A Relational Approach to Complex Dataflows

Yannis Chronis, Yannis Foufoulas, Vaggelis Nikolopoulos, Alexandros Papadopoulos, Lefteris Stamatogiannakis, Christoforos Svingos, Yannis Ioannidis
{i.chronis, johnfouf, vgnikolop, alpap, estama, c.sviggos, yannis}@di.uoa.gr
MaDgIK Lab, Dept. of Informatics and Telecom., University of Athens, Greece.

ABSTRACT

Clouds have become an attractive platform for highly scalable processing of Big Data, especially due to the concept of elasticity, which characterizes them. Several languages and systems for cloud-based data processing have been proposed in the past, with the most popular among them being based on MapReduce [6]. In this paper, we present Exareme, a system for elastic large-scale data processing on the cloud that follows a more general paradigm. Exareme is an open source project [1][1]. The system offers a declarative language which is based on SQL with user-defined functions (UDFs) extended with parallelism primitives and an inverted syntax to easily express data pipelines. Exareme is designed to take advantage of clouds by dynamically allocating and deallocating compute resources, offering trade-offs between execution time and monetary cost.

1. INTRODUCTION

Modern applications face the need to process large amount of data using complex functions. Examples include complex analytics [13], similarity joins [11], and extract-transform-load (ETL) processes [14]. Such rich tasks are typically expressed using high-level APIs or languages [15] and are transformed into data intensive workflows, or simply dataflows. Exareme uses a master-worker architecture. Our language is based on SQL to express both intra-worker and inter-worker dataflows. We use UDFs and a inverted syntax to easily express local pipelines and complex computations. Inter-worker dataflows are described with simple parallelism primitives. These abstractions allow users to fine tune dataflows for different applications. All of the basic components of Exareme are designed to support the elastic properties of cloud infrastructures. We provide comparisons to other state of the art systems.

[1]The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreements 318338 project "Optique", 600932 project "MD PAEDIGREE", 604102 project "Human Brain".

2. SYSTEM OVERVIEW

The system architecture is shown in Figure 1. From a user's point of view, the system is used as a traditional database system: create / drop tables or indexes, import external data, issue queries. The queries are expressed in ExaDFL. ExaDFL is transformed into data processing flows (dataflows) represented as directed acyclic graphs (DAGs) that have arbitrary computations (operators) as nodes and producer-consumer interactions as edges between the nodes. The typical queries we target are complex data-intensive transformations that are expensive to execute, queries may run for several minutes or hours.

Exareme is separated into the following components: The **Master**, is the main entry point, through the gateway, to the system and is responsible for the coordination of the rest of the components. The **Execution Engine** communicates with the resource manager and schedules the operators of the query respecting their dependencies in the dataflow graph and the available resources. It also monitors the dataflow execution and handles failures.

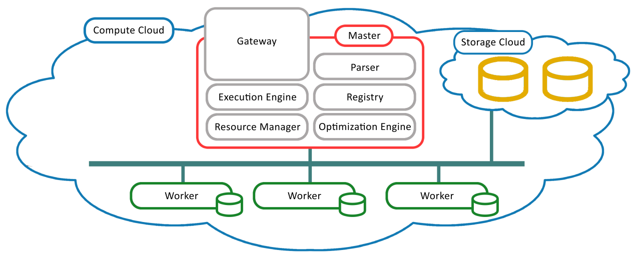

Figure 1: Exareme's architecture

All the information related to the data and the allocated VMs is stored in the **Registry**. The **Resource Manager** is responsible for the allocation and deallocation of VMs based on the demand. The **Optimization Engine** translates ExaDFL query into the distributed machine code of the system (similar to [12]) and creates the final execution plan by assigning operators to workers (Section 4.1). Finally, the **Worker** executes operators (relational operators and UDFs) and transfers intermediate results to other workers. Each worker fetches the partitions needed for the execution and caches them to its local disk for subsequent usage.
Madis is the core engine of the Worker[2], it is an extension of

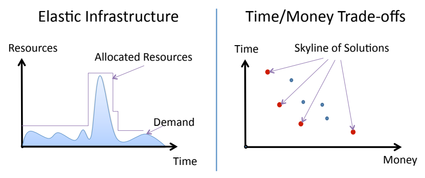

Figure 2: Dynamic infrastucture elasticities

SQLite, based on the APSW wrapper. It executes the computations described by ExaQL (3.1). Madis processes the data in a streaming fashion and performs pipelining when possible, even for UDFs. The UDFs are executed inside the database along with the relational operators to push them as close to the data as possible.

2.1 Data Model

Exareme adopts the relational data model and extends it with:
Complex Field Types: JSON, CSV, and TSV.
Table Partitions: A table is defined as a set of partitions and a partition is defined as a set of records having a particular property, i.e. the hash value of a column.
Partitioning: If the database has multiple tables as it happens in data warehouses, the largest tables are partitioned and all others are replicated along with the partitions. Data placement is crucial for performance and elasticity. We use a modification of consistent hashing [10], because it offers good theoretical bounds and can be accurately modeled. To increase flexibility and efficiency we use over-partitioning and replication. This way, changing the size of the virtual infrastructure will cause only data transfer and not the computation of a new partitioning.

2.2 Money/Time Trade-Offs

Exareme can express money/time trade-offs by examining variations of an execution plan, we refer to this notion as eco-elasticity [8] [7]. Exareme's scheduler creates different execution plans based on the algorithm described in 4.1. Along with every query the user can specify an SLA. Using the SLAs the scheduler chooses the execution plan based on its time and money requirements.

3. LANGUAGE

Queries are issued to Exareme using ExaDFL. ExaDFL is a dataflow language that describes DAGs and it's based on SQL extended with UDFs and data parallelism primitives. ExaDFL allows fine control, but requires an understanding of partitioning and data placement.
We are currently working on an optimizer that will produce ExaDFL from UDF extended SQL by applying classic database optimizations and transforming functions with their distributed version when it is necessary.
In this section we firstly present the language that describes intra-worker dataflows. Then we present the data parallelism primitives and at the end we present the language as one.
We use the following subset of TPC-H [3] : lineitem(**l_orderkey**, l_comment), orders(**o_orderkey**,o_clerk). Both are hash partitioned to 4 parts on their keys.

3.1 ExaQL

ExaQL is based on the SQL-92 standard. The relational primitives of SQL are a good way to express relations and data combinations. We use SQL to combine data and process them with UDFs, whenever the SQL abstractions are not sufficient or efficient to use. We enhanced the syntax of ExaQL to easily combine virtual table functions (UDTFs) into data pipelines.
Suppose we want to find the most frequent words that some clerks use in their comments when they buy or sell products. We have the names of the clerks in a compressed XML file that is accessible via HTTP. In ExaQL, we can express it as follows:

```
select word, count(*) as count
from(select STRSPLITV(l_comment) as word
        from lineitem, orders, (XMLPARSE '["/name"]'
            FILE 'http://../clerk.xml.gz') as clerk
        where l_orderkey = o_orderkey and o_clerk = name) as words
group by word
order by count desc;
```

The query uses the FILE UDTF to fetch, uncompress, and load the data on-the-fly from the HTTP server specified. It is not needed to import or create temporary tables, all the details are handled automatically by the system. The output of FILE is given to the XMLPARSE UDTF that parses the XML content and produces a table with the names of the clerks. Row function STRSPLITV takes a string as input and produces one nested table for each comment by splitting the words into rows. Notice that this behaviour is different from the row functions typically supported by database systems which produce a single value. This is an extension of Exareme for row and aggregate functions.

3.2 Data Parallelism Primitives

The support of simple primitives declaratively express potential data parallelism in the dataflow language itself and let the system decide the actual degree of parallelism at runtime. This is very helpful since the queries are expressed independently of the parallelism used.

3.2.1 Input Primitives

Figure 3 (top) shows the types of combinations supported on two partitioned tables R and S, where a query Q is executed on each partition pair indicated, as well as the type of reduction supported on a single partitioned table.
Direct : This combines two (or more) tables that either (a) both have been partitioned in the way required by the combination specified, e.g., a distributed join on tables hash-partitioned on the join attribute, or (b) one has been fully replicated and the other has been partitioned in some fashion, e.g., a join between a small table replicated to the locations of the partitions of much larger table.
Cartesian product: This combines two (or more) tables that have been partitioned in ways unrelated to the combination specified.
Tree: This performs a multi-level tree reduction on a single table, generalizing the two-level (combine and reduce) reduction of MapReduce. This is used when Q has aggregate functions that are algebraic or distributive and has been shown to exhibit very good performance in practice.

3.2.2 Output Primitives

Figure 3 (bottom):
Same: The default mode does, the output number of partitions is determined by the input.

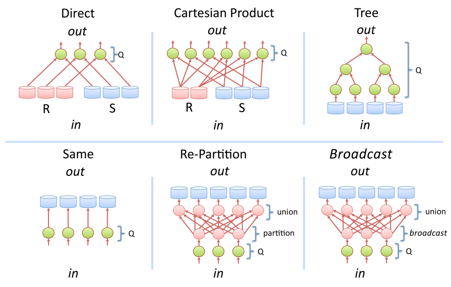

Figure 3: Input Partitioning (top), Output Partitioning (bottom)

Partition: Hash partitioning is used. This requires two steps: i) partition each of the input parts and ii) union each of the sub-partitions into the final output.
Broadcast: This creates full replicas of the output file, first broadcasting each partition to all relevant workers and then performing their union at each worker.

3.3 ExaDFL

All of the above compose ExaDFL according to the following grammar:

```
ExaDFL        := (<query>)+
query         := <parallelism> <ExaQL> ;
parallelism   := create distributed [temp]
                 table <name> [<output_comb>]
                 as [<input_comb>]
output        := [to <number>] [(hash | range)]
                 partition on <name>(,<name>)*
input         := direct | cprod | tree | extern
          (the rest is omitted due to space)
```

Two or more queries form a script. Each query has two semantically different parts: parallelism and ExaQL. The first part describes the input and output data parallelism used and the second part the computations that get executed on each input combination. The following ExaDFL dataflow is equivalent to the ExaQL query of our example:

```
// Query 1
create distributed temp table clerk to 4 as extern
   select name
   from (XMLPARSE '["/name"]' FILE 'http://../clerk.xml.gz');
// Query 2
create distributed temp table words as direct
   select word, count(*) as count_partial
   from (select STRSPLITV(l_comment) as word
                  from lineitem, orders, clerk
                  where l_orderkey = o_orderkey
                  and o_clerk = name)
   group by word;
// Query 3
create distributed table result as tree
   select word, sum(count_partial) as count
   from wordcount
   group by word
   order by count desc;
```

The first query is executed to download and parse the XML file. The extern directive declares that the query uses an external source and only one instance of the query should be created. The result is a table called clerk that is replicated to 4 partitions. The second query combines tables lineitem, orders, and clerk using the direct input combination. Notice that the result of the join is correct since tables lineitem and orders are partitioned on the join column and table clerk is replicated. Finally, the third query is used to create table result using a tree aggregation. This is possible because the aggregate function sum is distributive. All the temporary tables are deleted automatically at the end of the script.

4. QUERY OPTIMIZATION

In principle, the optimization process could proceed in one giant step, examining all execution plans that could answer the query and choosing the optimal that satisfies the required constraints. Given the size of the alternatives space in our setting, this approach is infeasible. Instead, our optimization process proceeds in multiple smaller steps, each one operating at some level and making assumptions about the levels below. This is in analogy to query optimization in traditional databases but with the following differences. The operators may represent arbitrary operations and may have performance characteristics that are not known. Furthermore, optimality may be subject to QoS or other constraints and may be based on multiple criteria, e.g., monetary cost of resources, quality of data, etc., and not just solely on performance.The resources available for the execution of a dataflow are not fixed a-priori but flexible and reservable on demand.

4.1 Sky

The dataflow scheduler we use, takes as input the dataflow DAG and assigns its nodes (operators) to workers. It does so by taking into account two types of constraints i) the dataflow (DAG) implied constraints based on the inter-operator dependencies captured by its edges, ii) the execution environment implied constraints due to resource limitations. In that respect, we categorize resources as time-shared and space-shared [9]. Time-shared resources can be used by multiple operators concurrently at very low overhead. Concurrent use of space-shared resources implies high overheads beyond workers limits of resources. We consider memory as the only space-shared resource, whereas CPU and network as time-shared resources. Constraints are imposed only by space-shared resources in every worker, at any given moment, memory must be sufficient for the execution of the running operators. The scheduling algorithm we propose is Dynamic Skyline (Sky) and is shown in Algorithm 1.

Sky is an iterative algorithm that incrementally computes the skylines of schedules, Figure 2. The algorithm begins by scheduling the operators from producers to consumers as defined by the DAG. Each operator with no inputs is a candidate for assignment. An operator is a candidate as soon as all of its inputs are available. The scheduler considers assigning every operator at an existing worker or at a new worker by adding a new VM. The result is a skyline of schedules (Figure 2). The final execution plan can be selected either manually or automatically based on SLAs. [16] The scheduler uses the following heuristics regarding data transferring. It transfers only intermediate results and, if possible, does not move original tables. Intermediate results are usually smaller than the original tables because queries with a single input usually contain filters and queries with multiple inputs usually join the tables using equi-joins. This type of join reduces the size of the output table. Some type of queries are executed very efficiently this way, especially when the small tables fit in memory. This is the usual case

Algorithm 1 Dynamic Skyline

Input: G: A dataflow graph.
Output: *skyline*: The skyline schedules.

```
 1: $ready \leftarrow$ {operators in $G$ that have no dependencies}
 2: $op_1 \leftarrow maxRunningTime(ready)$
 3: $vm_1 \leftarrow allocateNewVM()$
 4: $schedule_1 \leftarrow \{assign(op_1, vm_1, -, -)\}$
 5: $skyline \leftarrow \{schedule_1\}$
 6: while $ready \neq \oslash$ do
 7:    $next \leftarrow maxRunningTime(ready)$; $S \leftarrow \oslash$
 8:    for $s \in skyline$ do
 9:       if $next$ is pinned then
10:          $S \leftarrow S \cup \{s + assign(next, next.pin\_loc, -, -)\}$
11:       else
12:          for all containers $c$ of $s$ do
13:             $S \leftarrow S \cup \{s + assign(next, c, -, -)\}$
14:          end for
15:          // Consider allocating a new VM
16:          $new\_vm \leftarrow allocateNewVM()$
17:          $S \leftarrow S \cup \{s + assign(next, new\_vm, -, -)\}$
18:          releaseNotNeeded($s$)
19:       end if
20:    end for
21:    // Only skyline schedules (i.e., prune search space)
22:    $skyline \leftarrow$ skyline of $S$
23:    $ready \leftarrow ready - \{next\} \cup$ {operators in $G$ that dependency constraints no longer exist}
24: end while
25: return $skyline$
```

for OLAP workloads with star or snowflake schema. Another benefit with this approach is the exploitation of indexes if they exists on the original tables. In addition, we add gravity operators pinned to the location of the tables, so the movement of the original tables out of their initial location becomes an optimization choice.

5. EXPERIMENTAL EVALUATION

Environment: We used up to 64 VMs, each with 1 CPU, 4 GB of memory, and 20 GB of disk, provided by Okeanos[2]. The average network speed measured was 150 Mbps.
Datasets: We generated a total of 256 GB of the following tables, using the TPC-H benchmark [3]. (in the parenthesis we note the number of partition and the partitioning key) region(1), partsupp(1, ps_partkey), orders(128, o_orderkey), lineitem(128, l_orderkey), customer(1, c_custkey), part(1, p_partkey), nation(1), and supplier(1).
Measurements: We run each query 4 times and report the average of the last 3 measurements. We compared Exareme with Hive [15] (with both MR [4] and Tez [5] as backend, formerly known as Stinger) and System X (an industry leading commercial system). Figure 4 shows the results, to save space we have omitted some results, but only if Exareme is faster. Hive-stinger was always faster than Hive. The versions of the systems we used are Hive 0.13.1, Hadoop 2.5.1, Tez 0.5.0 (intermediate results are compressed (Snappy)). System X is faster for queries 1 and 6 that involve aggregations on the largest table (lineitem). We were not able to execute queries 8 and 9 on System X because of memory limits (System X is an in-memory system). Overall, we observe that Exareme is faster in most cases than the state-of-the art systems.

[2]okeanos.grnet.gr

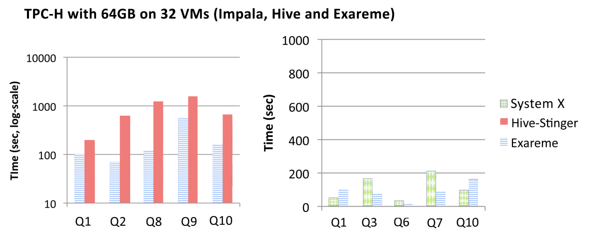

Figure 4: TPC-H with 64GB data and 32 VMs on System X, Hive and Exareme

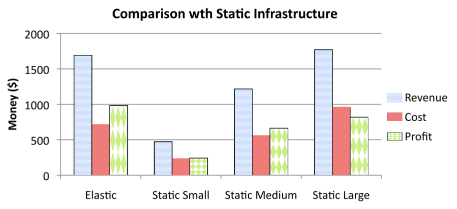

Figure 5: configuration with eco-elasticity vs. static layouts.

Figure 5 show the profit that is gained when exploiting eco-elasticity. As a baseline we use three static infrastructures that do not change over time small with 15 VMs, medium with 30 VMs, large with 60 VMs. We run the system for one hour using a client that issue Q1 in three phases, each of 1 hour duration. In the first and third phase, the Poisson parameter λ is set to 60 and in the second phase to 30 (the rate is doubled). The elastic layout allocator produces a better-fitted layout that adapts to the workload changes and yields the highest profit compared to all static choices.

6. ACKNOWLEDGEMENTS

The authors would like to thank Herald Kllapi and Manolis Tsangaris.

7. REFERENCES

[1] Exareme. http://www.exareme.org/.
[2] Madis. https://github.com/madgik/madis.
[3] TPC-H Benchmark, http://www.tpc.org/tpch/.
[4] Apache. Hadoop, http://hadoop.apache.org/.
[5] Apache. Tez, http://tez.apache.org/.
[6] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In *OSDI*, 2004.
[7] E. Deelman et al. The cost of doing science on the cloud: the montage example. In *IEEE/ACM SC*, 2008.
[8] K. et al. Schedule optimization for data processing flows on the cloud. *SIGMOD '11*, page 289, 2011.
[9] M. Garofalakis and Y. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. VLDB '97.
[10] D. R. Karger et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663, 1997.
[11] H. Kllapi, B. Harb, and C. Yu. Near neighbor join. In *ICDE*.
[12] M. J. Litzkow et al. "Condor - A Hunter of Idle Workstations". In *ICDCS*, pages 104–111, 1988.
[13] S. Melnik et al. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.
[14] A. Simitsis. Modeling and managing etl processes. In *VLDB PhD Workshop*, 2003.
[15] A. Thusoo et al. "Hive - a petabyte scale data warehouse using Hadoop". In *ICDE*, 2010.
[16] H. R. Varian. *"Intermediate Microeconomics : A Modern Approach"*.