# *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?*

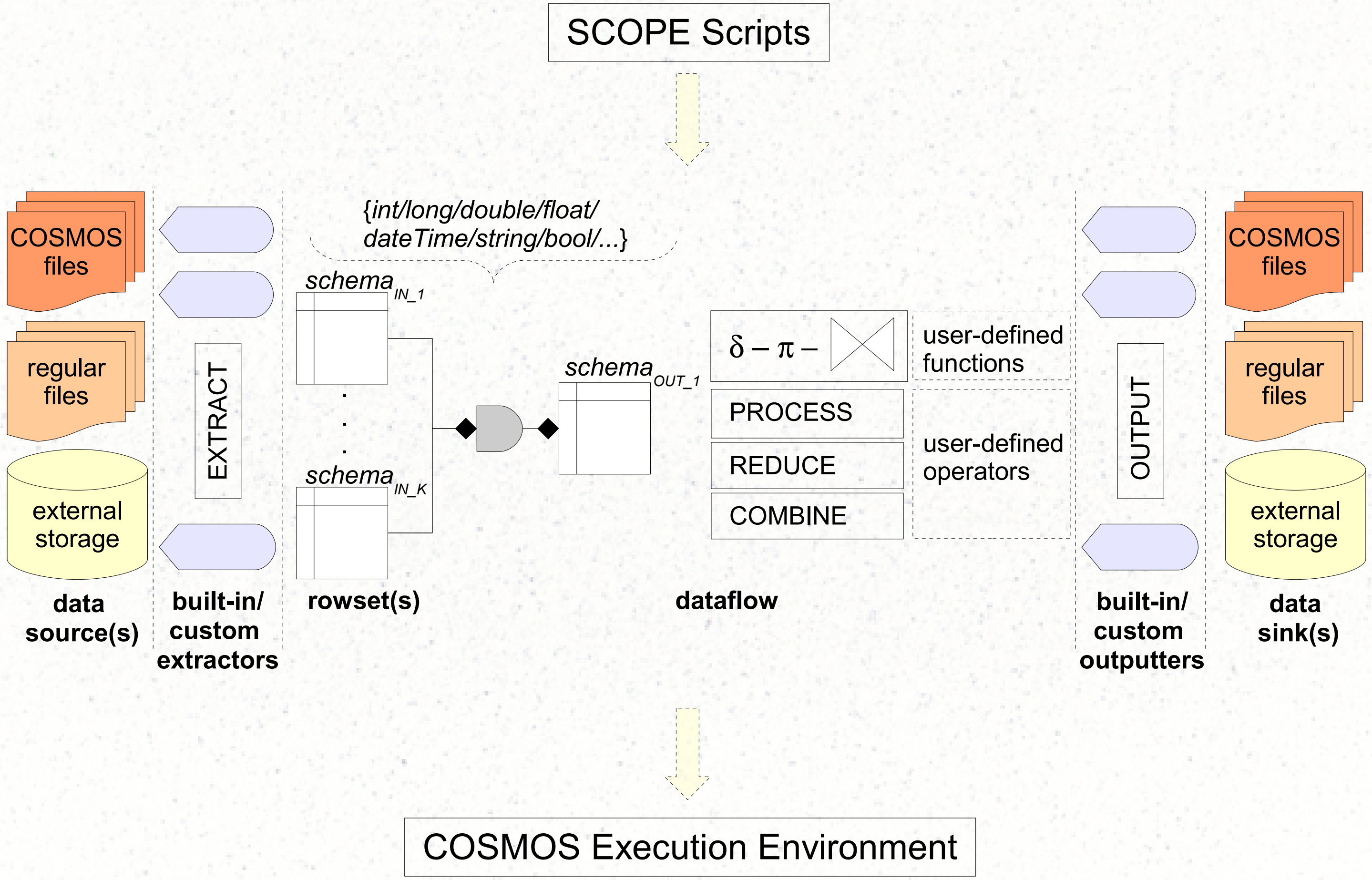
- A hybrid scripting language supporting not only user-defined map-reduce-merge operations, but also SQL-flavored constructs to define large-scale data analysis tasks
- How about PigLatin?
  - Somewhere in between SQL and MapReduce

```
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

$$t = \left(\texttt{`alice'}, \left\{ \begin{array}{c} (\texttt{`lakers'}, \texttt{1}) \\ (\texttt{`iPod'}, \texttt{2}) \end{array} \right\}, \left[\texttt{`age'} \rightarrow 20\right]\right)$$

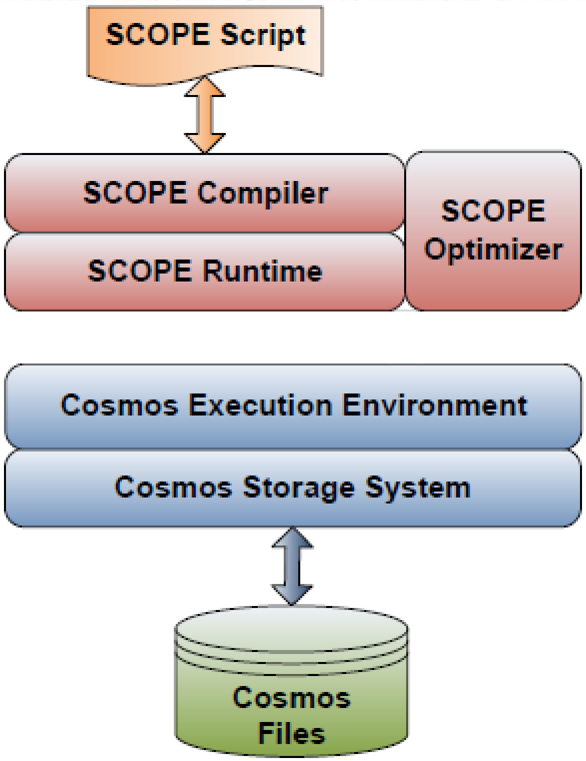
# Overview



SCOPE Scripts
COSMOS files
regular files
external storage
data source(s)
EXTRACT
built-in/ custom extractors
{int/long/double/float/ dateTime/string/bool/...}
schema IN_1
schema IN_K
rowset(s)
schema OUT_1
δ – π –
user-defined functions
PROCESS
REDUCE
COMBINE
user-defined operators
dataflow
OUTPUT
built-in/ custom outputters
COSMOS files
regular files
external storage
data sink(s)
COSMOS Execution Environment

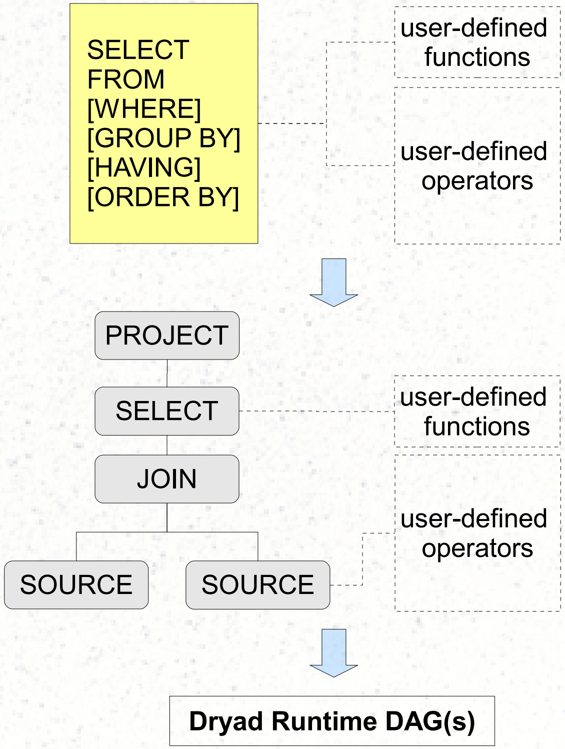
# *Background on Cosmos*

- 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



# SCOPE Scripting Language

- 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

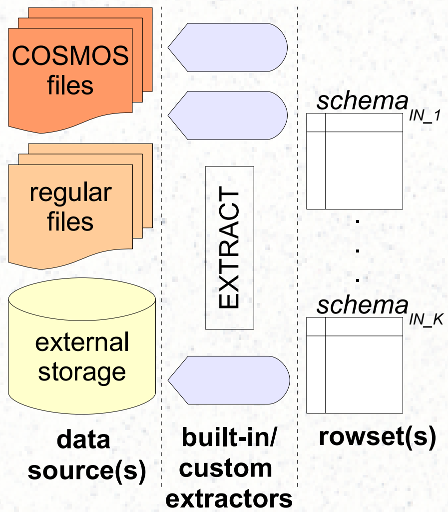
- SCOPE scripts consist of a sequence of *commands*. Sometimes, it is possible to break a single SCOPE *command* into a series of smaller *commands* which are tied together by <u>named inputs</u> (i.e. placeholders or variables):

```
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 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*

```
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";
```

# *Input & Output*




(1) EXTRACT column[:<type>] [, …]

(2) FROM < input_stream(s) >

(3) USING <Extractor> [(args)]

(4) [HAVING <predicate>]

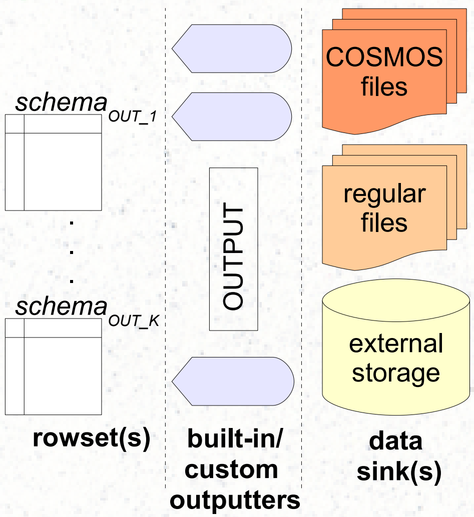
(1) schema of the rowset to be produced

(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>] [, ...]
```

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

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

```
FROM
        <input stream(s)> USING <extractor> |

        <input> INNER        JOIN <input> [ON <equijoin>] [, ...]
                LEFT OUTER
                RIGHT OUTER
                FULL OUTER
```

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

Multiple joins are allowed; equijoins have higher priority.

**Employees**

| Emp. ID | Name |
|---|---|
| 1 | Alice |
| 2 | Bob |

**Departments**

| Emp. ID | Department |
|---|---|
| 1 | Sales |

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

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

```
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;
```

- Let's work the following example out:

| $R_a$ | $R_b$ | $R_c$ |
|---|---|---|
| 6 | 50 | Alice |
| 3 | 75 | Bob |
| 2 | 110 | Clarice |

| $S_a$ | $S_c$ |
|---|---|
| 5 | Bob |
| 15 | Bob |
| 25 | Clarice |

**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> ]

```
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

  *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.*

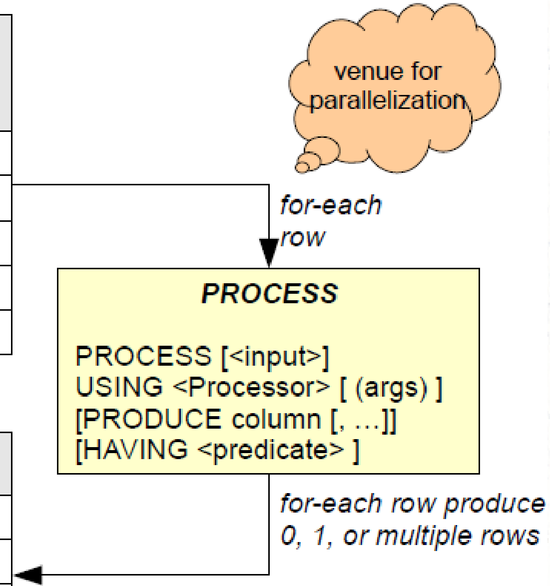
  (ii) if we were to omit user defined expressions and functions, what value-added features would this system bring on top of traditional parallel database solutions

# Process – Reduce – Combine

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

rowset

| emp-id | emp-info: dept-id | emp-info: bonus |
|---|---|---|
| 1 | B | innovation award ($100) |
| 1 | B | hard worker award ($50) |
| 2 | A | NULL ($0) |
| 3 | A | high-performer ($150) |
| 3 | A | innovation award ($100) |




rowset

| emp-id | dept-id | Bonus ($) |
|---|---|---|
| 1 | B | 100 |
| 1 | B | 50 |
| 2 | A | 0 |
| 3 | A | 150 |
| 3 | A | 100 |



(1) Yang, H., Dasdan, A., Hsiao, R., and Parker, D. S. 2007. Map-reduce-merge: simplified relational data processing on large clusters. In Proceedings of the 2007 ACM SIGMOD international Conference on Management of Data. SIGMOD '07.

# *Process – <u>Reduce</u> – Combine*

rowset

| emp-id | dept-id | Bonus ($) |
|---|---|---|
| 1 | B | 100 |
| 1 | B | 50 |
| 2 | A | 0 |
| 3 | A | 150 |
| 3 | A | 100 |

group
ON *emp-id*

venue for parallelization

***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*

rowset

| emp-id | dept-id | Bonus Sum ($) |
|---|---|---|
| 1 | B | 150 |
| 2 | A | 0 |
| 3 | A | 250 |

# Process – Reduce – Combine

presort dept-id, emp-id

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

rowset

presort dept-id

rowset

| 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> ]
```

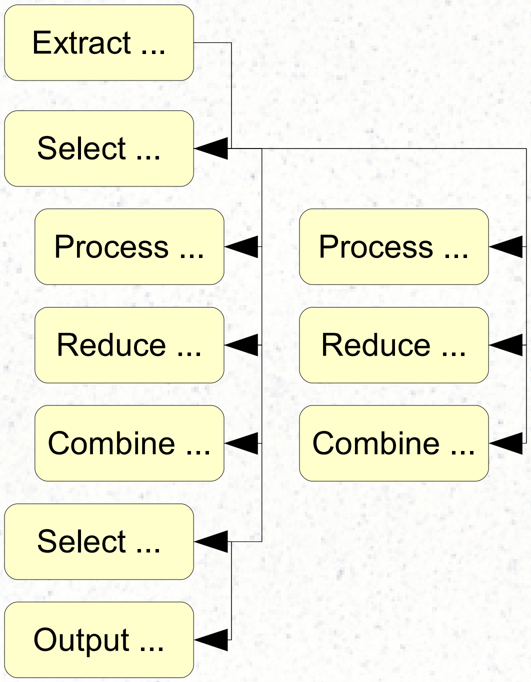
rowset

| 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

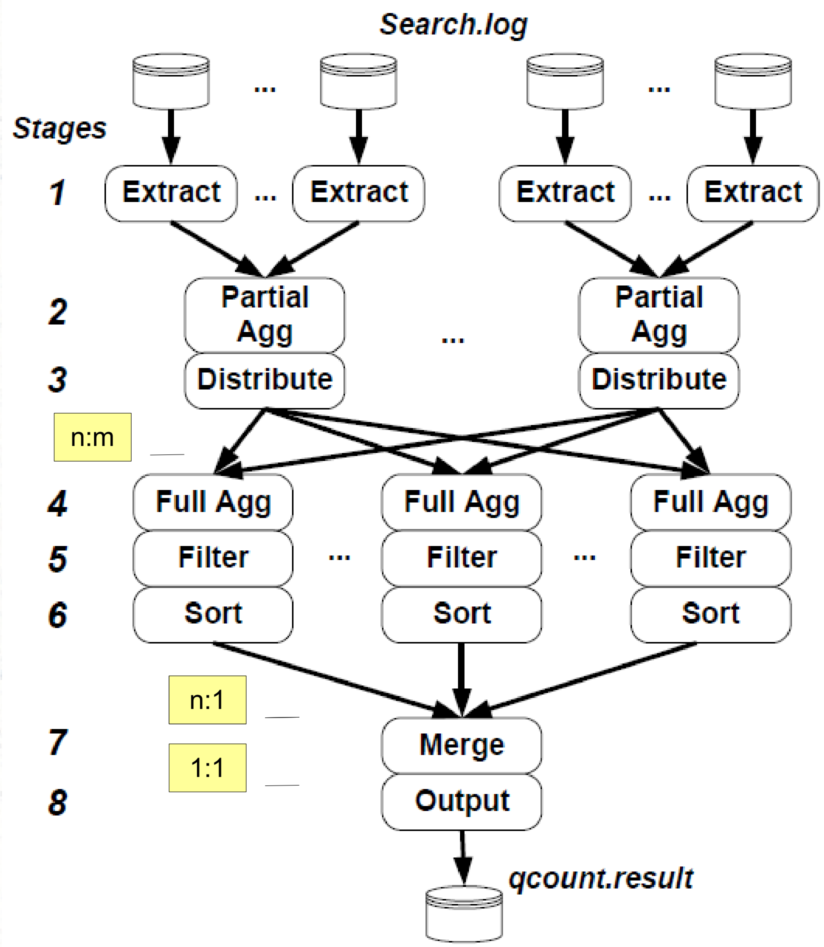
# So far...



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

# *SCOPE Compilation & Optimization*

- Compilation: SCOPE script → internal parse ***tree*** → [optimization] → 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



# SCOPE Compilation & Optimization

SELECT query, COUNT() AS count

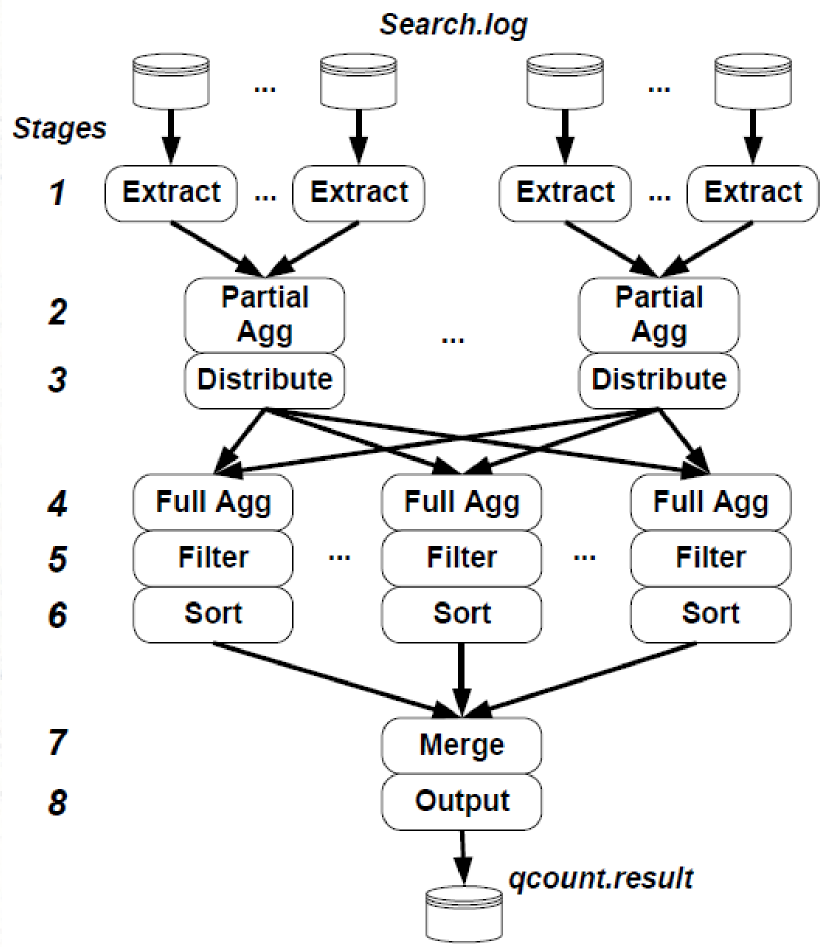
FROM "search.log" USING LogExtractor

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.*



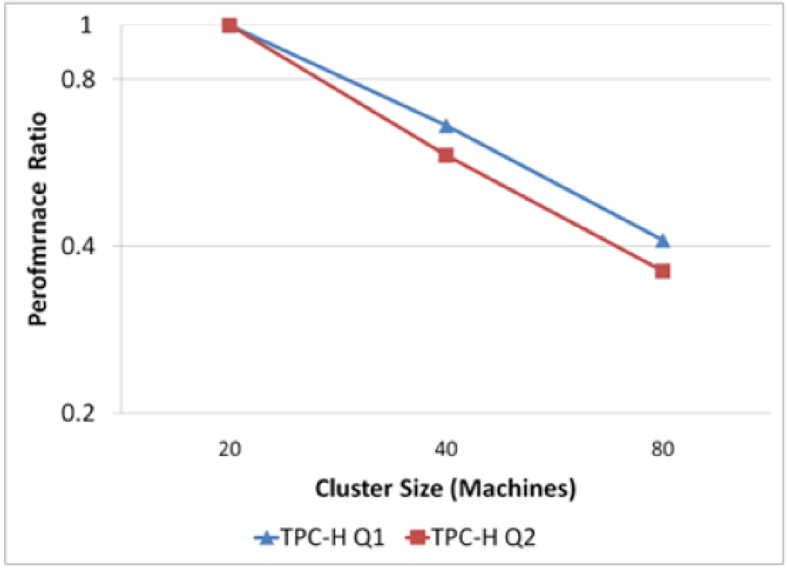
# *SCOPE Compilation & Optimization*

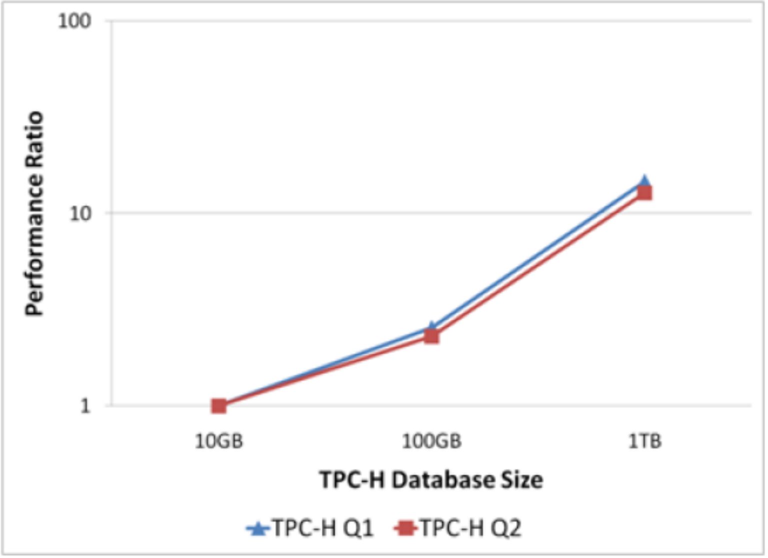
- <u>Run-time optimization</u>:
  - make optimization decisions based on network topology (~static)
    - racks of commodity machines
    - per-rack switch
    - common switch
      - reduce workload on common switch
  - why not as well optimize based on how the data is actually stored in the Cosmos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

# *Additional Slides*

```
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**

# Additional Slides

```
// 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;
```

# Additional Slides

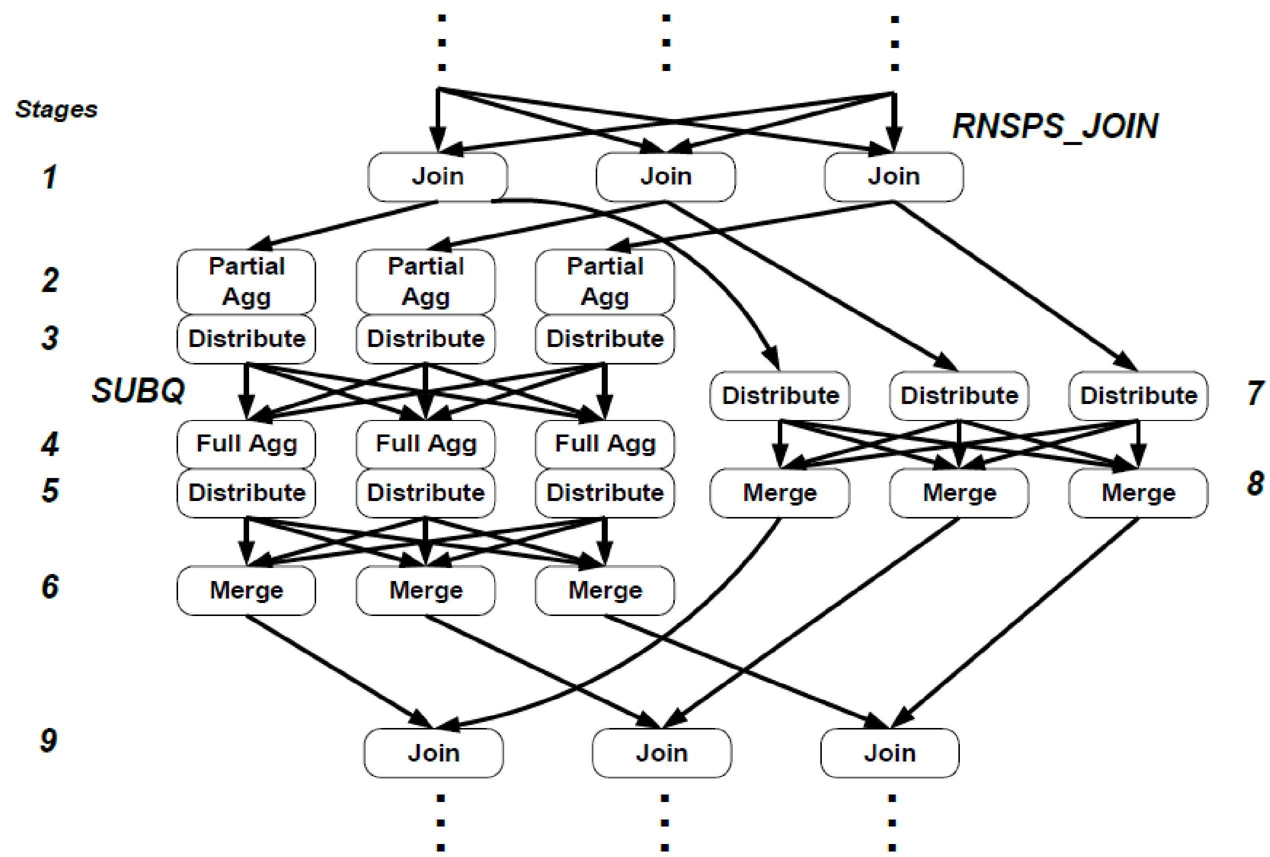



Figure 8: Sub Execution Plan for TPC-H Query 2