THE LONG ROAD TOWARDS ELASTIC DISTRIBUTED STREAM PROCESSING

Leonardo Querzoni
querzoni@diag.uniroma1.it

Auto-DaSP - Turin, August 28th 2018
ELASTIC COMPUTING

"[...] defines elasticity as the configurability and expandability of the solution [...] Centrally, it is the ability to scale up and scale down capacity based on subscriber workload.”
OCDA. Master Usage Model: Compute Infrastructure as a Service. Tech. rep., Open Data Center Alliance (OCDA), 2012

"Elasticity is basically a 'rename' of scalability [...]” and "removes any manual labor needed to increase or reduce capacity”
SCHOUTEN, E. (IBM) Rapid Elasticity and the Cloud, September 2012

"Rapid elasticity: Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.”

"the quantifiable ability to manage, measure, predict and adapt responsiveness of an application based on real time demands placed on an infrastructure using a combination of local and remote computing resources.”

"Elasticity measures the ability of the cloud to map a single user request to different resources.”
WOLSKI, R. Cloud Computing and Open Source: Watching Hype meet Reality, May 2011
ELASTIC COMPUTING

"[…] defines elasticity as the configurability and expandability of the solution […] Centrally, it is the ability to scale up and scale down capacity based on subscriber workload."
OCDA. Master Usage Model: Compute Infrastructure as a Service. Tech. rep., Open Data Center Alliance (OCDA), 2012

"Elasticity is basically a 'rename' of scalability […]" and "removes any manual labor needed to increase or reduce capacity"
SCHOUTEN, E. (IBM) Rapid Elasticity and the Cloud, September 2012

"Rapid elasticity: Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time."

"the quantifiable ability to manage, measure, predict and adapt responsiveness of an application based on real time demands placed on an infrastructure using a combination of local and remote computing resources."

"Elasticity measures the ability of the cloud to map a single user request to different resources."
WOLSKI, R. Cloud Computing and Open Source: Watching Hype meet Reality, May 2011
ELASTIC COMPUTING

Load

Static Provisioning

Workload vs. Time

THE LONG ROAD TOWARDS ELASTIC DISTRIBUTED STREAM PROCESSING - AUTODASP 2018
ELASTIC COMPUTING

- Load
- Static Provisioning

Graph showing workload vs. time with static provisioning and load curves.
ELASTIC COMPUTING

Workload

I didn’t have any accurate numbers so I just made up this one.

Studies have shown that accurate numbers aren’t any more useful than the ones you make up.

How many studies showed that?

Eighty-seven.

Load

Static Provisioning

Time
ELASTIC COMPUTING

Workload

Time

Load
Static Provisioning
Elastic Provisioning
ELASTIC COMPUTING

Load

Static Provisioning

Elastic Provisioning

Workload vs. Time
ELASTIC COMPUTING

The diagram illustrates the comparison between static and elastic provisioning in the context of workload and load over time.

- **Load**
- **Static Provisioning**
- **Elastic Provisioning**

The graph shows how workloads and loads change over time, highlighting the differences and advantages of elastic provisioning over static provisioning.
ELASTIC COMPUTING

Load
Static Provisioning
Elastic Provisioning

Workload
Time

Underprovisioning
Overprovisioning

THE LONG ROAD TOWARDS ELASTIC DISTRIBUTED STREAM PROCESSING - AUTODASP 2018
ELASTIC COMPUTING

- Load
- Static Provisioning
- Elastic Provisioning

Elastic provisioning

Workload

Time
ELASTIC COMPUTING

Elastic computing drove the success of cloud providers

- Virtually infinite resources
- On-demand provisioning
- Near-instant availability
- Automatic scale-out
- Pay-what-you-use
ELASTIC COMPUTING

Elastic processing of big-data is today a reality
DISTRIBUTED STREAM PROCESSING

Data Stream Processing Engine:

- continuously calculate results for persistent queries
- on (potentially) unbounded data streams
- using operators: algebraic (filters, join, aggregation) or user defined
- stateless/stateful
Data stream processing (DSP) was in the past considered a solution for very specific problems.

- Financial trading
- Logistics tracking
- Factory monitoring

Today the potentialities of DSPs start to be used in more general settings.

**DSP : online processing = MR : batch processing**
ELASTIC STREAM VS BATCH

Why is realizing elastic stream processing more difficult?

- **Data in motion vs data at rest**
  - Variable data rates
  - No obvious ways to characterize data content

- **Latency-sensitive applications**
  - Batch applications are typically throughput-oriented

- **Long term executions**
  - Batch jobs are expected to be short-lived
  - Stream processing applications are designed to stay up and running for hours/days/week/months
HOW TO SCALE DSP

A few optimization strategies are known to deal with these issues:

HOW TO SCALE DSP

A few optimization strategies are known to deal with these issues:

HOW TO SCALE DSP

A few optimization strategies are known to deal with these issues:


**FUSION**

**FISSION**
A few optimization strategies are known to deal with these issues:

A few optimization strategies are known to deal with these issues:


**FUSION**

**FISSION**

**PLACEMENT**

**LOAD BALANCING**
CURRENT SOLUTIONS

Most of the existing solutions apply a standard MAPE-K (monitor, analyze, plan, and execute) model:
Performance about the runtime execution of stream applications is gathered at several possible collection points:

- **Hosts-level**
  - memory/cpu utilization
  - interprocess communications

- **Network-level**
  - communications among hosts in the cluster
  - link congestion

- **Application level**
  - Metrics exposed by the framework (e.g. operator selectivity, buffer congestion, etc.)
  - Metrics exposed by software stacks (e.g. thread CPU utilization, heap size, etc.)
CURRENT SOLUTIONS

ANALYZE

Collected data is analyzed to take scale-in/out decisions. Conditions are usually expressed on thresholds:

- **Static** - rely on domain knowledge or sysadmin expertise
- **Dynamic** - thresholds are automatically recomputed at runtime depending on monitored data

Thresholds can be checked (Heinze et al, 2014)

- **Locally** - they evaluate the current status of each single host
- **Globally** - represent the system as a whole
ANALYZE

Collected data is analyzed to take scale-in/out decisions.

Decisions can be

- **Reactive**: decisions are based only on conditions expressed on monitored data (i.e. on past events)
- **Proactive**: decisions are based on models that, on the basis of monitored data, predict future expected behaviors.

Other fundamental factors that you should take into account:

- **State migration**
  - Partitioned vs. “monolithic” state
- **Reconfiguration approach** and its overhead
  - Pause & Resume vs. Parallel Tracks
A new configuration for the runtime is planned that aims at maximizing a metric or satisfying some objective functions.

- Heuristics (many different greedy approaches)
- Integer Linear Programming (Cardellini et al., 2017)
- Predictive performance modelling (Li et al., 2016)
- Game Theory (Mencagli, 2016)

...
EXECUTE

This phase is usually framework-dependent

- The framework scheduler must be instructed to deploy the plan output from the previous phase
- Depending on the framework you may incur possible extra-latencies!
  - Full re-scheduling vs. incremental deployment update
OPEN PROBLEMS

There are still a few open questions that require further research:

- **Interplay between operator scaling and resource scaling** [Lombardi et al. TPDS 2018]
- Interplay between operator parallelism and operator placement [Cardellini et al., CCP&E 2017]
- Interplay between operator parallelism and co-location (e.g. Spark/Flink)
- Interplay among applications sharing the same cluster
- **Sensitivity to load imbalance** [Gedik et al, 2014, Rivetti et al. 2015]
- **Sensitivity to data distribution** [Rivetti et al. 2015]
- Latency vs throughput goals [Cardellini et al., CCP&E 2017, Luthra et al, DEBS 2018]
Current solutions typically address the problem of elastically scale operator through fusion

- Resources are considered static (i.e. over-provisioning) or...
- they assume that a new resource can be “magically” instantiated for each new operator instance (i.e. “joint scaling”)

Why can’t we consider operator instances and computing resources as distinct solutions to possibly different problems?

- Scale-in/out operator instances through the DSP framework.
- Scale-in/out computing resources through the cloud provide APIs.
- Use an autonomic controller to symbiotically manage both solutions.
OPERATOR/RESOURCE SCALING

Why can’t we consider operator instances and computing resources as distinct solutions to possibly different problems?
OPERATOR/RESOURCE SCALING

OPERATOR/RESOURCE SCALING


Leads input workload patterns using neural networks
OPERATOR/RESOURCE SCALING


Learns input workload patterns using neural networks

Learns how the application topology handles the workload
**OPERATOR/RESOURCE SCALING**


- Learns input workload patterns using neural networks
- Learns how the application topology handles the workload
- Learns how each operator uses its assigned computing resources when the workload varies

- Learns input workload patterns using neural networks
- Learns how the application topology handles the workload
- Learns how each operator uses its assigned computing resources when the workload varies
- Learns how much overhead is produced by the DSP framework
OPERATOR/RESOURCE SCALING


- Learns input workload patterns using neural networks
- Learns how the application topology handles the workload
- Learns how each operator uses its assigned computing resources when the workload varies
- Learns how much overhead is produced by the DSP framework

The outputs from the profilers constitute the application description parameters
### OPERATOR/RESOURCE SCALING

**ELYSIUM**: Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems [Lombardi et al., TPDS 2017]

<table>
<thead>
<tr>
<th>Selectivity Profiler</th>
<th>CPU Performance Profiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. For each edge $xy$ compute: $\alpha(xy) = \frac{\text{out}(xy)}{\text{in}(x)}$</td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image)

1. For each edge $xy$ compute: $\alpha(xy) = \frac{\text{out}(xy)}{\text{in}(x)}$

2. Store $\alpha$ in $\alpha$ Selectivity Table

<table>
<thead>
<tr>
<th>Edge</th>
<th>Selectivity $\alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>1.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>BC</td>
<td>0.5</td>
</tr>
<tr>
<td>BD</td>
<td>2.0</td>
</tr>
</tbody>
</table>

1. Build CPU Performance Tables

2. Store alfa in α Selectivity Table

Selectivity Profiler

1. For each edge xy compute: $\alpha(xy) = \frac{\text{out}(xy)}{\text{in}(x)}$

2. Store alfa in $\alpha$ Selectivity Table

CPU Performance Profiler

1. Build CPU Performance Tables

Selectivity Table

<table>
<thead>
<tr>
<th>Edge</th>
<th>Selectivity $\alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>BC</td>
<td>0.5</td>
</tr>
<tr>
<td>BD</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Input Stream (tuple/sec) | CPU (MHz)
-------------------------|---------|
100                      | 200     |
200                      | 400     |
...                      | ...     |
1,000                    | 2,000   |
OPERATOR/RESOURCE SCALING


1. For each edge xy compute: \( \alpha(xy) = \text{out}(xy) / \text{in}(x) \)

2. Store \( \alpha \) in \( \alpha \) Selectivity Table

<table>
<thead>
<tr>
<th>Edge</th>
<th>Selectivity ( \alpha )</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>1.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>BC</td>
<td>0.5</td>
</tr>
<tr>
<td>BD</td>
<td>2.0</td>
</tr>
</tbody>
</table>

1. Build CPU Performance Tables

2. Create and train an ANN for each operator:

input stream \( \rightarrow \) ANN \( \rightarrow \) CPU
OPERATOR RESOURCE SCALING

OPERATOR/RESOURCE SCALING


Estimates how many computing resources are needed by an operator instance in a given topology configuration
OPERATOR/RESOURCE SCALING


Calculates a new scaling configuration and simulates its scheduling allocating the minimum number of needed resources

Estimates how many computing resources are needed by an operator instance in a given topology configuration
Algorithm 1. AutoScaling Algorithm

1: function `COMPUTE_CONFIG` (Estimator $E$, Scheduler $S$, List (Application) $apps$, List (int) $input_loads$)
2: for all application $a_k$ in $apps$ do
3: for all operator $o_i$ in $a_k$ do
4: $ir_i \leftarrow E.getOperatorInputRate(input_loads_k, o_i)$
5: $p_i \leftarrow 1$
6: while $E.getOperatorInstanceCpuUsage(o_i, \frac{ir_i}{p_i}) > core_{max}.thr$ & $p_i < max_{parallel_i}$ do
7: $p_i \leftarrow p_i + 1$
8: while $E.getOperatorInstanceCpuUsage(o_i, \frac{ir_i}{p_i}) < core_{min}.thr$ & $p_i > 1$ do
9: $p_i \leftarrow p_i - 1$
10: $worker_nodes \leftarrow 1$
11: while true do
12: allocation $\leftarrow S.allocate(apps, worker_nodes)$
13: $cpu_usages \leftarrow E.getCpuUsages(allocation, input_loads)$
14: if $\forall x \in cpu_usages : x \leq cpu_{max}.thr$ then
15: return $worker_nodes, \{p_i\}$
16: $worker_nodes \leftarrow worker_nodes + 1$

Algorithm 1. AutoScaling Algorithm

1: function COMPUTECONFIG(Estimator $E$, Scheduler $S$, List $\langle$ Application $\rangle$ apps, List $\langle$ int $\rangle$ input_loads)
2:     for all application $a_k$ in apps do
3:         for all operator $o_i$ in $a_k$ do
4:             $ir_i \leftarrow E.getOperatorInputRate(input_loads_k, o_i)$
5:             $p_i \leftarrow 1$
6:             while $E.getOperatorInstanceCpuUsage(o_i, \frac{ir_i}{p_i}) > core_{max}\_thr \& p_i < max\_parallo$ do
7:                 $p_i \leftarrow p_i + 1$
8:             while $E.getOperatorInstanceCpuUsage(o_i, \frac{ir_i}{p_i}) < core_{min}\_thr \& p_i > 1$ do
9:                 $p_i \leftarrow p_i - 1$
10:         worker_nodes $\leftarrow 1$
11:     while true do
12:         allocation $\leftarrow S.allocate(apps, worker_nodes)$
13:         cpu_usages $\leftarrow E.getCpuUsages(allocation, input_loads)$
14:         if $\forall x \in cpu\_usages : x \leq cpu_{max}\_thr$ then
15:             return worker_nodes, $\{p_i\}$
16:         worker_nodes $\leftarrow worker\_nodes + 1$

For each application calculates the number of parallel instances of each operator needed to sustain the predicted input rate.

**Algorithm 1. AutoScaling Algorithm**

1: function COMPUTECONFIG(Estimator $E$, Scheduler $S$, List (Application) $apps$, List (int) $input_loads$)
2: for all application $a_k$ in $apps$ do
3:   for all operator $o_i$ in $a_k$ do
4:     $ir_i \leftarrow E.getOperatorInputRate(input\_loads_k, o_i)$
5:     $p_i \leftarrow 1$
6:     while $E.getOperatorInstanceCpuUsage(o_i, \frac{ir_i}{p_i}) > core\_max\_thr$ & $p_i < max\_parall_o_i$ do
7:       $p_i \leftarrow p_i + 1$
8:     while $E.getOperatorInstanceCpuUsage(o_i, \frac{ir_i}{p_i}) < core\_min\_thr$ & $p_i > 1$ do
9:       $p_i \leftarrow p_i - 1$
10:    worker_nodes $\leftarrow 1$
11: while true do
12:   allocation $\leftarrow S.allocate(apps, worker\_nodes)$
13:   cpu\_usages $\leftarrow E.getCpuUsages(allocation, input\_loads)$
14:   if $\forall x \in cpu\_usages : x \leq cpu\_max\_thr$ then
15:     return worker\_nodes, $\{p_i\}$
16:   worker\_nodes $\leftarrow worker\_nodes + 1$

Simulates the scheduling of applications and increases the number of provisioned computing resources until no host is (predictably) overloaded.
OPERATOR/RESOURCE SCALING


- Comparison between real and estimated total CPU usage (in Hz) for all instances of Counter and StopWordFilter operators. Sinusoidal input rates.
OPERATOR/RESOURCE SCALING


- ELYSIUM handling concurrent applications with different input rates.
OPERATOR/RESOURCE SCALING


- ELYSIUM proactive vs reactive behavior
OPERATOR/RESOURCE SCALING


- ELYSIUM resource saving

![Bar charts showing throughput degradation and nodes saved for different scenarios]
SENSITIVITY TO LOAD IMBALANCE

Most solutions for elastic scaling assume that

- input rate can vary in time
- input content is uniformly distributed

The second assumption is unrealistic.

- Think about the distribution of keys in obvious applications (e.g. rolling top-k words)

Non-uniform content distribution has strong impact at runtime

- Skewed memory footprint for partitioned-state operators
- Skewed load on parallelized operator instances
- Things get more complicated if computations with different complexities are performed for different tuples.
SENSITIVITY TO LOAD IMBALANCE

Example:

- Implementation in Storm of the 3rd query of the DEBS 2013 Grand Challenge
- Running on a 8-cores 2 GHz Intel Xeon (16 logical cores) with 32 GB of RAM

Throughput (tuples / s) as a function of the number of instances

![Graph showing throughput vs number of instances]
SENSITIVITY TO LOAD IMBALANCE

Solution: [Gedik et al., 2014]

- Ad-hoc mapping of “heavy hitters” to operator instances. Hashing for the rest.

9 key values make up for roughly 38% of the stream
SENSITIVITY TO LOAD IMBALANCE

Solution: [Gedik et al., 2014]

- Ad-hoc mapping of “heavy hitters” to operator instances. Hashing for the rest.

9 key values make up for roughly 38% of the stream

Few key values may cause most unbalance

1. One-to-one mapping
SENSITIVITY TO LOAD IMBALANCE

Solution: [Gedik et al., 2014]

- Ad-hoc mapping of “heavy hitters” to operator instances. Hashing for the rest.

![Graph showing skewed key value distribution with heavy hitters (HH1, HH2, HH3, HH4) and non-frequent key values (Sparse Items) with probability distribution.](image-url)
SENSITIVITY TO LOAD IMBALANCE

Solution: [Gedik et al., 2014]

- Technical issues
  - Large number of keys. How to keep track of all their frequencies in the stream?
  - While achieving balance, the system should also maintain low migration cost
- Solution: Use lossy counting to keep track of key frequencies
  - Count elements in rounds
  - Remove less frequent elements at each round end
  - Space saving has tighter theoretical bounds on memory complexity
- Use several counters over tumbling windows to emulate a sliding window
  - Can keep track of load distributions evolving in time
  - Manages state migration
- Problem: is the impact from non-frequent keys (sparse items) really negligible?
SENSITIVITY TO LOAD IMBALANCE

Solution: [Rivetti et al., DEBS 2015]

- Ad-hoc mapping of heavy hitters AND groups of sparse items.

- 991 key values make up for roughly 62% of the stream

- Sparse Items do not cause unbalance

- Handle Sparse Items with the standard solution
SENSITIVITY TO LOAD IMBALANCE

Solution: [Rivetti et al., DEBS 2015]

- Ad-hoc mapping of heavy hitters AND groups of sparse items.

![Skewed key value distribution graph]

- Worst Case partitioning
SENSITIVITY TO LOAD IMBALANCE

Solution: [Rivetti et al., DEBS 2015]

- Ad-hoc mapping of heavy hitters AND groups of sparse items.
SENSITIVITY TO LOAD IMBALANCE

Solution: [Rivetti et al., DEBS 2015]

- Ad-hoc mapping of heavy hitters AND groups of sparse items.

Each single key value does not cause unbalance

2. Many-to-one mapping
SENSITIVITY TO LOAD IMBALANCE

Solution: [Rivetti et al., DEBS 2015]

- Ad-hoc mapping of heavy hitters AND groups of sparse items.

![Diagram showing sensitivity to load imbalance with key value distribution](image)

- Each single key value does not cause unbalance.
SENSITIVITY TO LOAD IMBALANCE

Solution: [Rivetti et al., DEBS 2015]

- Ad-hoc mapping of heavy hitters AND groups of sparse items.
SENSITIVITY TO LOAD IMBALANCE

Solution: [Rivetti et al., DEBS 2015]

- Mapping also sparse items actually makes a difference
SENSITIVITY TO LOAD IMBALANCE

Solution: [Rivetti et al., DEBS 2015]

- Mapping also sparse items actually makes a difference

Throughput (tuples / s) as a function of the number of instances

![Graph showing throughput as a function of the number of instances. The x-axis represents the number of instances (k), and the y-axis represents the throughput (tuples / s). Two lines are shown: one for Apache Storm Standard and another for DKG Key Grouping.](image)
SENSITIVITY TO LOAD IMBALANCE

With stateless operators the same happens if computation latency depends on the tuple content:

\[ \sigma = \begin{cases} \text{Short} & \text{Execution time} \\ \text{Long} & \text{execution time} \end{cases} \]

Round-Robin

Unfeasible → Approximation?

Online
Full-Knowledge

Completion Time
Gain

OP O₁

OP O₂

OP O₁

OP O₂
SENSITIVITY TO LOAD IMBALANCE

Solution: [Rivetti et al., 2016]

- Dynamically schedule incoming tuples
SENSITIVITY TO LOAD IMBALANCE

Solution: [Rivetti et al., 2016]

- Dynamically schedule incoming tuples
THANKS!

Time for questions?