Introduction to Data Streaming

Vincenzo Gulisano, Ph.D.

CHALMERS
UNIVERSITY OF TECHNOLOGY

Agenda

- Motivation
- The data streaming processing paradigm
- Challenges and research questions
- Conclusions
- Bibliography

Agenda

- Motivation
- The data streaming processing paradigm
- Challenges and research questions
- Conclusions
- Bibliography

IoT enables for increased awareness, security, power-efficiency, ...

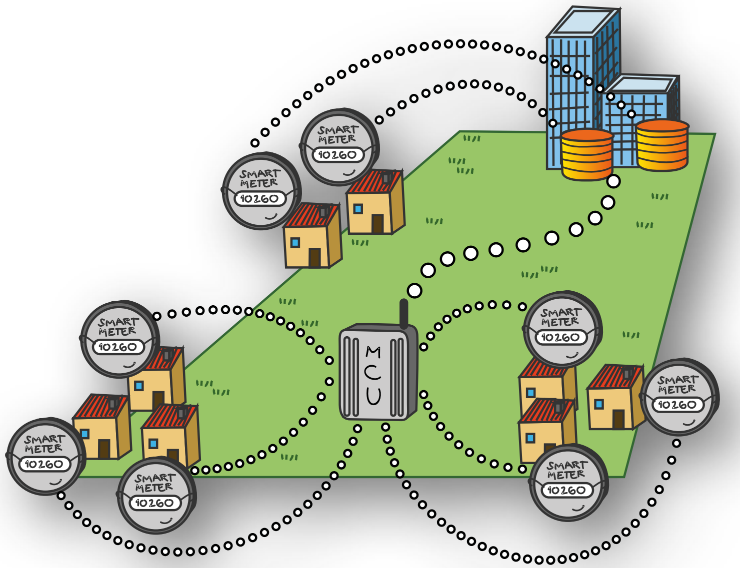
BUT

large IoT systems are complex

WHICH IMPLIES
(AMONG OTHER THINGS)

traditional data analysis techniques alone are not adequate!

IoT enables for increased awareness, security, power-efficiency, ...



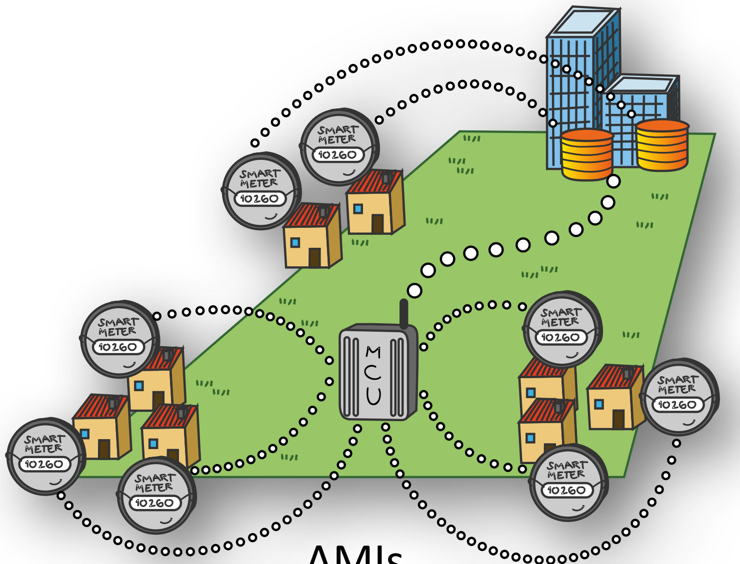
AMIs [1,2,3,4]

- demand-response
- scheduling [7]
- micro-grids
- detection of medium size blackouts [8]
- detection of non technical losses
- ...

VNs [5,6]

- autonomous driving
- platooning
- accident detection [9]
- variable tolls [9]
- congestion monitoring [10]
- ...

large IoT systems are complex



AMIs

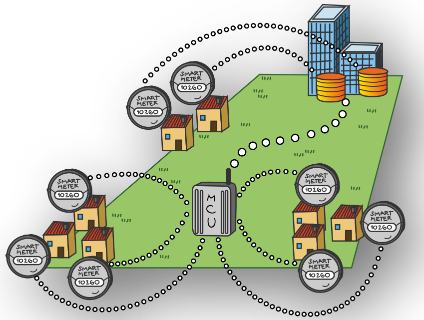
VNs

Characteristics [15]:

1. edge location
2. location awareness
3. low latency
4. geographical distribution
5. large-scale
6. support for mobility
7. real-time interactions
8. predominance of wireless
9. heterogeneous
10. interoperability / federation
11. interaction with the cloud

WHICH IMPLIES
(AMONG OTHER THINGS)

traditional data analysis techniques alone are not adequate! [13,14]



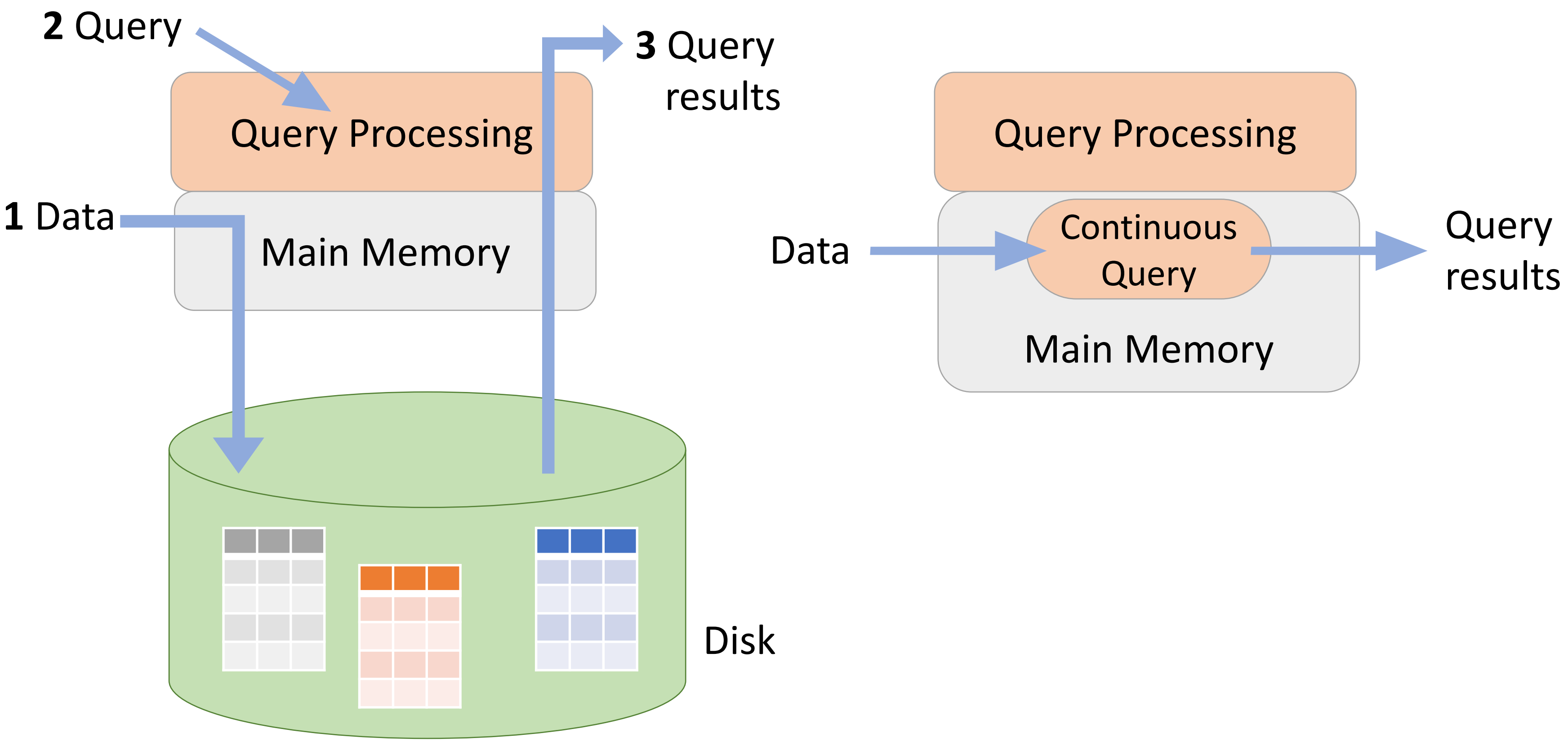
1. does the infrastructure allow for billions of readings per day to be transferred continuously?
2. the latency incurred while transferring data, does that undermine the utility of the analysis?
3. is it secure to concentrate all the data in a single place? [11]
4. is it smart to give away fine-grained data? [12]

Agenda

- Motivation
- The data streaming processing paradigm
- Challenges and research questions
- Conclusions
- Bibliography

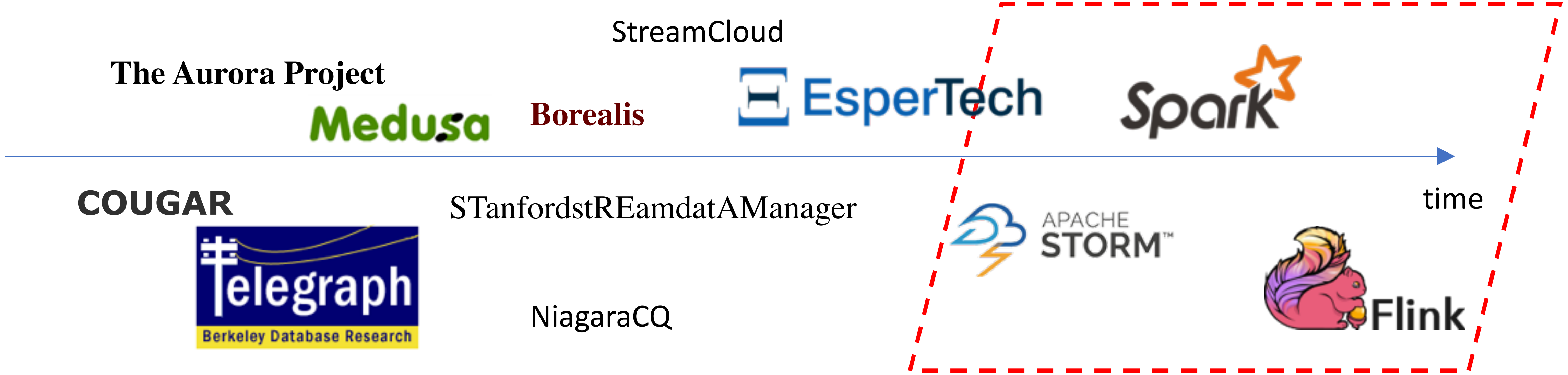
Motivation

DBMS vs. DSMS



Before we start... about data streaming and Stream Processing Engines (SPEs)

An incomplete list of SPEs (cf. related work in [16]):



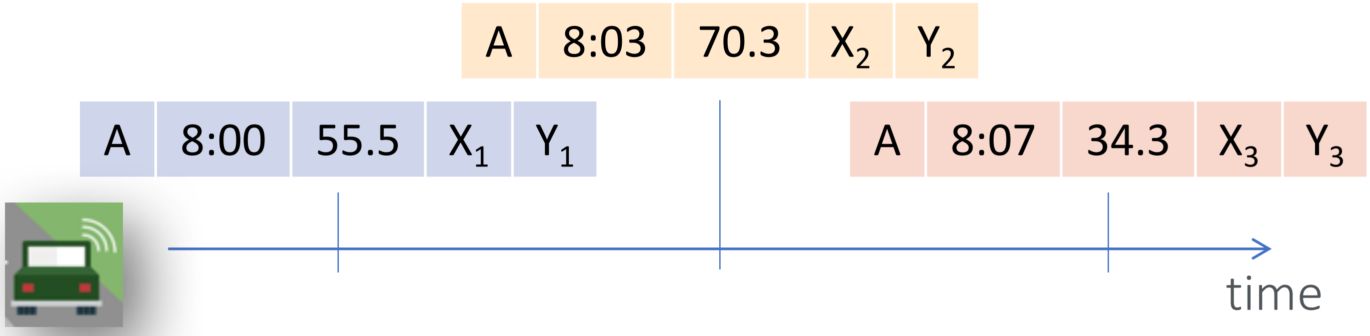
Covering all of them / discussing which use cases are best for each one out of scope...
the following show connection between what is being presented and a certain SPE

data stream: unbounded sequence of tuples sharing the same schema

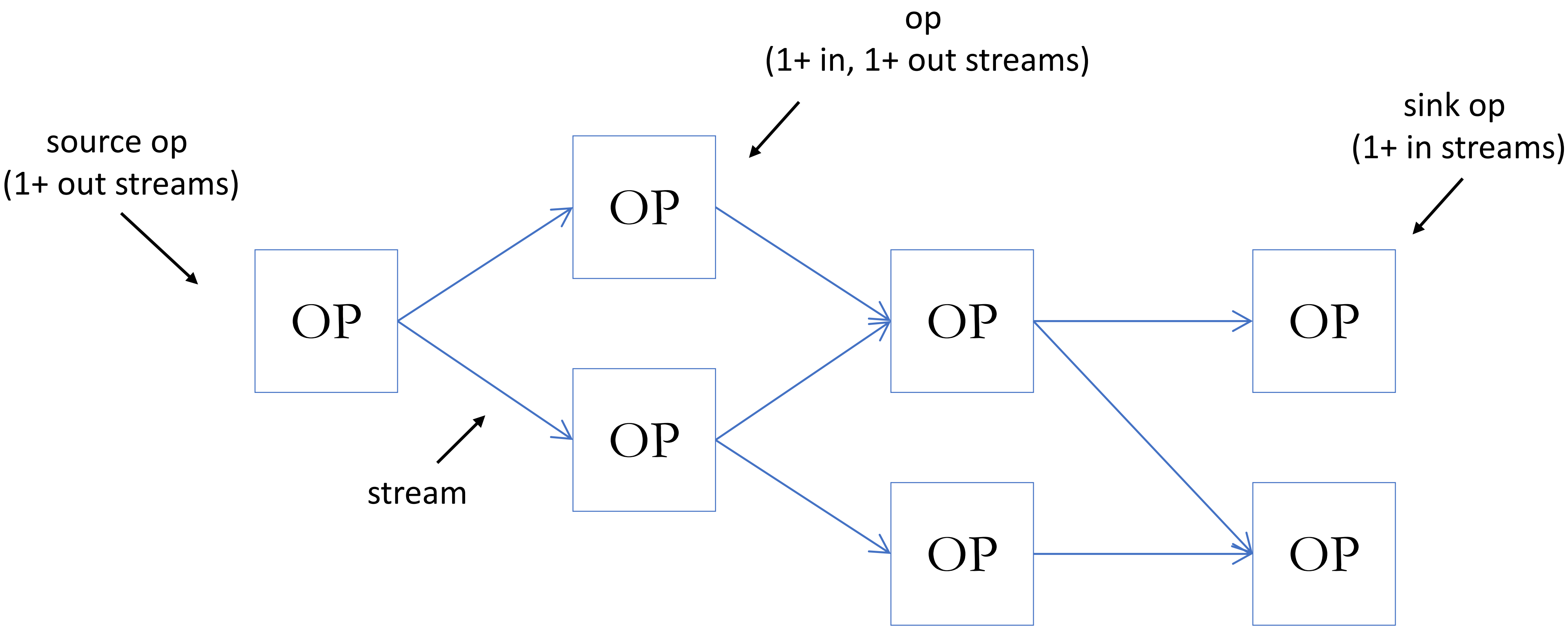
Example: vehicles' speed reports

Field	Field
vehicle id	text
time (secs)	text
speed (Km/h)	double
X coordinate	double
Y coordinate	double

Let's assume each source (e.g., vehicle) produces and delivers a timestamp sorted stream



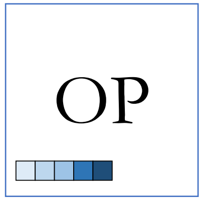
continuous query (or simply **query**): Directed Acyclic Graph (DAG) of streams and operators



data streaming **operators**

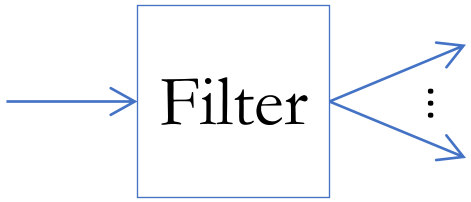
Two main types:

- Stateless operators
  - do not maintain any state
  - one-by-one processing
  - if they maintain some state, such state does not evolve depending on the tuples being processed

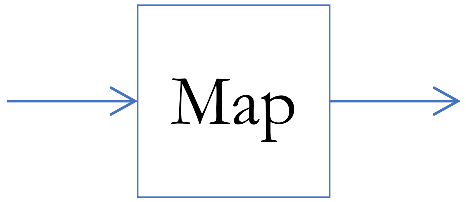


- Stateful operators
  - maintain a state that evolves depending on the tuples being processed
  - produce output tuples that depend on multiple input tuples

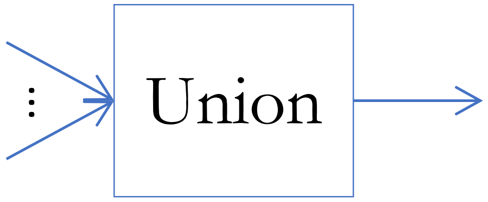
stateless operators



Filter / route tuples based on one (or more) conditions

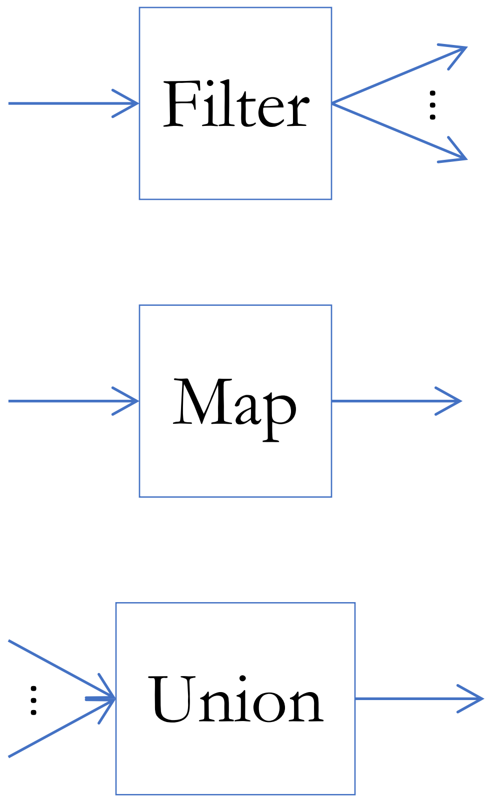


Transform each tuple



Merge multiple streams (with the same schema) into one

stateless operators



Consider this example. Suppose you have a stream called "stream" that contains the fields "x", "y", and "z". To run a filter MyFilter that takes in "y" as input, you would say:

```
stream.each(new Fields("y"), new MyFilter())
```

Suppose the implementation of MyFilter is this:

```
public class MyFilter extends BaseFilter {
   public boolean isKeep(TridentTuple tuple) {
       return tuple.getInteger(0) < 10;
   }
}
```

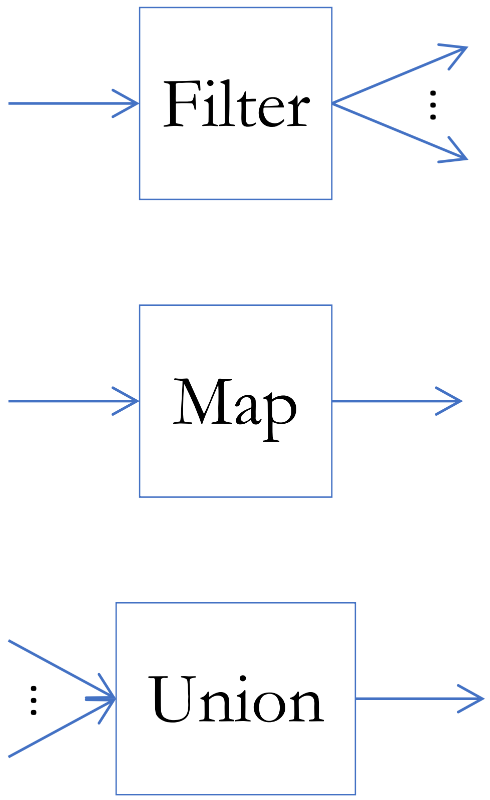
This will keep all tuples whose "y" field is less than 10. The TridentTuple given as input to MyFilter will only contain the "y" field. Note that Trident is able to project a subset of a tuple extremely efficiently when selecting the input fields: the projection is essentially free.

Let's now look at how "function fields" work. Suppose you had this function:

```
public class AddAndMultiply extends BaseFunction {
   public void execute(TridentTuple tuple, TridentCollector collector) {
       int i1 = tuple.getInteger(0);
       int i2 = tuple.getInteger(1);
       collector.emit(new Values(i1 + i2, i1 * i2));
   }
}
```

source: http://storm.apache.org/releases/2.0.0-SNAPSHOT/Trident-tutorial.html

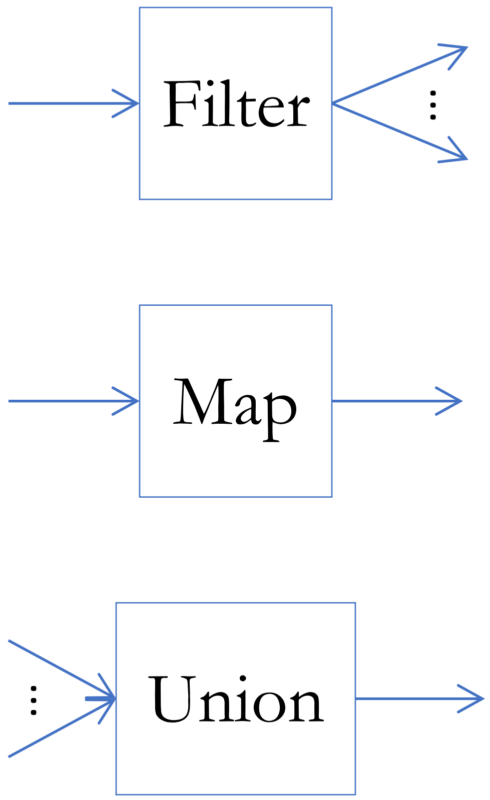
stateless operators



<table>
<tr><th>Transformation</th><th>Description</th></tr>
<tr><td>Map
DataStream → DataStream</td><td>Takes one element and produces one element. A map function that doubles the values of the input stream:
<pre>
DataStream<Integer> dataStream = //...
dataStream.map(new MapFunction<Integer, Integer>() {
    @Override
    public Integer map(Integer value) throws Exception {
        return 2 * value;
    }
});
</pre></td></tr>
<tr><td>FlatMap
DataStream → DataStream</td><td>Takes one element and produces zero, one, or more elements. A flatmap function that splits sentences to words:
<pre>
dataStream.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public void flatMap(String value, Collector<String> out)
        throws Exception {
        for(String word: value.split(" ")){
            out.collect(word);
        }
    }
});
</pre></td></tr>
<tr><td>Filter
DataStream → DataStream</td><td>Evaluates a boolean function for each element and retains those for which the function returns true. A filter that filters out zero values:
<pre>
dataStream.filter(new FilterFunction<Integer>() {
    @Override
    public boolean filter(Integer value) throws Exception {
        return value != 0;
    }
});
</pre></td></tr>
</table>

source: https://ci.apache.org/projects/flink/flink-docs-release-1.0/apis/streaming/index.html

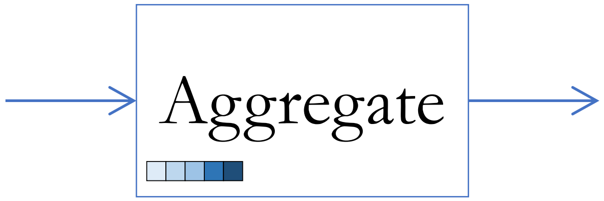
stateless operators



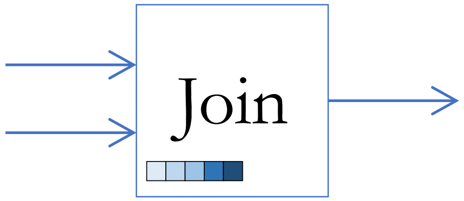
Transformation	Meaning
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.

source: http://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams

stateful operators



Aggregate information from multiple tuples
(e.g., max, min, sum, ...)



Join tuples coming from 2 streams given a certain predicate

stateful operators

```
stream.aggregate(new Fields("val2"), new Sum(), new Fields("sum"))
```

The output stream would only contain a single tuple with a single field called "sum", representing the sum of all "val2" fields in that batch.

With grouped streams, the output will contain the grouping fields followed by the fields emitted by the aggregator. For example:

```
stream.groupBy(new Fields("val1"))
     .aggregate(new Fields("val2"), new Sum(), new Fields("sum"))
```

In this example, the output will contain the fields "val1" and "sum".

source: http://storm.apache.org/releases/2.0.0-SNAPSHOT/Trident-tutorial.html

count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative so that it can be computed in parallel.

source: http://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams

Aggregations KeyedStream → DataStream	Rolling aggregations on a keyed data stream. The difference between min and minBy is that min returns the minimun value, whereas minBy returns the element that has the minimum value in this field (same for max and maxBy).

```
keyedStream.sum(0);
keyedStream.sum("key");
keyedStream.min(0);
keyedStream.min("key");
keyedStream.max(0);
keyedStream.max("key");
keyedStream.minBy(0);
keyedStream.minBy("key");
keyedStream.maxBy(0);
keyedStream.maxBy("key");
```

source: http://spark.apache.org/docs/latest/streaming-programming-guide.html#transformations-on-dstreams

Wait a moment!

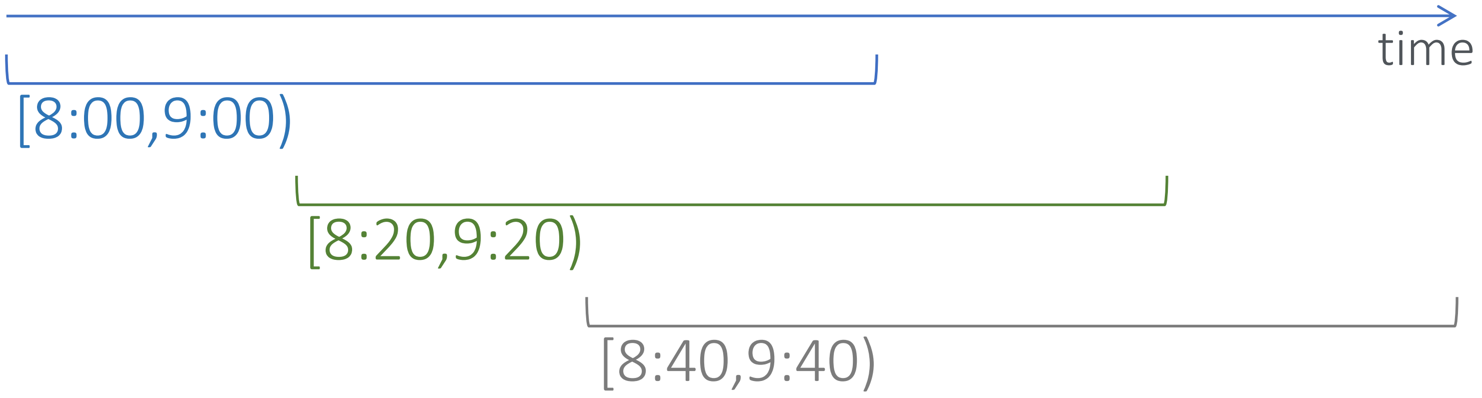
if streams are unbounded, how can we aggregate or join?

windows and **stateful analysis** [16]

Stateful operations are done over windows:

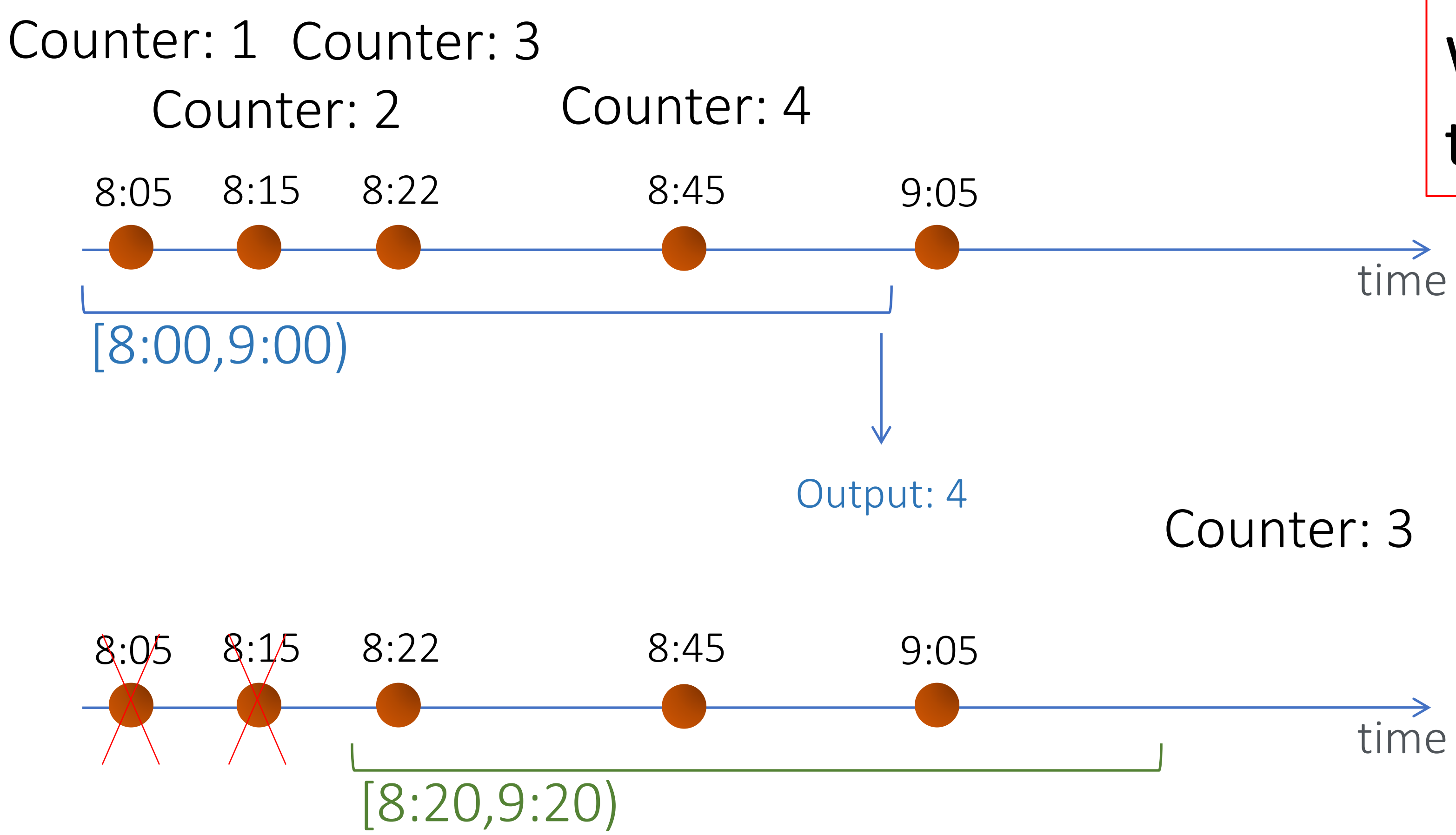
- Time-based (e.g., tuples in the last 10 minutes)
- Tuple-based (e.g., given the last 50 tuples)

Usually applications rely on time-based sliding windows



time-based sliding window aggregation (count)

we assumed each source produces and delivers a timestamp sorted stream! What happens if this is not the case?



windows and stateful analysis

```
// Reduce last 30 seconds of data, every 10 seconds
val windowedWordCounts = pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b), Seconds(30), Seconds(10))
```

Some of the common window operations are as follows. All of these operations take the said two parameters - *windowLength* and *slideInterval*.

Transformation	Meaning
window(*windowLength*, *slideInterval*)	Return a new DStream which is computed based on windowed batches of the source DStream.
countByWindow(*windowLength*, *slideInterval*)	Return a sliding window count of elements in the stream.
reduceByWindow(*func*, *windowLength*, *slideInterval*)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*. The function should be associative so that it can be computed correctly in parallel.

Transformation	Description
Tumbling time window KeyedStream → WindowedStream	Defines a window of 5 seconds, that "tumbles". This means that elements are grouped according to their timestamp in groups of 5 second duration, and every element belongs to exactly one window. The notion of time is specified by the selected TimeCharacteristic (see time). `keyedStream.timeWindow(Time.seconds(5));`
Sliding time window KeyedStream → WindowedStream	Defines a window of 5 seconds, that "slides" by 1 seconds. This means that elements are grouped according to their timestamp in groups of 5 second duration, and elements can belong to more than one window (since windows overlap by at most 4 seconds) The notion of time is specified by the selected TimeCharacteristic (see time). `keyedStream.timeWindow(Time.seconds(5), Time.seconds(1));`
Tumbling count window KeyedStream → WindowedStream	Defines a window of 1000 elements, that "tumbles". This means that elements are grouped according to their arrival time (equivalent to processing time) in groups of 1000 elements, and every element belongs to exactly one window. `keyedStream.countWindow(1000);`
Sliding count window KeyedStream → WindowedStream	Defines a window of 1000 elements, that "slides" every 100 elements. This means that elements are grouped according to their arrival time (equivalent to processing time) in groups of 1000 elements, and every element can belong to more than one window (as windows overlap by at most 900 elements). `keyedStream.countWindow(1000, 100)`

basic operators and **user-defined operators**

Besides a set of basic operators, SPEs usually allow the user to define ad-hoc operators (e.g., when existing aggregation are not enough)

Flink

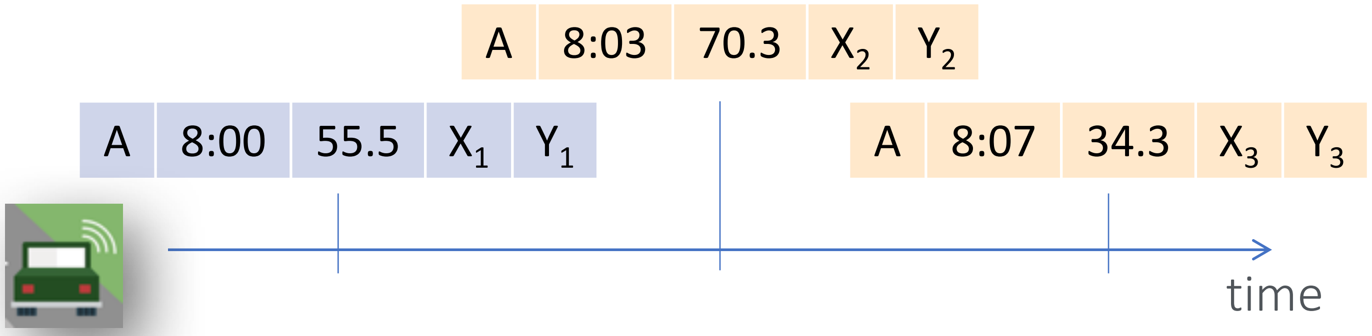
Aggregations KeyedStream → DataStream	Rolling aggregations on a keyed data stream. The difference between min and minBy is that min returns the minimun value, whereas minBy returns the element that has the minimum value in this field (same for max and maxBy).

```
keyedStream.sum(0);
keyedStream.sum("key");
keyedStream.min(0);
keyedStream.min("key");
keyedStream.max(0);
keyedStream.max("key");
keyedStream.minBy(0);
keyedStream.minBy("key");
keyedStream.maxBy(0);
keyedStream.maxBy("key");
```

Window Apply WindowedStream → DataStream AllWindowedStream → DataStream	Applies a general function to the window as a whole. Below is a function that manually sums the elements of a window. **Note:** If you are using a windowAll transformation, you need to use an AllWindowFunction instead.

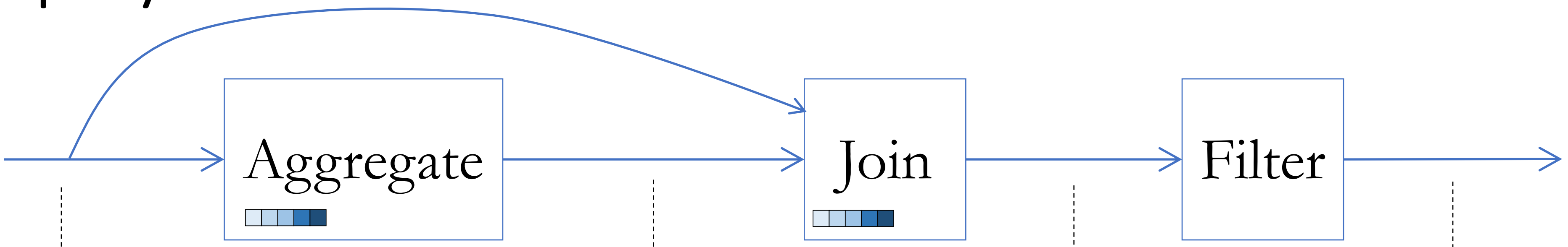
```
windowedStream.apply (new WindowFunction<Tuple2<String,Integer>, Integer, Tuple, W
indow>() {
    public void apply (Tuple tuple,
            Window window,
            Iterable<Tuple2<String, Integer>> values,
            Collector<Integer> out) throws Exception {
        int sum = 0;
        for (value t: values) {
            sum += t.f1;
        }
        out.collect (new Integer(sum));
    }
});

// applying an AllWindowFunction on non-keyed window stream
allWindowedStream.apply (new AllWindowFunction<Tuple2<String,Integer>, Integer, Wi
ndow>() {
    public void apply (Window window,
            Iterable<Tuple2<String, Integer>> values,
            Collector<Integer> out) throws Exception {
        int sum = 0;
        for (value t: values) {
            sum += t.f1;
        }
        out.collect (new Integer(sum));
    }
});
```

sample query



For each vehicle, raise an alert if the speed of the latest report is more than 2 times higher than its average speed in the last 30 days.

sample query



Compute average speed for each vehicle during the last 30 days

Join on vehicle id

Check condition

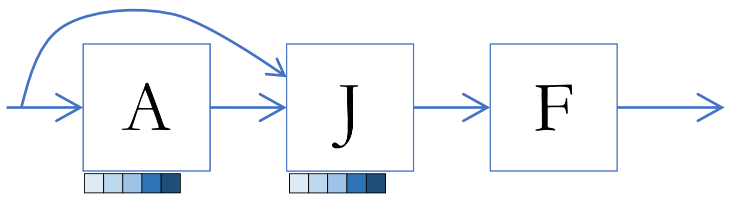
Field
vehicle id
time (secs)
speed (Km/h)
X coordinate
Y coordinate

Field
vehicle id
time (secs)
avg speed (Km/h)

Field
vehicle id
time (secs)
avg speed (Km/h)
speed (Km/h)

Field
vehicle id
time (secs)
speed (Km/h)

sample query



Notice:

- the same semantics can be defined in several ways (using different operators and composing them in different ways)
- Using many basic building blocks can ease the task of distributing and parallelizing the analysis (more in the following...)

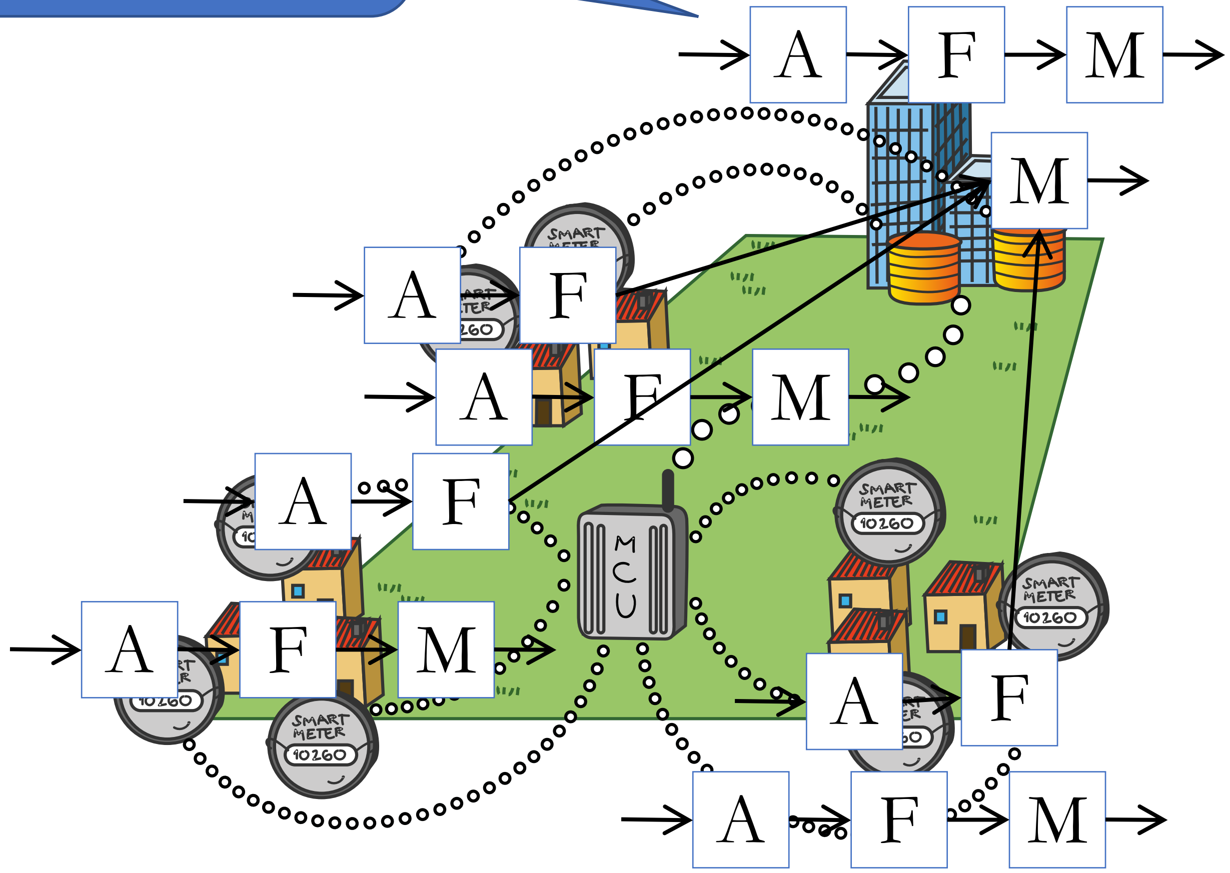
Why data streaming, then?

Expressive

Online

Parallel & Distributed

Sample query that e.g. validates data / raises alarms...



Expressive

Online

Parallel & Distributed

Agenda

- Motivation
- The data streaming processing paradigm
- Challenges and research questions
- Conclusions
- Bibliography

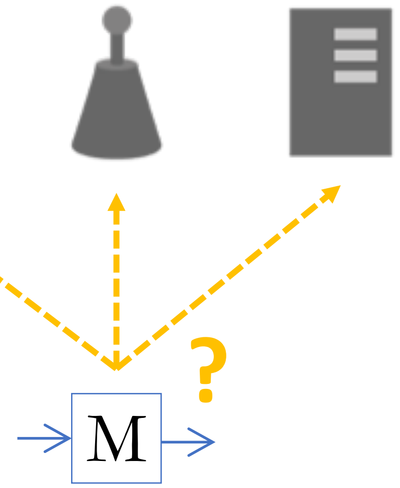
Challenges and research questions

1. Distributed deployment
2. Parallel deployment
3. Ordering and determinism
4. Shared-nothing vs shared-memory parallelism
5. Load balancing
6. Elasticity
7. Fault tolerance
8. Data sharing

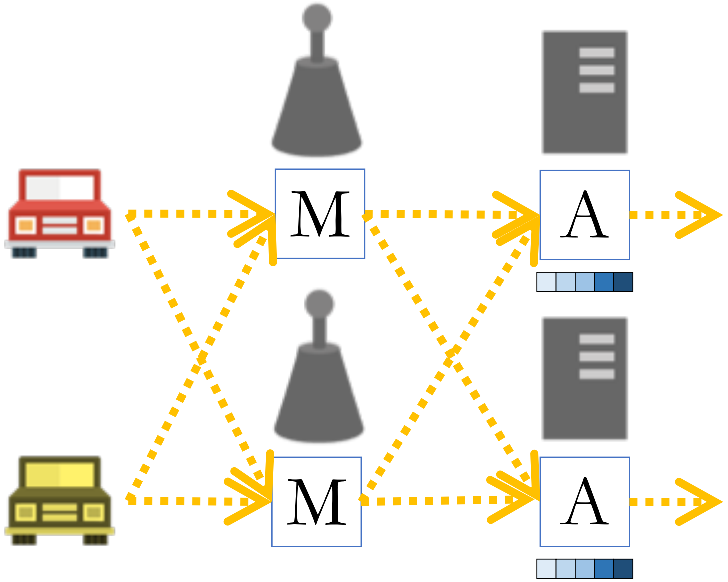
Before we start…

Following examples are from vehicular networks

1 - Distributed deployment – where to place a given operator? [17,4,18,19]

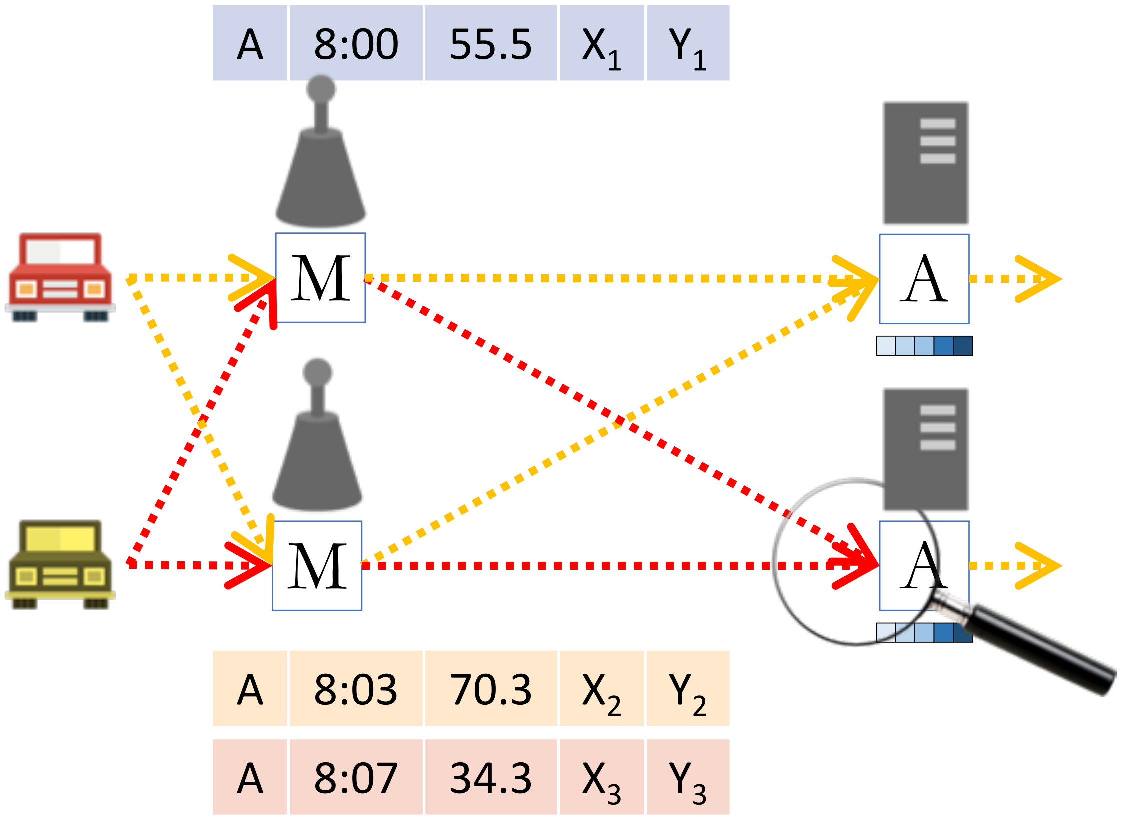


2 - Parallel deployment – how do we parallelize the analysis? [20,21]



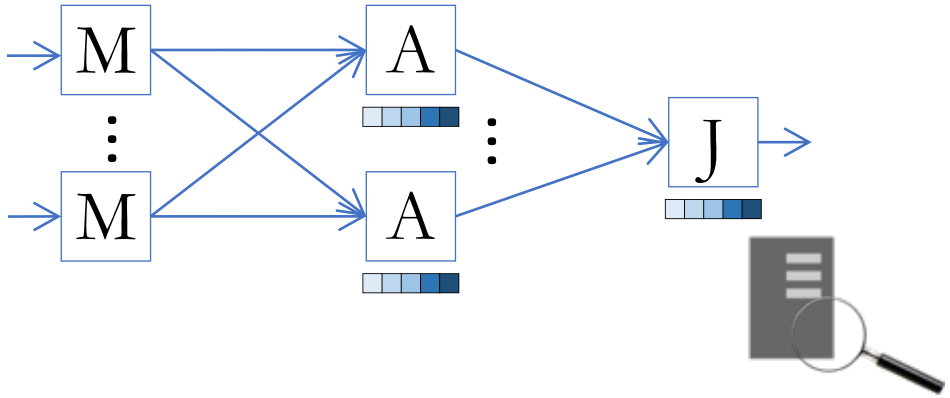
3 – Ordering and determinism [22,23,24]

A	8:00	55.5	X_1	Y_1
A	8:03	70.3	X_2	Y_2
A	8:07	34.3	X_3	Y_3



What if tuple with timestamp 8:00 arrives after tuple with timestamp 8:07?

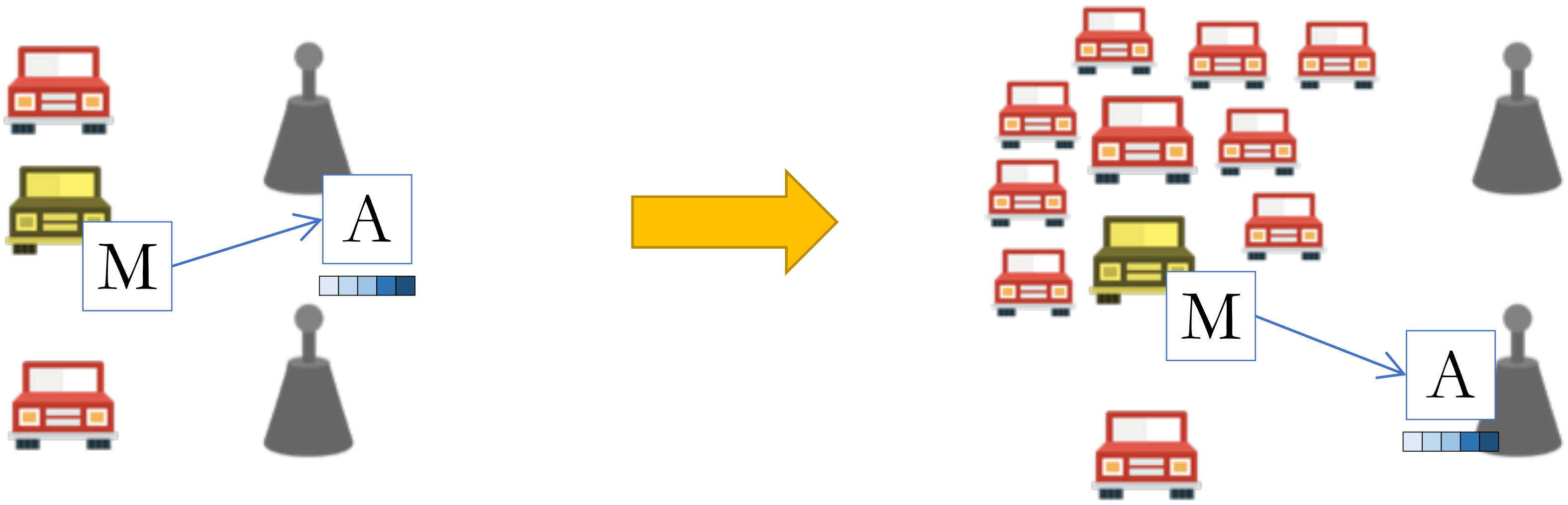
4 – shared-nothing vs. shared-memory parallelism [25]



How to take advantage of multi-core architectures?

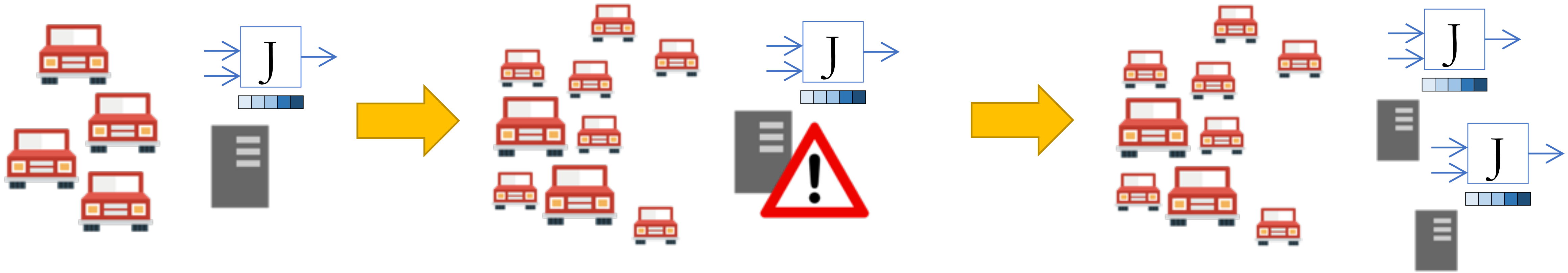
How to boost inter-node parallelism and intra-node parallelism?

5 – load balancing & state transfer [20,26]



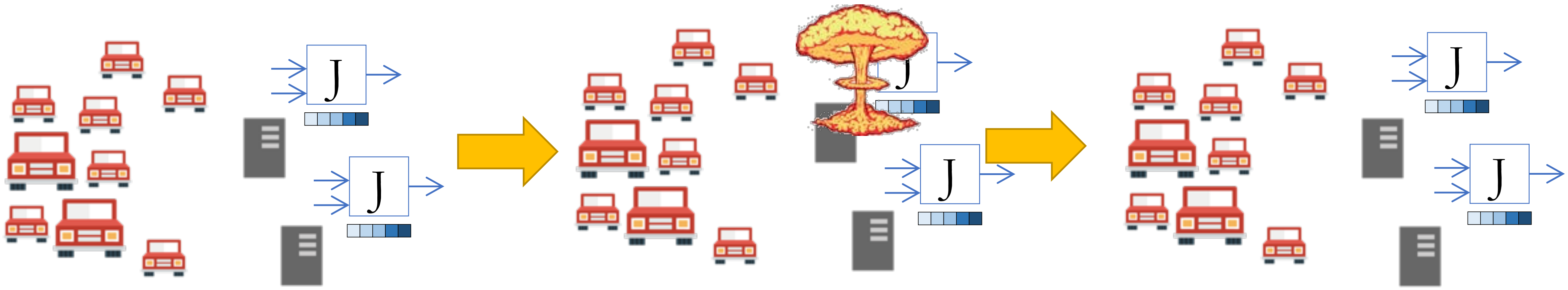
If we shift the processing of a certain subset of tuples from node A to node B, how do transfer its previous state?

6 – elasticity [20,27]



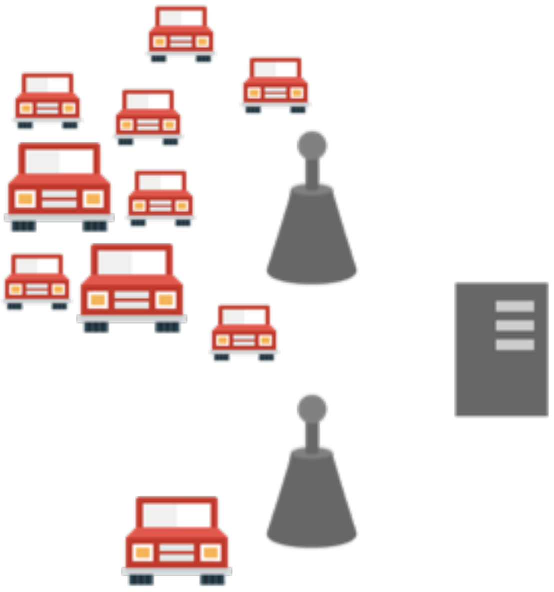
How / when to provision or decommission new resources depending on the analysis' costs fluctuations?

7 – fault tolerance [16, 28, 29]



How to replace a failed node minimizing recovery time (making it transparent to the end user)?

8 – data sharing (differential privacy) [2,30,31,32]

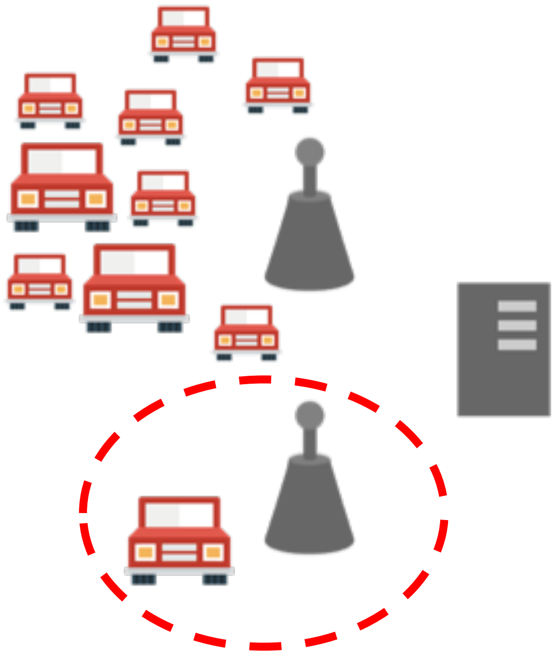


Suppose we are interested in publishing vehicles' average speed over a window of one hour...

How to prevent privacy leaks?

We could aggregate by RSU!

8 – data sharing (differential privacy)



Wait a moment!
what if a single vehicle is connected to a certain RSU?

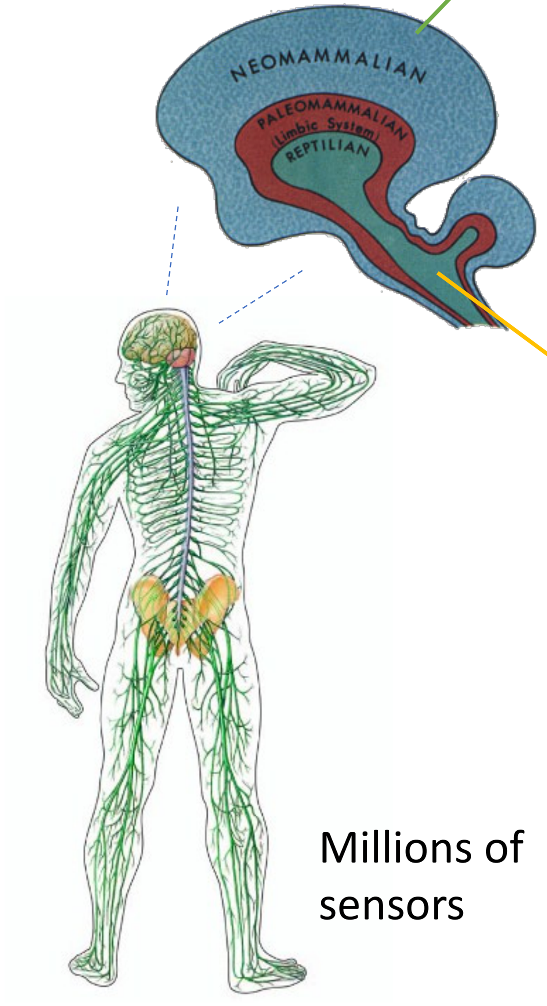
Whether a certain mechanism preserves or not the privacy of the underlying data depends on the knowledge of the adversary

Differential privacy assumes the worst case scenario!

Agenda

- Motivation
- The data streaming processing paradigm
- Challenges and research questions
- Conclusions
- Bibliography

Humans



- Store information
- Iterate multiple times over data
- Think, do not rush through decisions

Should I (really) have an extra piece of cake?

- ”Hard-wired” routines
- Real-time decisions
- High-throughput / low-latency

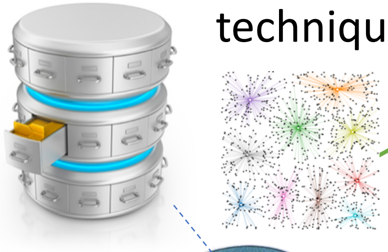
Danger!!!
Run!!!

Millions of years of evolution

Millions of sensors

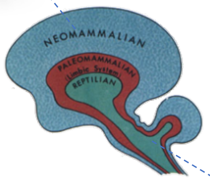
Computers (cyber-physical / IoT systems)

Databases, data mining techniques...



- Store information
- Iterate multiple times over data
- Think, do not rush through decisions

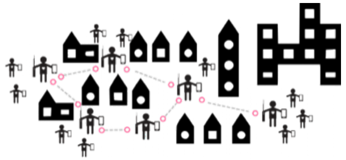
What traffic congestion patterns can I observe frequently?



Data streaming, distributed and parallel analysis

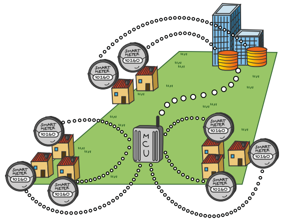
Spark

Flink

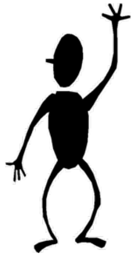


- Continuous analysis
- Real-time decisions
- High-throughput / low-latency

Years / Decades of evolution



Millions of sensors



Don’t take over, car in opposite lane!

Agenda

- Motivation
- The data streaming processing paradigm
- Challenges and research questions
- Conclusions
- Bibliography

Bibliography

1. Zhou, Jiazhen, Rose Qingyang Hu, and Yi Qian. "Scalable distributed communication architectures to support advanced metering infrastructure in smart grid." IEEE Transactions on Parallel and Distributed Systems 23.9 (2012): 1632-1642.
2. Gulisano, Vincenzo, et al. "BES: Differentially Private and Distributed Event Aggregation in Advanced Metering Infrastructures." Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security. ACM, 2016.
3. Gulisano, Vincenzo, Magnus Almgren, and Marina Papatriantafilou. "Online and scalable data validation in advanced metering infrastructures." IEEE PES Innovative Smart Grid Technologies, Europe. IEEE, 2014.
4. Gulisano, Vincenzo, Magnus Almgren, and Marina Papatriantafilou. "METIS: a two-tier intrusion detection system for advanced metering infrastructures." International Conference on Security and Privacy in Communication Systems. Springer International Publishing, 2014.
5. Yousefi, Saleh, Mahmoud Siadat Mousavi, and Mahmood Fathy. "Vehicular ad hoc networks (VANETs): challenges and perspectives." 2006 6th International Conference on ITS Telecommunications. IEEE, 2006.
6. El Zarki, Magda, et al. "Security issues in a future vehicular network." European Wireless. Vol. 2. 2002.
7. Georgiadis, Giorgos, and Marina Papatriantafilou. "Dealing with storage without forecasts in smart grids: Problem transformation and online scheduling algorithm." Proceedings of the 29th Annual ACM Symposium on Applied Computing. ACM, 2014.
8. Fu, Zhang, et al. "Online temporal-spatial analysis for detection of critical events in Cyber-Physical Systems." Big Data (Big Data), 2014 IEEE International Conference on. IEEE, 2014.

Bibliography

9. Arasu, Arvind, et al. "Linear road: a stream data management benchmark." Proceedings of the Thirtieth international conference on Very large data bases-Volume 30. VLDB Endowment, 2004.
10. Lv, Yisheng, et al. "Traffic flow prediction with big data: a deep learning approach." IEEE Transactions on Intelligent Transportation Systems 16.2 (2015): 865-873.
11. Grochocki, David, et al. "AMI threats, intrusion detection requirements and deployment recommendations." Smart Grid Communications (SmartGridComm), 2012 IEEE Third International Conference on. IEEE, 2012.
12. Molina-Markham, Andrés, et al. "Private memoirs of a smart meter." Proceedings of the 2nd ACM workshop on embedded sensing systems for energy-efficiency in building. ACM, 2010.
13. Gulisano, Vincenzo, et al. "Streamcloud: A large scale data streaming system." Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on. IEEE, 2010.
14. Stonebraker, Michael, Uğur Çetintemel, and Stan Zdonik. "The 8 requirements of real-time stream processing." ACM SIGMOD Record 34.4 (2005): 42-47.
15. Bonomi, Flavio, et al. "Fog computing and its role in the internet of things." Proceedings of the first edition of the MCC workshop on Mobile cloud computing. ACM, 2012.

Bibliography

16. Gulisano, Vincenzo Massimiliano. *StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine*. Diss. Informatica, 2012.
17. Cardellini, Valeria, et al. "Optimal operator placement for distributed stream processing applications." Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems. ACM, 2016.
18. Costache, Stefania, et al. "Understanding the Data-Processing Challenges in Intelligent Vehicular Systems." Proceedings of the 2016 IEEE Intelligent Vehicles Symposium (IV16).
19. Giatrakos, Nikos, Antonios Deligiannakis, and Minos Garofalakis. "Scalable Approximate Query Tracking over Highly Distributed Data Streams." Proceedings of the 2016 International Conference on Management of Data. ACM, 2016.
20. Gulisano, Vincenzo, et al. "Streamcloud: An elastic and scalable data streaming system." IEEE Transactions on Parallel and Distributed Systems 23.12 (2012): 2351-2365.
21. Shah, Mehul A., et al. "Flux: An adaptive partitioning operator for continuous query systems." Data Engineering, 2003. Proceedings. 19th International Conference on. IEEE, 2003.

Bibliography

22. Cederman, Daniel, et al. "Brief announcement: concurrent data structures for efficient streaming aggregation." Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures. ACM, 2014.
23. Ji, Yuanzhen, et al. "Quality-driven processing of sliding window aggregates over out-of-order data streams." Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems. ACM, 2015.
24. Ji, Yuanzhen, et al. "Quality-driven disorder handling for concurrent windowed stream queries with shared operators." Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems. ACM, 2016.
25. Gulisano, Vincenzo, et al. "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join." Big Data (Big Data), 2015 IEEE International Conference on. IEEE, 2015.
26. Ottenwälder, Beate, et al. "MigCEP: operator migration for mobility driven distributed complex event processing." Proceedings of the 7th ACM international conference on Distributed event-based systems. ACM, 2013.
27. De Matteis, Tiziano, and Gabriele Mencagli. "Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing." Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2016.
28. Balazinska, Magdalena, et al. "Fault-tolerance in the Borealis distributed stream processing system." ACM Transactions on Database Systems (TODS) 33.1 (2008): 3.
29. Castro Fernandez, Raul, et al. "Integrating scale out and fault tolerance in stream processing using operator state management." Proceedings of the 2013 ACM SIGMOD international conference on Management of data. ACM, 2013.

Bibliography

30. Dwork, Cynthia. "Differential privacy: A survey of results." International Conference on Theory and Applications of Models of Computation. Springer Berlin Heidelberg, 2008.
31. Dwork, Cynthia, et al. "Differential privacy under continual observation." Proceedings of the forty-second ACM symposium on Theory of computing. ACM, 2010.
32. Kargl, Frank, Arik Friedman, and Roksana Boreli. "Differential privacy in intelligent transportation systems." Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks. ACM, 2013.