ]

- joined input: - <join_type> - JOIN <inputs> [ON <equijoins>]

- Supports basic aggregation functions
- No subqueries

No subqueries - Example

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

- Equivalent query in SCOPE

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

Deep integration with C#

- SCOPE supports C# expressions and built-in .NET functions/library
- User-defined scalar expressions
- User-defined aggregation functions

```csharp
cs
R = SELECT A.C AS ar, B.Time() AS Rt
FROM E
WHERE StringCompare(C, "xyz") > 2;

using System;

namespace ScopeExample
{
    public class Example
    {
        public static void Main(string[] args)
        {
            string input; // input from user
            output = string.Empty; // output from program

            // Code block for processing data

            // Yield in C#
        }
    }
}
```

User Defined Operators

- SCOPE supports three extensible commands: PROCESS, REDUCE, COMBINE
- Complements SELECT for complicated analysis
- Easy to customize by extending built-in C# components
- Easy to reuse code in other SCOPE scripts
- Any resemblance with already seen operators?
  - Apply, Fork (Dryad)
  - FILTER, FLATTEN, COGROUP (PigLatin)

PROCESS

- PROCESS command takes a rowset as input, processes each row, and outputs a sequence of rows

```csharp
PROCESS [input] |
USING <Processors> [ ... ] |
[where <predicate> = ... ] |
[HAVING <predicate> = ... ] |
```
REDUCE

• REDUCE command takes a grouped rowset, processes each group, and outputs zero, one, or multiple rows per group.

• Why do we need REDUCE when you have GROUP BY?

COMBINE

• COMBINE command takes two matching input rowsets, combines them in some way, and outputs a sequence of rows.

• Example: MultiSetDifference

Importing Scripts

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

Enables modularity and information hiding

SCOPE Execution

• SCOPE Compiler
  - Generates query plan using default plan for each command
  - Combines adjacent operators into a single vertex when possible

• SCOPE Optimizer
  - Based on Cascades framework
  - Cost-based
  - Not completely implemented
  - Some tricks
  - Not enough details in paper

• SCOPE Runtime (Probably, Dryad)
  - Composable physical operators
  - Operators are implemented in iterator model
  - Executes series of operators in pipelined fashion

Example Query Plan

1. Extract the input cosine file
2. Partially aggregate at the rank level
3. Partition on “query”
4. Folly aggregate
5. Apply filter on “count”
6. Sort results in parallel
7. Merge results
8. Output as a cosine file

SCOPE vs. Other languages

<table>
<thead>
<tr>
<th>Language</th>
<th>SQL-like</th>
<th>Embedded SQL</th>
<th>Scripting/Some resemblance to SQL</th>
<th>Rule-based (basic)</th>
<th>Rule-based (basis)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Default Plan</td>
<td>DAG</td>
<td>Default Plan</td>
<td>Default Plan</td>
<td></td>
</tr>
<tr>
<td>Optimizer</td>
<td>No details</td>
<td>?</td>
<td>Rule-based</td>
<td>Rule-based</td>
<td></td>
</tr>
<tr>
<td>Runtime</td>
<td>Cosmos (Pipelined)</td>
<td>Dryad (EPG)</td>
<td>MapReduce (Pipelined)</td>
<td>MapReduce (Pipelined)</td>
<td></td>
</tr>
</tbody>
</table>
Discussion

• Performance
  – Not enough details
• Operators can be improved
  – Extractors, Combiners
• SQL flavor language?
  – DryadLINQ vs. SCOPE