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

	Google	Yahoo!	Microsoft
Storage	GFS/BigTable (Files: Chunk)	HDFS (Files: Chunk)	Cosmos (Files: Extent)
Computation	MR		Cosmos/Dryad
Interface	Sawzall/MR	PigLatin	SCOPE/DryadLINQ

The distinction is not entirely clear

SCOPE Introduction (1/2)

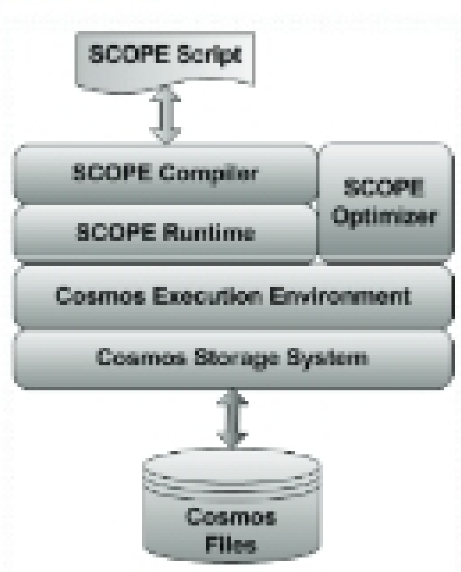
- Used in Live Search (informal)
  - Web data analysis, user log analysis, relevance studies
- Infrastructure
  - Large shared nothing commodity hardware
- Programming goals similar to DyradLINQ
  - Sequential, single machine programming abstraction
  - SQL emphasis
    - MR is too rigid
    - Writing 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 MapRecuce-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

Scenario 1:

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

OUTPUT TO "qcount.result"
```

Scenario 2:

```
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"
```

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>] [, ...]
FROM < input_stream(s) >
USING <Extractor> [(args)]
[HAVING <predicate>]
```

```
OUTPUT [<input>]
TO <output_stream>
[USING <Outputter> [(args)]]
```

  - 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 <select_list_item> [ASC | DESC] [, ...]]

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

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

- Supports basic aggregation functions
- No subqueries

No subqueries - Example

```
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

```
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;
```

Deep integration with C#

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

```
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)
{ ... }
#ENDCS
```

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

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

```
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)
   { ... }
}
```

- Yield in C#

REDUCE

- REDUCE command takes a grouped rowset, processes each

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

```
public class MyReducer : Reducer
{
  public override Schema Produce(string[] requestedColumns, string[] args, Schema inputSchema)
  { … }

  public override IEnumerable<Row> Reduce(RowSet input, Row outRow, string[] args)
  { … }
}
```

- 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

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

```
COMBINE S1 WITH S2
ON S1.A==S2.A AND S1.B==S2.B AND S1.C==S2.C
USING MyCombiner
PRODUCE D, E, F
```

```
public class MyCombiner : Combiner
{
  public override Schema Produce(string[] requestedColumns, string[] args,
                                 Schema leftSchema, string leftTable, Schema rightSchema, string rightTable)
  { … }

  public override IEnumerable<Row> Combine(RowSet left, RowSet right, Row outputRow, string[] args)
  { … }
}
```

- Example: MultiSetDifference

Importing Scripts

IMPORT <script_file>

[PARAMS<par_name> = <value> [,…]]

```
Script Definition:
E = EXTRACT query
    FROM @@logfile@@
    USING LogExtractor ;

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

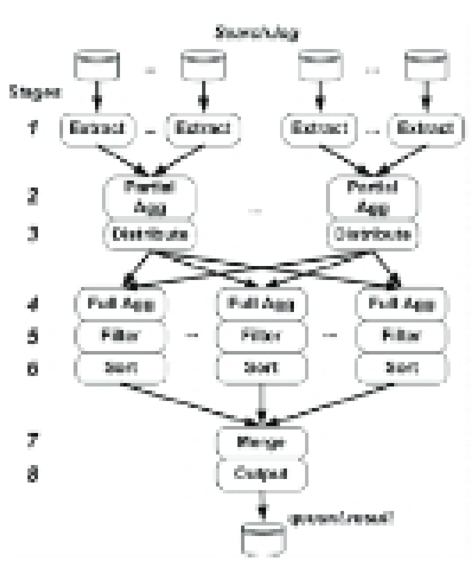
```
Query:
Q1 = IMPORT "MyView.script"
     PARAMS logfile="Queries_Jan.log",
            limit=1000;

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

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




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

OUTPUT TO "qcount.result"
```

1. Extract the input cosmos file
2. Partially aggregate at the rack level
3. Partition on "query"
4. Fully aggregate
5. Apply filter on "count"
6. Sort results in parallel
7. Merge results
8. Output as a cosmos file

SCOPE vs. Other languages

	SCOPE	DryadLINQ	PigLatin	Hive
Language	SQL-like	Embedded SQL	Scripting/Some resemblance to SQL	SQL-like
Compiler	Default Plan	DAG	Default Plan	Default Plan
Optimizer	No details ?	?	Rule-based (basic)	Rule-based (basic)
Runtime	Cosmos (Pipelined)	Dryad (EPG)	Hadoop (Pipelined)	Hadoop (Pipelined)