Contributions to Large-Scale Data Processing Systems

PhD Defense

Matthieu Caneill

February 5, 2018

Daniel Hagimont
INP Toulouse
ENSEEIHT

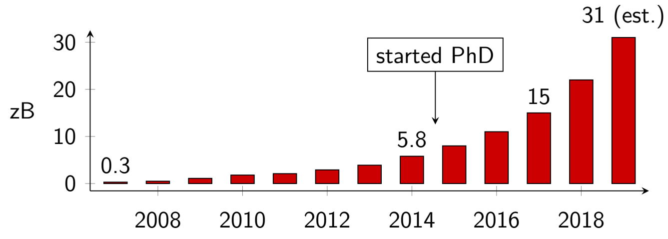
Jean-Marc Menaud
IMT Atlantique

Sihem Amer-Yahia
CNRS / Université Grenoble Alpes

Noël De Palma
Université Grenoble Alpes

Motivation

Worldwide data production



$1 zetabyte = 1000 exabytes = 10^6 petabytes = 10^9 terabytes$

(1 zetabyte is 2 billion times my hard drive)

Motivation

Applications

- Genome sequencing and querying (human: 3 B base pairs)
- Web and social networks (Facebook: 600 TB/day in 2014)
- Particle physics (CERN: 1 PB/s of collision data)
- etc.

Problems

- Data management at scale
- Data processing in reasonable time
- ... and reasonable price

Research questions

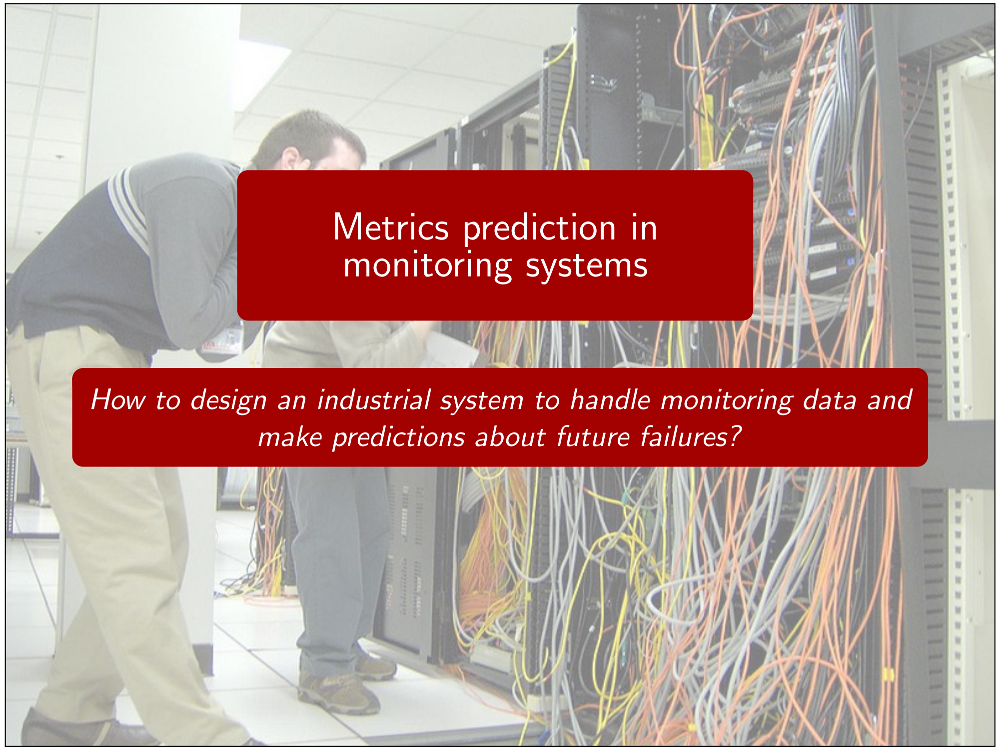
How to design...

- An industrial system to handle monitoring data and make predictions about future failures?
- An algorithm to improve locality in distributed streaming engines?
- A framework to compose data processing algorithms in a descriptive fashion, while reasoning on high level abstractions?

Outline

Structure of this presentation

1. Online metrics prediction in monitoring systems
2. Locality data routing
3. λ-blocks
4. Conclusion



Metrics prediction in monitoring systems

How to design an industrial system to handle monitoring data and make predictions about future failures?

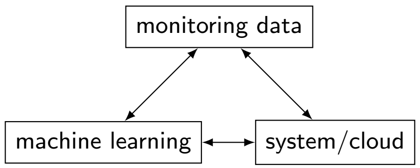
Metrics prediction in monitoring systems

Actors and roles of Smart Support Center

- **Coservit**: Monitoring services
- **HP**: Cloud computing, hardware
- **LIG – AMA**: Machine learning
- **LIG – ERODS**: Cloud computing, systems

Metrics prediction in monitoring systems

Scope of Smart Support Center



- Monitoring insights
- Failure prediction
- Infrastructure scaling
- More server uptime

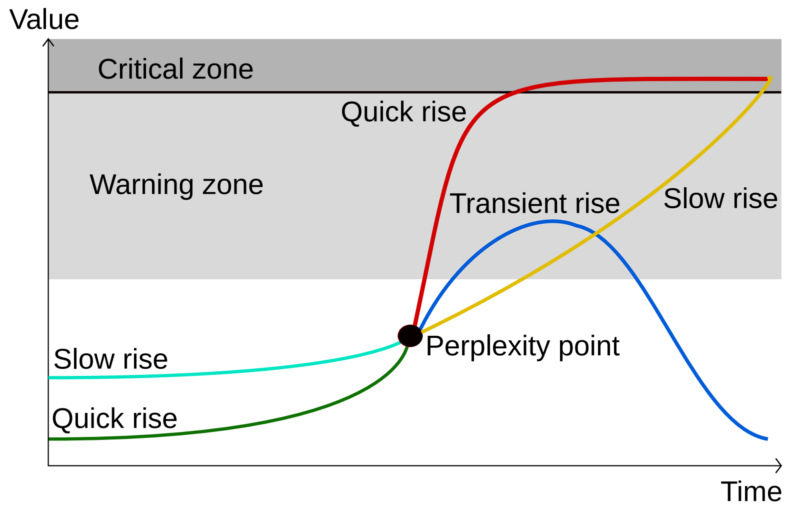
Challenges

- Scale monitoring infrastructure (from 1 to N nodes)
- System design for low latency analytics
- Fault tolerance

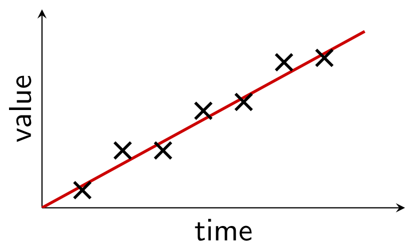
Metrics

- Monitoring metric: observation point on a server in a datacenter
- CPU load, memory, service status
- Reported by agents, processed, and stored
- Computed as time-series
- Associated to thresholds: warning and critical

Metrics behaviour: 6 scenarios

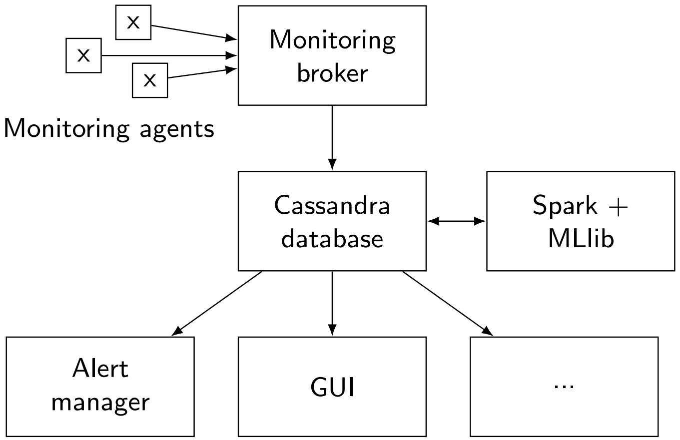


Linear regression



- Ability to identify local trends (few hours)
- Fast to compute
- Good candidate to avoid false positives (peaks)
- Library: MLlib (part of Apache Spark)

System architecture



System architecture

Desired properties

- Scalable: up to a few servers (150 CPU cores) to handle Coservit's load
- End-to-end fault tolerance: metrics can never be lost
- Performances: "fast" to compute metrics predictions

Evaluation

Setup

- Hardware: 4 servers (16–28 cores, 128–256 GB RAM)
- Dataset: Replay on production data recorded at Coservit
- 424 206 metrics, 1.5 billion data points monitored on 25 070 servers

Evaluation

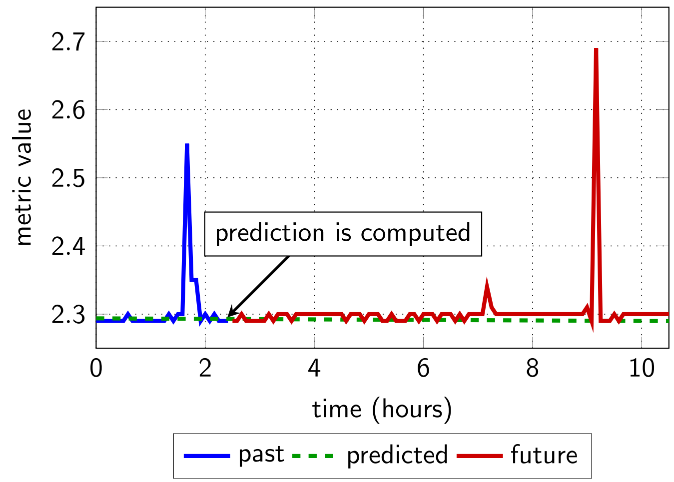


Figure: swap memory

Evaluation

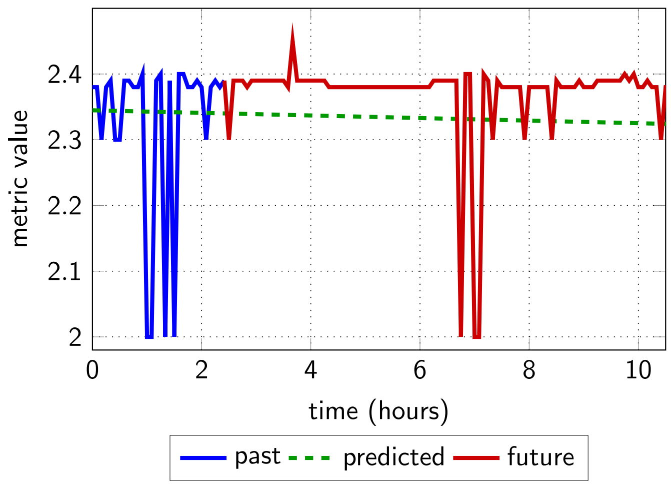


Figure: physical memory

Evaluation

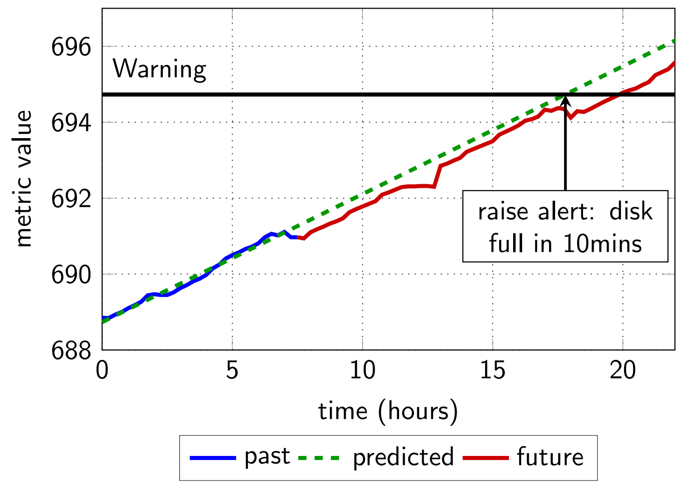


Figure: disk partition

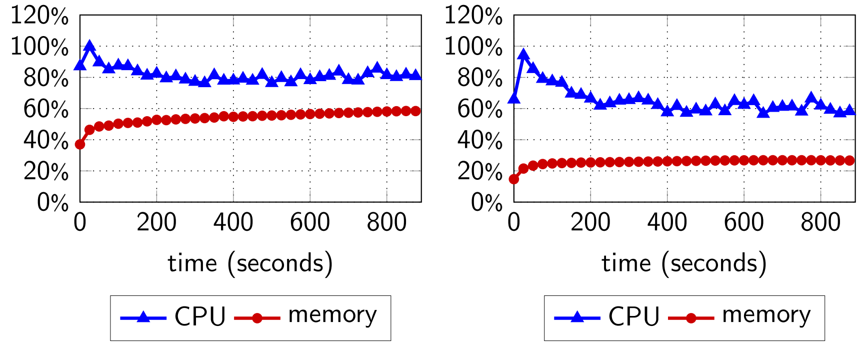
Evaluation

Metric blacklisting

- Some metrics are too volatile and hard to predict
- To avoid false positives/negatives, and save resources, they are blacklisted
- Root Mean Square Error evaluated weekly
- Metrics (temporarily) blacklisted if their RMSE $>$ threshold
- 58.5% of the metrics have a low RMSE $\rightarrow$ good predictions

Evaluation

CPU load and memory consumption



(a) master (b) slave-1

Figure: Running on 4 machines and 100 cores for 15 minutes.

Time repartition

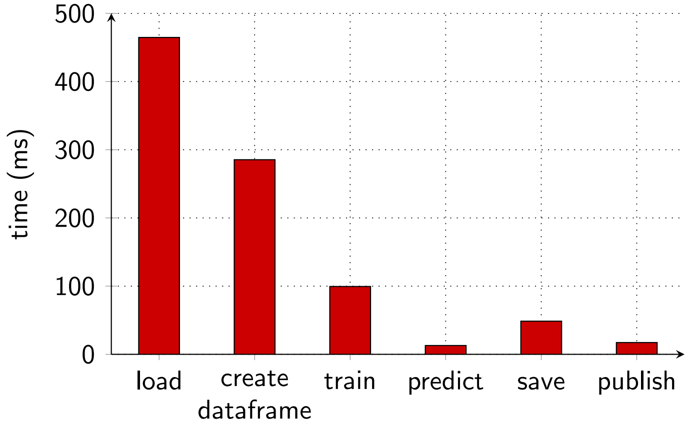


Figure: Time repartition for predicting a metric.

Load handling

- End-to-end process for the prediction of 1 metric: 1 second.
- One monitoring server (with 24 cores) can handle the load of 1440 metrics (at worst), which is 85 servers on average.

Load handling: linear scaling

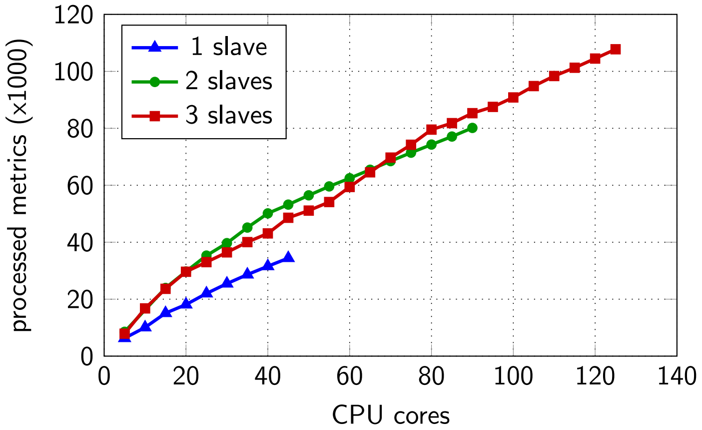


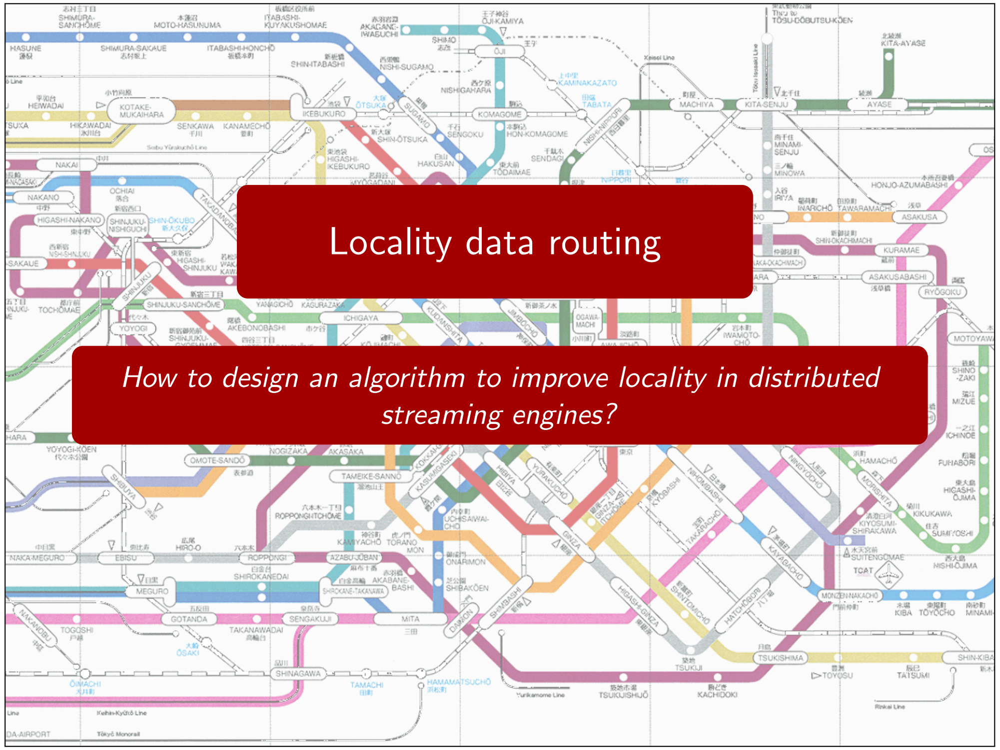
Figure: Amount of metrics handled in 15 minutes.

Positioning

No published work exhibits the same system (end-to-end system for monitoring metrics prediction, storage and blacklisting).

Prediction models

- Hardware failures [CAS12]
- Capacity planning (e.g. Microsoft Azure [mic])
- Datacenter temperature (e.g. Thermocast [LLL+11])
- Monitoring metrics (e.g. Zabbix [zab] with manual tuning)



Locality data routing

How to design an algorithm to improve locality in distributed streaming engines?

Locality data routing

Actors

Collaboration with **Vincent Leroy** (SLIDE) and **Ahmed El-Rheddane** (ERODS).

Locality data routing

Distributed streaming engines

Goals

- Real-time message handling
- Real-time metric calculations
- Parallelization
- Fault-tolerance

Locality data routing

Distributed streaming engines

Apache Storm → topologies.

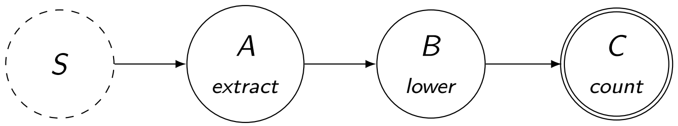


Figure: Trending hashtags topology.

S sends tweets, operator A extract hashtags, B converts them to lowercase, and C counts the frequency of each hashtag.

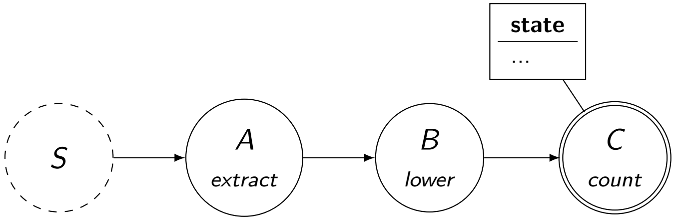
Division into tasks → distribution and parallelization made easy.

Locality data routing

Stateful operators

States are associated to keys

For example, the operator C can keep the list of trending hashtags (values) per location (keys).



Locality data routing

Stateful operators

Parallelization

To keep a consistent state, same keys must be routed to the same instance.

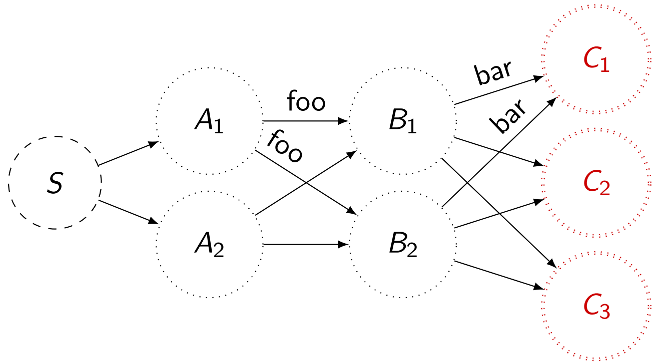
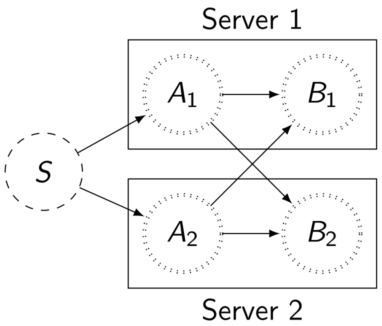


Figure: Tasks A and B are stateless, C is stateful.

Locality data routing

Situation

Let's have two stateful operators, each with two instances.



Goal

Minimize the traffic between the machines: $A1 \to B2$ and $A2 \to B1$. By default, $locality = 1/parallelism$

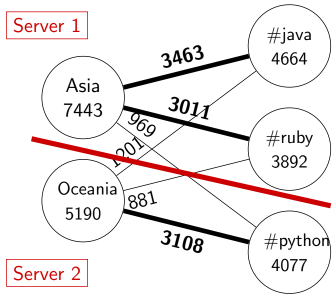
Constraint

Keep a good load balance between the machines.

Locality data routing

Keys correlation

Dynamically instrument the keys couples and represent them with a bipartite graph.

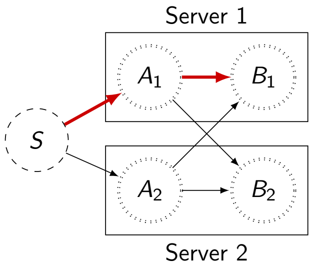


Routing tables

- S: $Asia \to A_1$
  $Oceania \to A_2$
- A_1: $\mathbf{\#java \to B_1}$
  $\mathbf{\#ruby \to B_1}$
  $\#python \to B_2$
- A_2: $\mathbf{\#python \to B_2}$
  $\#java \to B_1$
  $\#ruby \to B_1$

Graph partitioning $\to$ optimized routing, favorizing local links.

Message: #python is pretty cool!
Posted from: Oceania



S

key	**route**
Oceania	A1
Asia	A2

A

key	**route**
python	B1
java	B2

Trends evolve with time

Correlations between keys change frequently.

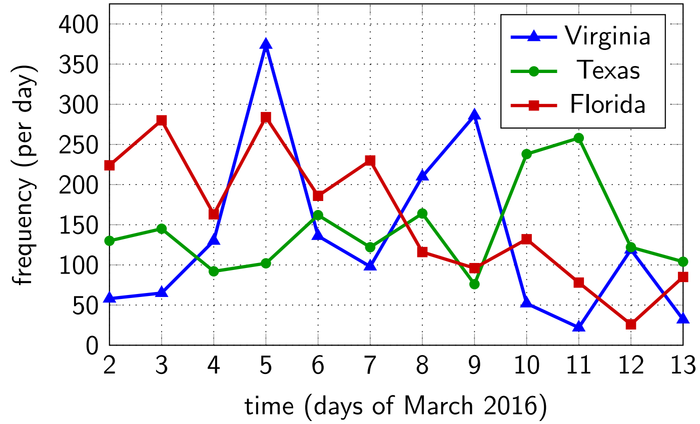


Figure: #nevertrump, in March 2016

Locality decay

- Keys correlations evolve with time.
- Routing tables optimized by examining old data lead to decreased locality.

Reconfiguration

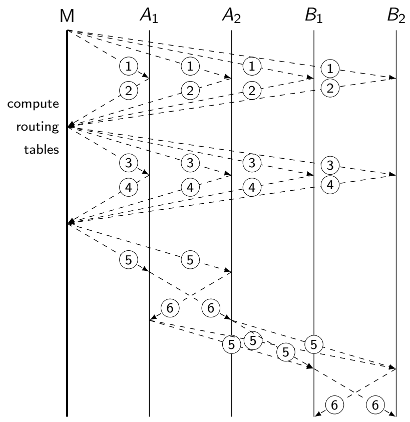
- We re-compute the tables every N minutes.
- Difficulty: keep the state consistent.

Reconfiguration protocol

Solution: online reconfiguration protocol

- update the routing tables in a live system
- without losing any message and state

Reconfiguration protocol



1. Get statistics
2. Send statistics

Partition graph, compute routing tables

3. Send reconfiguration
4. Send ACK
5. Propagate
6. Transfer key states

Propagate to next operator

Evaluation

Datasets

- From Flickr and Twitter
- Fields: location (country or place), hashtag
- Size: 173M records (Flickr), 100M (Twitter)

Setup

- 8× 128 GB RAM, 20 cores.
- Computation of aggregated statistics (stateful workers).
- Parallelism (2..6), network speed (1Gb/s | 10Gb/s), message size (0..20kB).

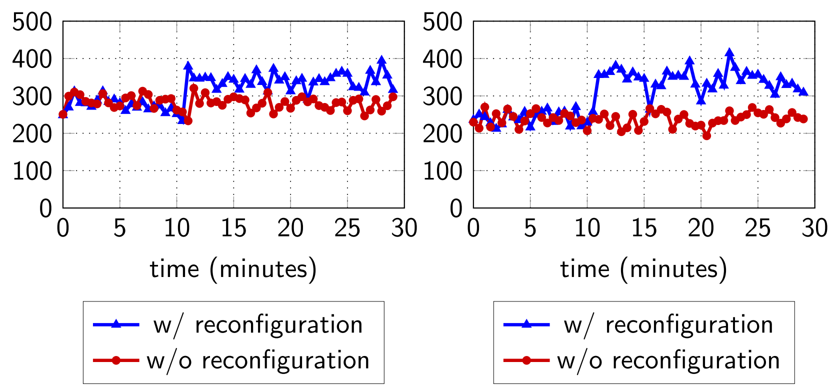
Evaluation

Great speed-up when network is the bottleneck.

Highly dependent on message size.

Evaluation – Flickr

Throughput (Ktuples/s) on 10Gb/s network, parallelism 6

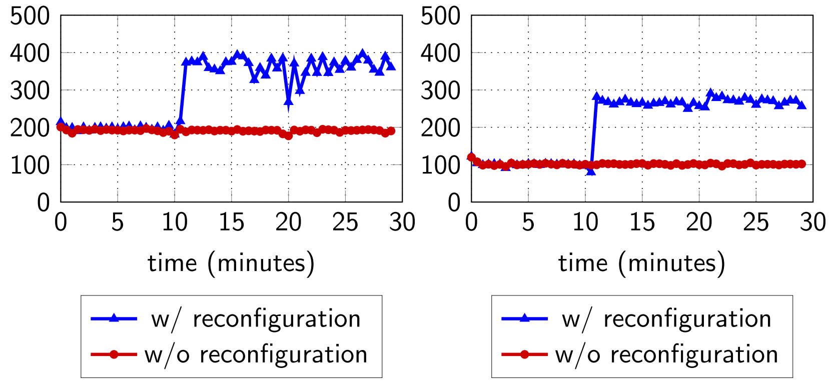


(a) message size=4kB (b) message size=8kB

Locality data routing

Evaluation – Flickr

Throughput (Ktuples/s) on 1Gb/s network, parallelism 6



(a) message size=4kB (b) message size=8kB

Locality data routing

Evaluation – Flickr

Average throughput with 1Gb/s network, 4kB message size

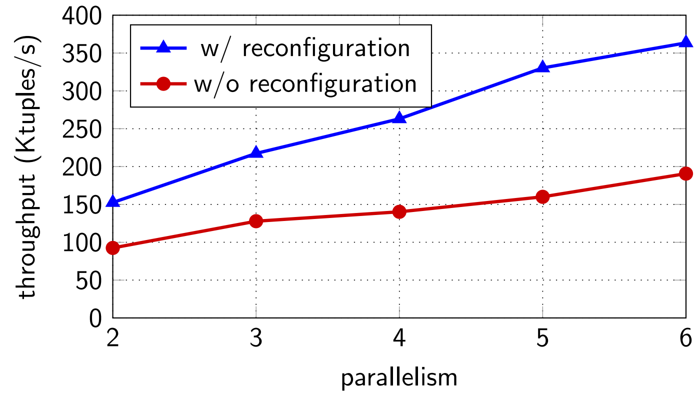
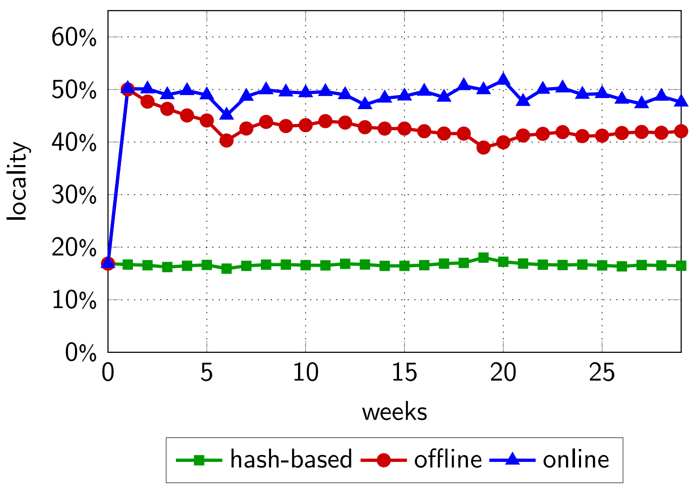


Figure: Average throughput, measured after the first reconfiguration.

Locality data routing

Evaluation – Flickr

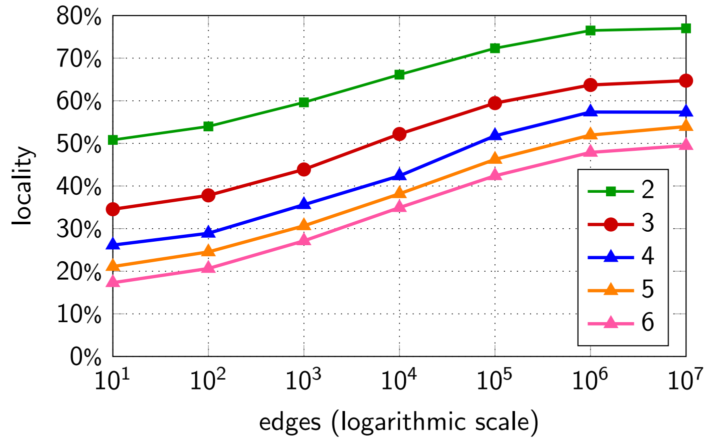
Locality, with parallelism 6



Locality data routing

Evaluation – Flickr

Locality when changing the number of collected key correlations



Locality data routing

Related work

Scheduling: placement of operators on servers

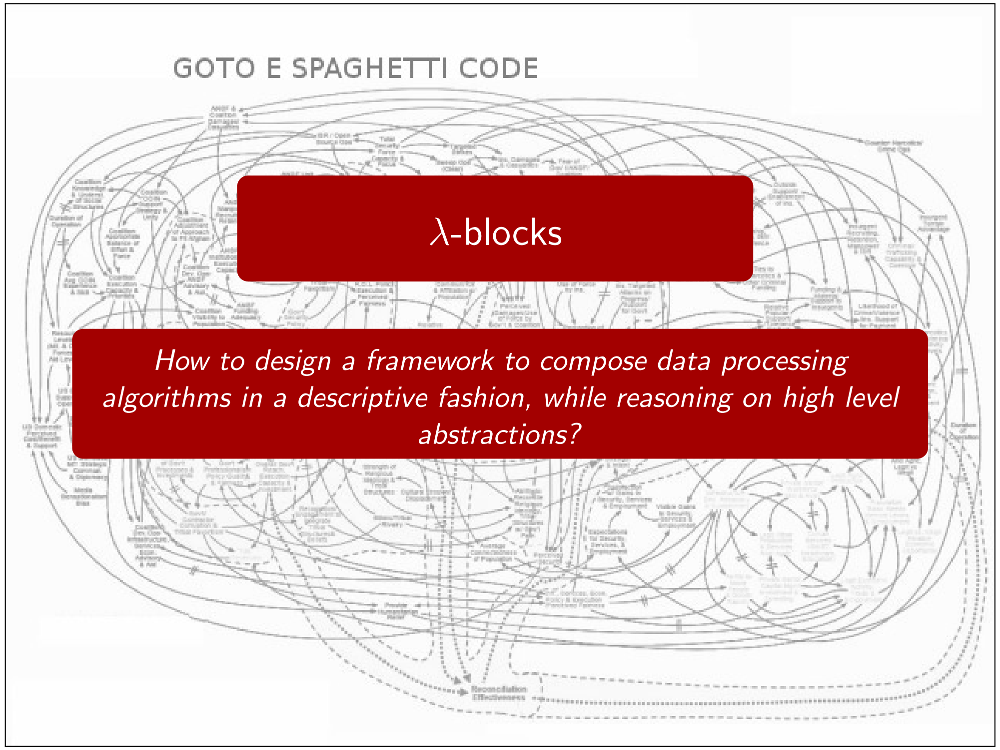
- Using the topology [ABQ13]
- Using observed communication patterns [ABQ13]
- Using observed and/or estimated CPU and memory patterns [FB15, PHH+15]

Load balancing: limit impact of data skew

- Partial key grouping [NMG+15]
- Special routing for frequent keys [RQA+15]

Co-location of correlated keys

- Databases partitions [CJZM10], social networks [BJJL13]



λ-blocks

How to design a framework to compose data processing algorithms in a descriptive fashion, while reasoning on high level abstractions?

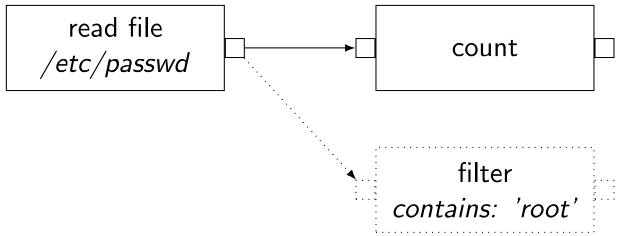
λ-blocks

Design goals

- A data processing abstraction
- A graph of code blocks to represent an end-to-end processing system
- Separation of concerns: low-level data operations, high-level data processing programs
- Maximize reuse of code
- Compatible with existing (specialized) frameworks and possibility to mix them
- Graph manipulation toolkit
- Bring simplicity to large-scale data processing

λ-blocks

Topologies



Topologies

```python
"""Counts system users.
"""

def main():
    with open('/etc/passwd') as f:
        return len(f.readlines())

if __name__ == '__main__':
    print(main())
```

~~$ wc -l /etc/passwd~~

λ-blocks

Topologies

```yaml
---
name: count_users
description: Count number of system users
modules: [lb.blocks.foo]
---
- block: readfile
  name: my_readfile
  args:
    filename: /etc/passwd

- block: count
  name: my_count
  inputs:
    data: my_readfile.result
```


λ-blocks

Blocks

- read_http
- plot_bars
- show_console
- write_line
- write_lines
- split
- concatenate
- map_list
- flatMap
- flatten_list
- group_by_count
- sort
- get_spark_context
- spark_readfile
- spark_text_to_words
- spark_map
- spark_filter
- spark_flatMap
- spark_mapPartitions
- spark_sample
- spark_union
- spark_intersection
- spark_distinct
- spark_groupByKey
- spark_reduceByKey
- spark_aggregateByKey
- spark_sortByKey
- spark_join
- spark_cogroup
- spark_cartesian
- spark_pipe
- spark_coalesce
- spark_repartition
- spark_reduce
- spark_collect
- spark_count
- spark_first
- spark_take
- spark_takeSample
- spark_takeOrdered
- spark_saveAsTextFile
- spark_countByKey
- spark_foreach
- spark_add
- spark_swap
- twitter_search
- cat
- grep
- cut
- head
- tail

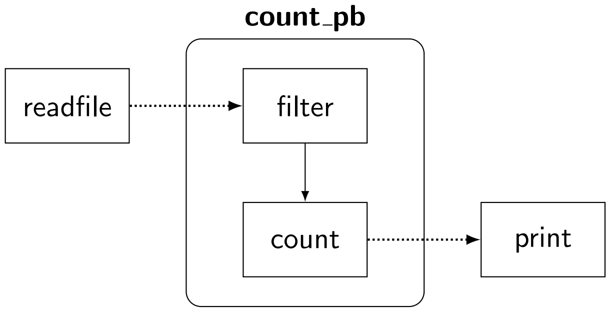
λ-blocks

Blocks

```python
@block(engine='localpython')
def take(n: int=0):
    """Truncates a list of integers.

    :param int n: The length of the desired result.
    :input List[int] data: The list of items to truncate.
    :output List[int] result: The truncated result.
    """
    def inner(data: List[int])->ReturnType[List[int]]:
        assert n <= len(data)
        return ReturnEntry(result=data[:n])
    return inner
```

Sub-topologies



Sub-topologies

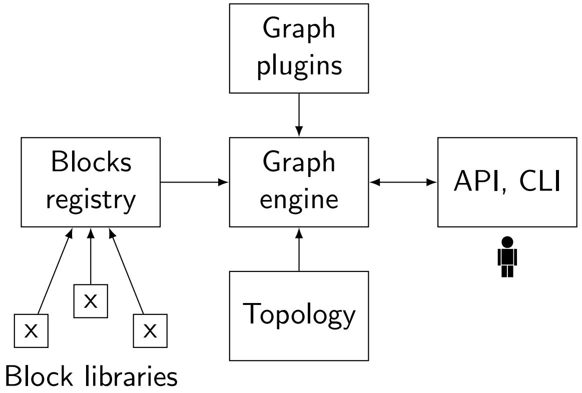
```
---
name: count_pb
---
- block: filter
  name: filter
  args:
    contains: error
  inputs:
    data: $inputs.data
- block: count
  name: count
  inputs:
    data: filter.result
```

```
---
name: foo_errors
---
- block: readfile
  name: readfile
  args:
    filename: foo.log

- topology: count_pb
  name: count_pb
  bind_in:
    data: readfile.result
  bind_out:
    result: count.result

- block: print
  name: print
  inputs:
    data: count_pb.result
```


Architecture



Graph manipulations

- Verification (e.g. type checking)
- Instrumentation
- Caching
- Debugging tools
- Optimizations
- Monitoring
- Program reasoning and semantics

Graph manipulations

- Reasoning on the computation graph as a high-level object
- Plugin system
- Hooks:
  - **before_graph_execution**
    pre-processing, optimizations, verifications
  - **after_graph_execution**
    post-processing
  - **before_block_execution**
    observation, optimizations
  - **after_block_execution**
    observation

Graph manipulation example: instrumentation (excerpt)

```
by_block = {} # timing by block: begin, duration

@before_block_execution
def store_begin_time(block):
    name = block.fields['name']
    by_block[name]['begin'] = time.time()


@after_block_execution
def store_end_time(block, results):
    name = block.fields['name']
    by_block[name]['duration'] = \
      time.time() - by_block[name]['begin']
```


Graph manipulation example: instrumentation (excerpt)

```
@after_graph_execution
def show_times(results):
    longest_first = sorted(by_block, reverse=True)
    for blockname in longest_first:
        print('{}\t{}'.format(
            blockname,
            by_block[blockname]['duration'])
```


Graph manipulation example: instrumentation

block	duration (ms)
read http	818
write lines	54
grep	49
split	20

Evaluation

Setup

- Wordcount over https: local machine, 8 cores, 16 GB RAM
- Wordcount over disk: local machine, 8 cores, 16 GB RAM
- PageRank on Spark: Spark on 1 server (24 cores, 128 GB RAM)

Evaluation

Performances

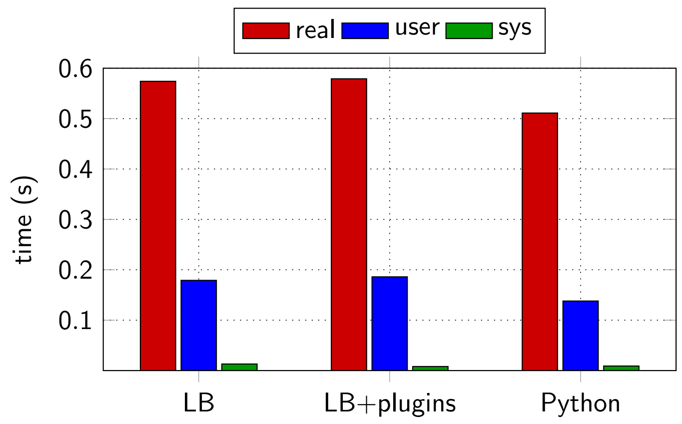


Figure: Wordcount over https: Twitter feed.

Evaluation

Performances

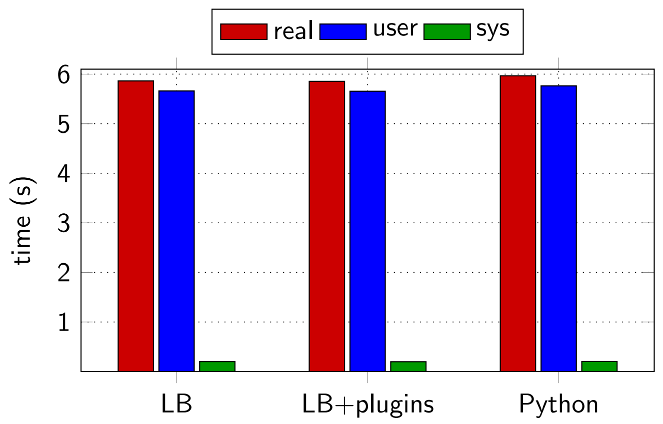


Figure: Wordcount over disk: Wikipedia dataset.

Evaluation

Performances

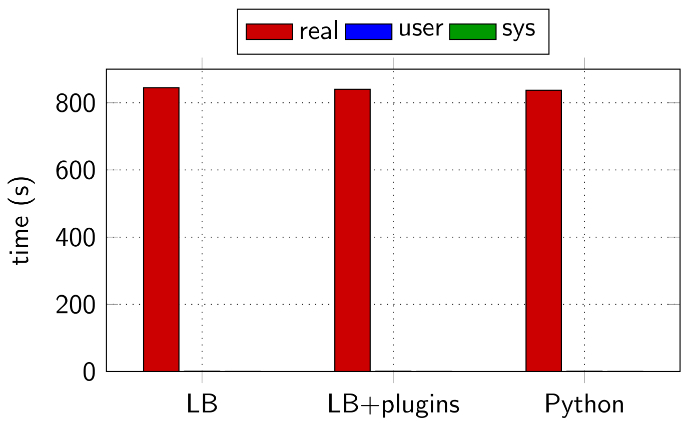


Figure: PageRank on Wikipedia hyperlinks with Spark.

Evaluation

Maximum overhead measured per topology: 50 ms

Related work

Dataflow programming

- ML pipelines: scikit-learn [PVG+11], Spark [The17a], Orange framework [DCE+13]
- Real-time: Apache Beam [apa], StreamPipes [RKHS15]

Blocks programming

- Recognition over recall, immediate feedback [BGK+17]

Graphs from configuration

- Pyleus [Yel16], Storm Flux [The17b]

Other

- "Serverless" architectures and stateless functions [JVSR17]

Conclusion

Context

Computer systems to process large quantities of data.

Problems: how to design...

- An industrial system to handle monitoring data and make predictions about future failures?
- An algorithm to improve locality in distributed streaming engines?
- A framework to compose data processing algorithms in a descriptive fashion, while reasoning on high level abstractions?

Conclusion

Contributions

	Metrics prediction	**Locality routing**	**λ-blocks**
What it is	Industrial system	Online routing library	Data processing abstraction
Layer	End-to-end	Low	High
Improves	**Uptimes**	**Throughput**	**Programmability**

Future work

Metrics prediction in monitoring systems

- Predictions on long-term global trends
- Ticketing mechanism

Locality data routing

- Replace binary locality/non-locality with distance
- Smarter way to determine when to reschedule
- Extend to more complex topologies

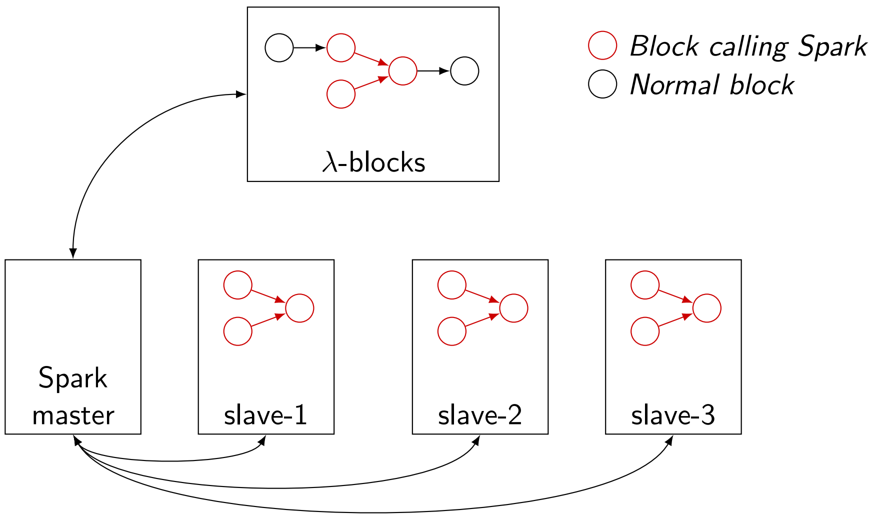
Future work

λ-blocks

- Explore more graph manipulation abstractions (complexity analysis, serialization, verification. . .)
- Streaming and online operations
- Tight integration with clusters (data storage, caches, etc)

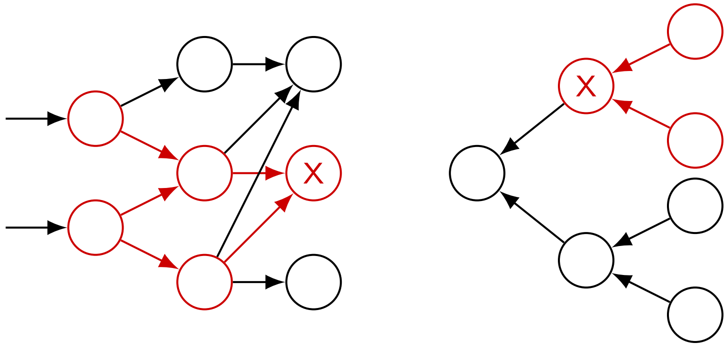
Thanks! Questions?

Using a Spark cluster



λ-blocks

Signature algorithm



$$H(B) = h(B.name, \quad \text{block name (not instance name)}$$
$$B.args, \quad \text{list of (name, value) tuples}$$
$$B.inputs) \quad \text{list of (name, H(block), connector) tuples}$$

λ-blocks

Evaluation

Engine instrumentation

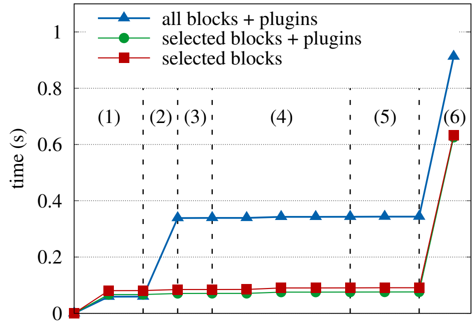


Figure: Wordcount program running under different setups.
(1) Startup (modules import, etc); (2) Blocks registry creation, block modules import; (3) Plugin import; (4) YAML parsing and graph creation; (5) Graph checks; (6) Graph execution.

Metrics prediction in monitoring systems

Database schema

metrics

metric_id	uuid
metric_name	text
group_id	uuid

predictions

metric_id	uuid
timestamp	int
predicted_values	list

measurements

metric_id	uuid
timestamp	int
warn	text
crit	text
max	double
min	double
value	double
metric_name	text
metric_unit	text

Images credits

- *Data Center operators verifying network cable integrity*, CC-BY-SA, `https://commons.wikimedia.org/wiki/File:Dc_cabling_50.jpg`
- *Tokyo metro map*, `http://bento.com/subtop5.html`
- *Goto e spaghetti code*, `http://blogbv2.altervista.org/HD/il-goto-e-la-buona-programmazione-parte-ii/`

Bibliography I

Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni.
Adaptive online scheduling in storm.
In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 207–218. ACM, 2013.

Apache Beam.
`https://beam.apache.org/`.

David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak.
Learnable programming: Blocks and beyond.
Commun. ACM, 60(6):72–80, May 2017.

Bibliography II

Xiao Bai, Arnaud Jégou, Flavio Junqueira, and Vincent Leroy.
Dynasore: Efficient in-memory store for social applications.
In *Middleware 2013 - ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings*, pages 425–444, 2013.

T. Chalermarrewong, T. Achalakul, and S. C. W. See.
Failure prediction of data centers using time series and fault tree analysis.
In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 794–799, Dec 2012.

Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden.
Schism: A workload-driven approach to database replication and partitioning.
Proc. VLDB Endow., 3(1-2):48–57, September 2010.

Bibliography III

Janez Demšar, Tomaž Curk, Aleš Erjavec, Črt Gorup, Tomaž Hočevar, Mitar Milutinovič, Martin Možina, Matija Polajnar, Marko Toplak, Anže Starič, Miha Štajdohar, Lan Umek, Lan Žagar, Jure Žbontar, Marinka Žitnik, and Blaž Zupan.
Orange: Data mining toolbox in python.
Journal of Machine Learning Research, 14:2349–2353, 2013.

Lorenz Fischer and Abraham Bernstein.
Workload scheduling in distributed stream processors using graph partitioning.
In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 124–133, 2015.

Bibliography IV

Eric Jonas, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht.
Occupy the cloud: Distributed computing for the 99%.
arXiv preprint arXiv:1702.04024, 2017.

Lei Li, Chieh-Jan Mike Liang, Jie Liu, Suman Nath, Andreas Terzis, and Christos Faloutsos.
Thermocast: A cyber-physical forecasting model for datacenters.
In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1370–1378, New York, NY, USA, 2011. ACM.

Bibliography V

Microsoft cloud azure.
https://docs.microsoft.com/en-us/azure/machine-learning/machine-learning-algorithm-choice.

Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini.
The power of both choices: Practical load balancing for distributed stream processing engines.
In *31st IEEE International Conference on Data Engineering, ICDE*, pages 137–148, 2015.

Bibliography VI

Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell.
R-storm: Resource-aware scheduling in storm.
In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 149–161, New York, NY, USA, 2015. ACM.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay.
Scikit-learn: Machine learning in Python.
Journal of Machine Learning Research, 12:2825–2830, 2011.

Bibliography VII

Dominik Riemer, Florian Kaulfersch, Robin Hutmacher, and Ljiljana Stojanovic.
Streampipes: solving the challenge with semantic stream processing pipelines.
In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 330–331. ACM, 2015.

Nicoló Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola.
Efficient key grouping for near-optimal load balancing in stream processing systems.
In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 80–91, New York, NY, USA, 2015. ACM.

Bibliography VIII

The Apache Spark developers.
ML Pipelines.
https://spark.apache.org/docs/latest/ml-pipeline.html, 2017.

The Apache Storm developers.
Flux.
http://storm.apache.org/releases/2.0.0-SNAPSHOT/flux.html, 2017.

YelpArchive.
Pyleus.
https://github.com/YelpArchive/pyleus, 2016.

Zabbix prediction triggers.
https://www.zabbix.com/documentation/3.0/manual/config/triggers/prediction.