POLITECNICO
MILANO 1863

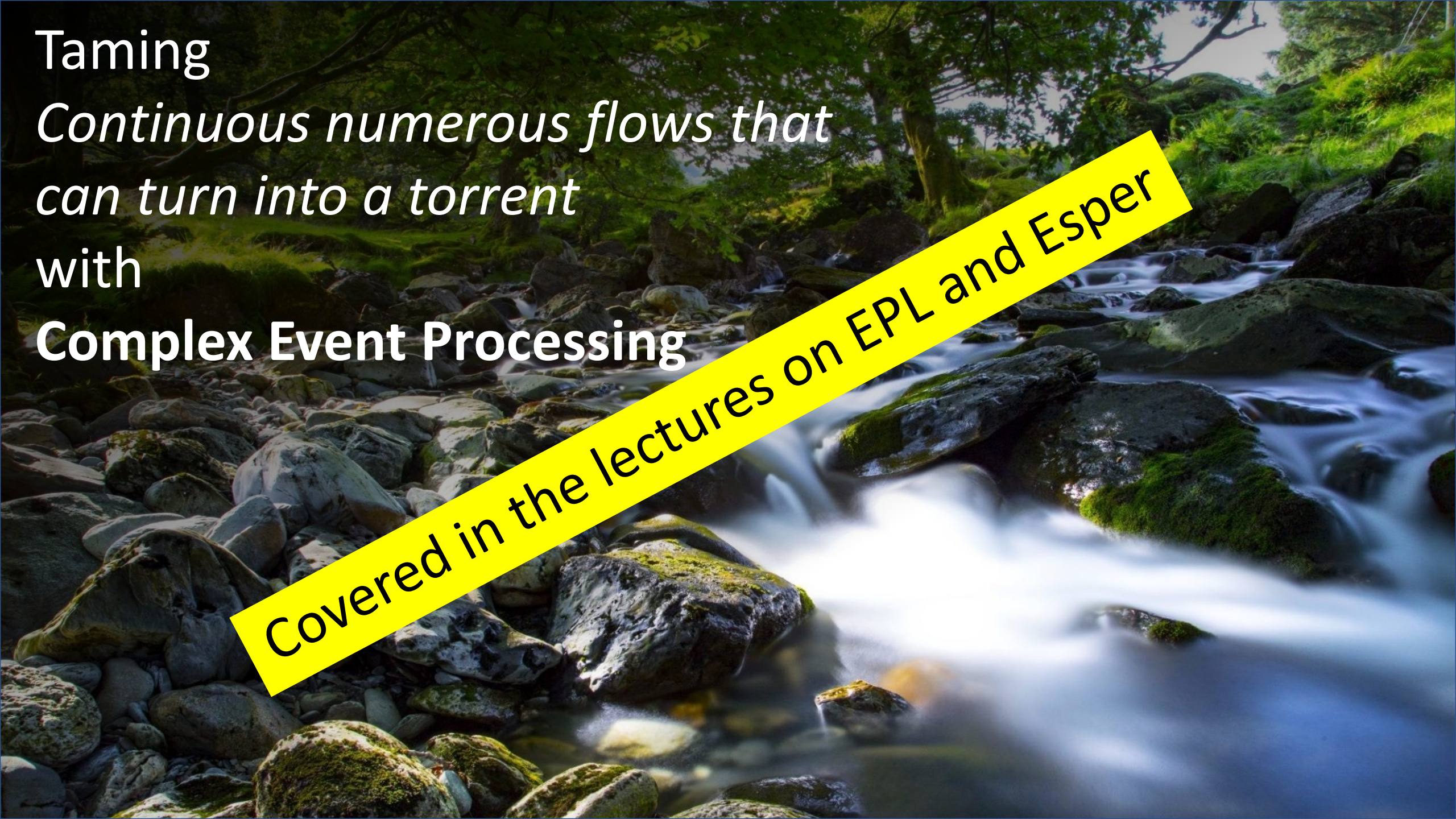
Scaling stream processing with Kafka and Spark

Emanuele Della Valle

emanuele.dellavalle@polimi.it

POLITECNICO
MILANO 1863

Position of this class in the course


Taming
Continuous numerous flows that
can turn into a torrent
with
Complex Event Processing
Covered in the lectures on EPL and Esper

Taming
Myriads of tiny flows
that you can collect
with
Event-based Systems

Image by Pexels from Pixabay

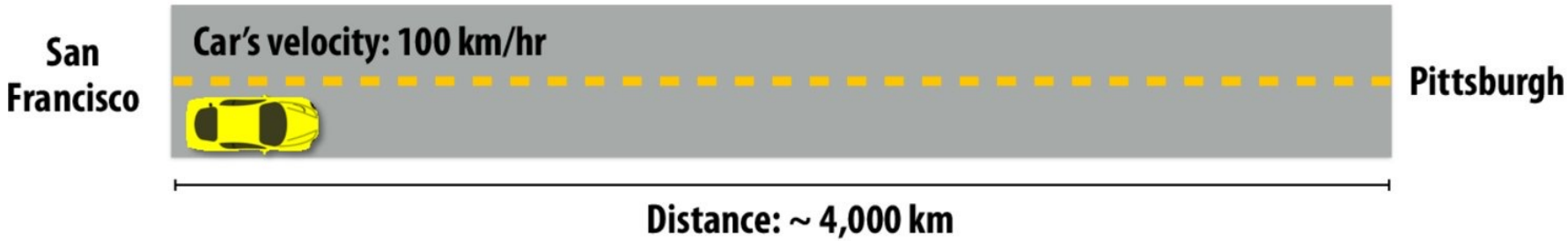
Taming
Continuous massive flows
than you cannot stop
with
Data Stream Mng. Systems

Throughput
Latency
Data/Message

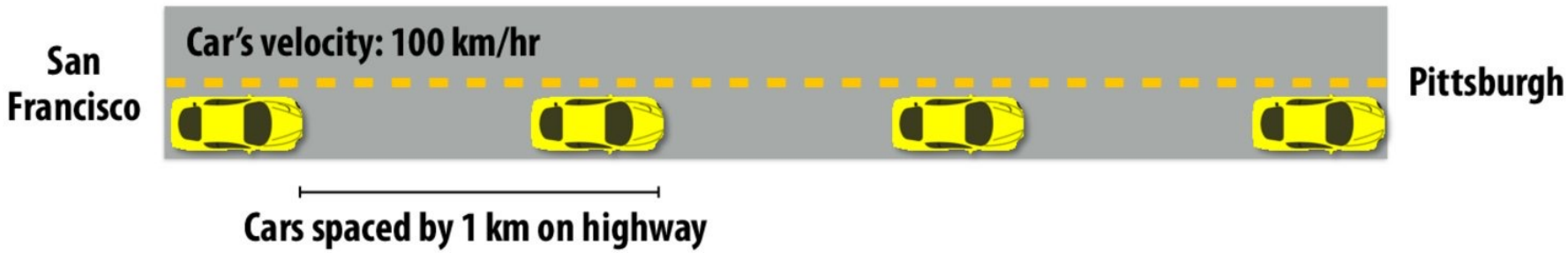
Few important concepts

Everyone wants to get to Pittsburgh!

(Latency vs. throughput review)

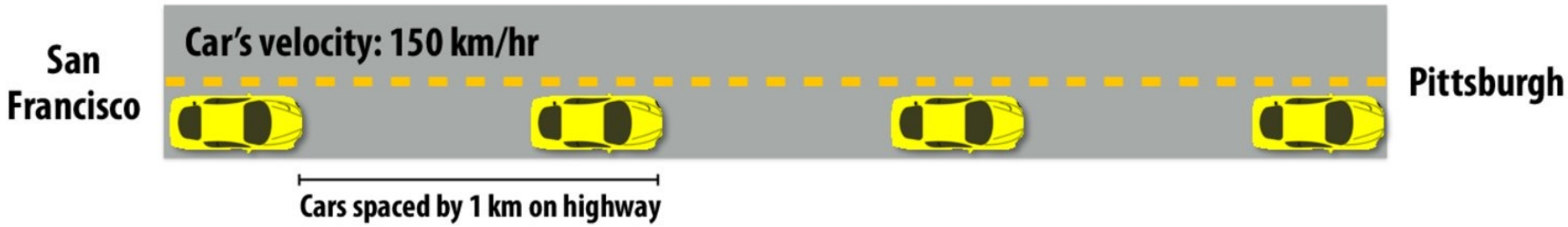


Latency of moving a person from San Francisco to Pittsburgh: 40 hours



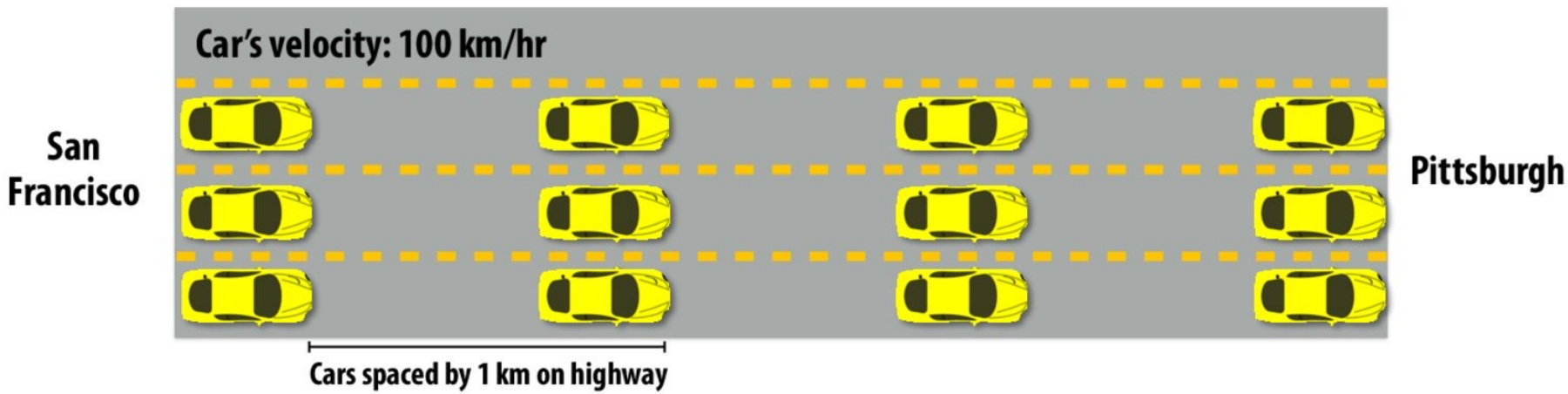
Throughput: 100 people per hour (1 car every 1/100 of an hour)

Improving throughput



Approach 1: drive faster!

Throughput = 150 people per hour (1 car every 1/150 of an hour)



Approach 2: build more lanes!

Throughput: 300 people per hour (3 cars every 1/100 of an hour)

Review: latency vs throughput

POLITECNICO
MILANO 1863

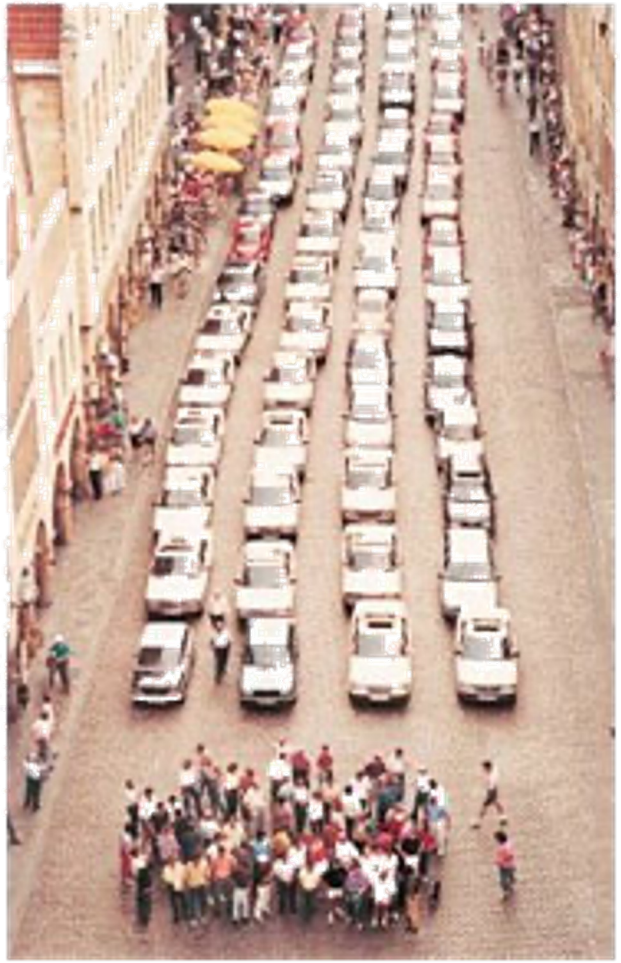
Latency

The amount of time needed for an operation to complete.
A memory load that misses the cache has a latency of 200 cycles
A packet takes 20 ms to be sent from my computer to Google
Asking a question on Piazza gets response in 10 minutes

throughput

The rate at which operations are performed.
Memory can provide data to the processor at 25 GB/sec.
A communication link can send 10 million messages per second
The TAs answer 50 questions per day on Piazza

space required to transport 60 people



car

bus

bicycle

(Poster in city of Muenster Planning Office, August 2001) Credit: PressOffice City of Munster, Germany

Data/Message matters!

POLITECNICO
MILANO 1863

- Fixed the size of a message (i.e., the space between two cars)
- The more people fit in the car, the higher the throughput
- The larger the data points per message, the higher the throughput

- **Compression** and **binary encoding** are the typical ways to increase data per message

Taming
Myriads of tiny flows
that you can collect
with
Event-based Systems

Image by Pexels from Pixabay

Kafka History

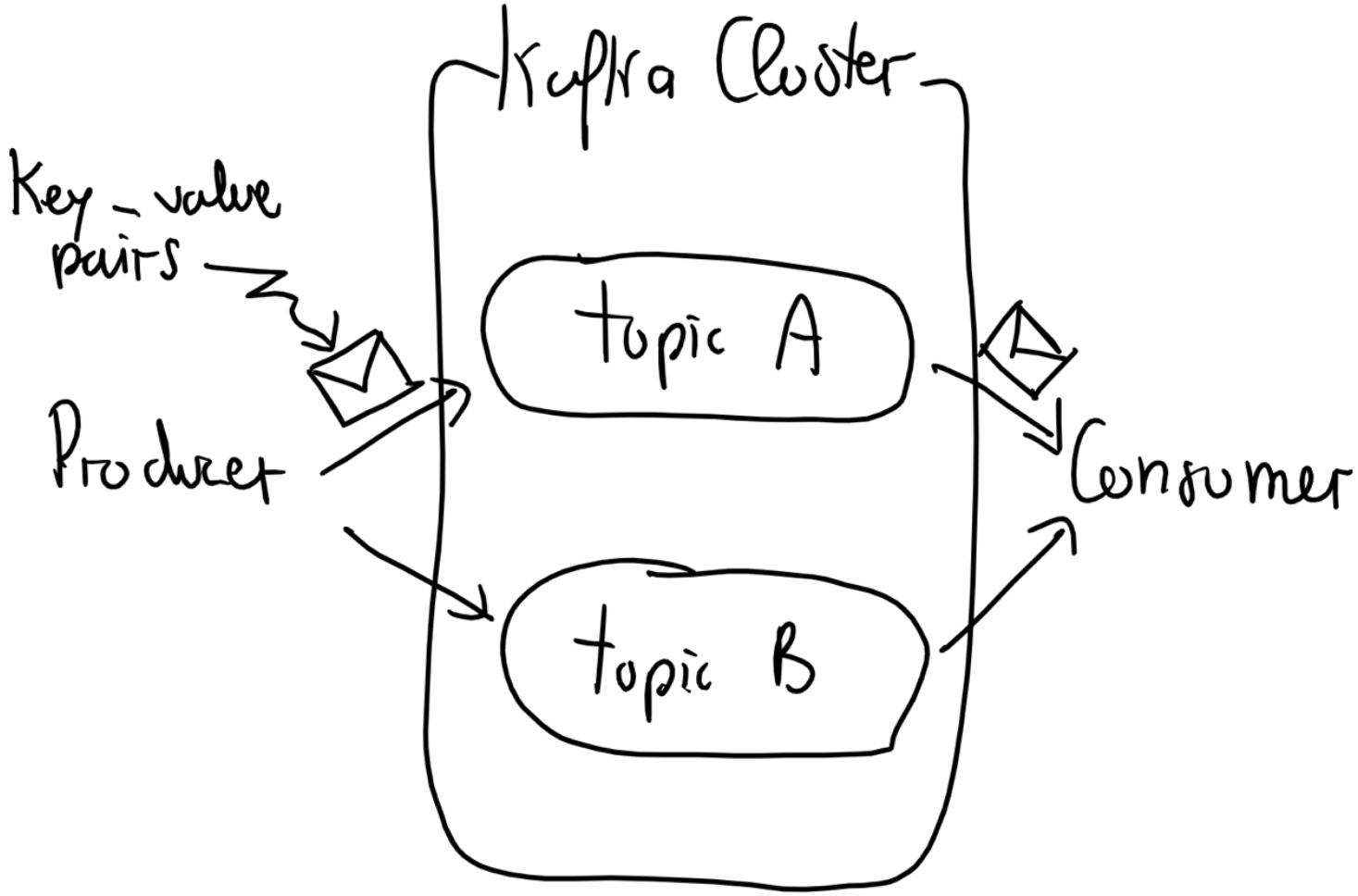
POLITECNICO
MILANO 1863

- In 2010, it was created at LinkedIn to
  - Simplifying data pipelines
  - Processing streaming data for batch and real-time analytics
- In 2012, it became a top-level Apache project
- Now, it is now at the core of LinkedIn's architecture
  - Performs extremely well at very large scale
  - Produces over 2 *trillion* messages per day

  and it is in use at many organizations
  - Twitter, Netflix, Goldman Sachs, Hotels.com, IBM, Spotify, Uber, Square, Cisco...

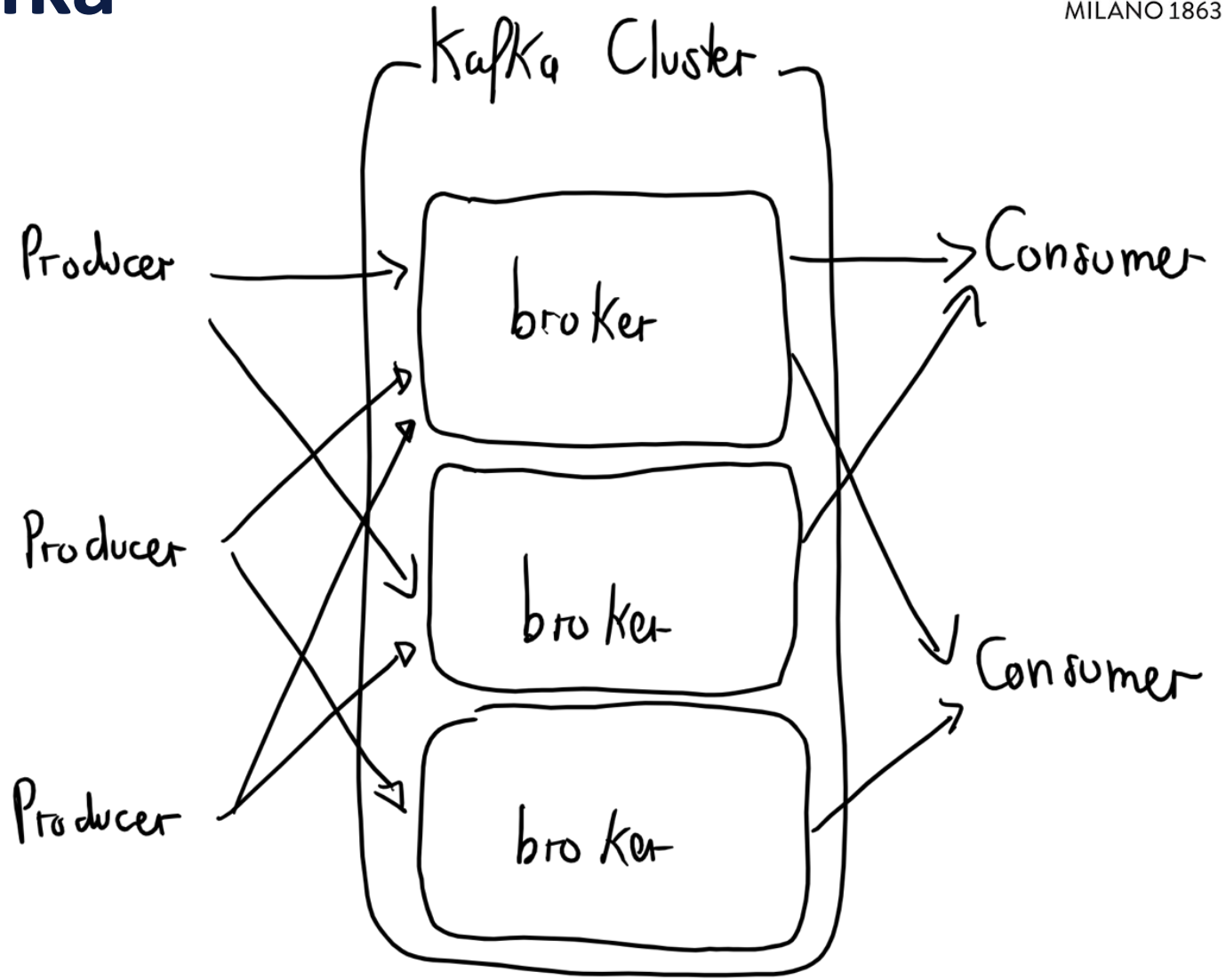
A Conceptual View of Kafka

- **Producers** send messages on topics
- **Consumers** read messages from topics
- **Messages** are key-value pairs
- **Topics** are streams of messages
- Kafka cluster manages topics



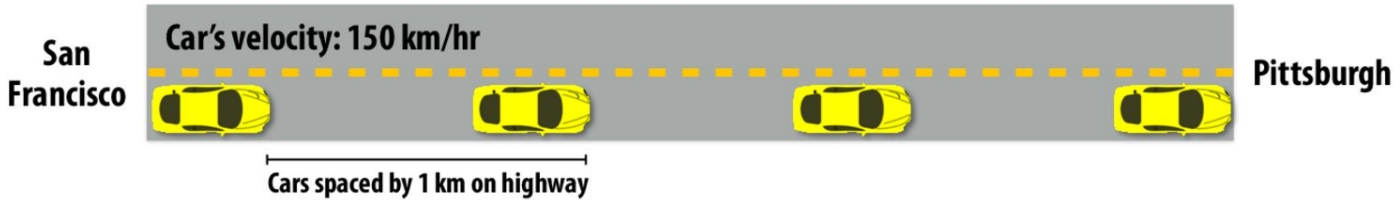
A System View of Kafka

- **Brokers** are the main storage and messaging components of the Kafka cluster



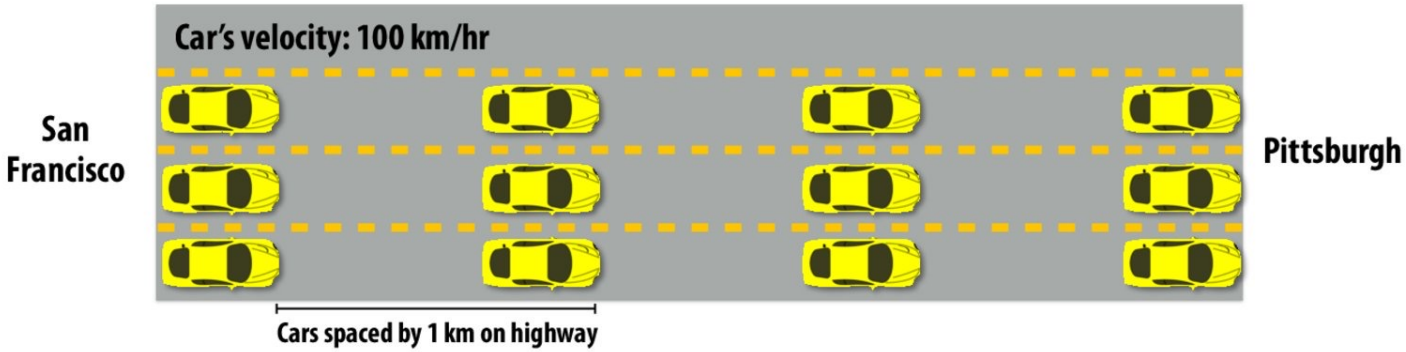
Notice the similarity

Improving throughput



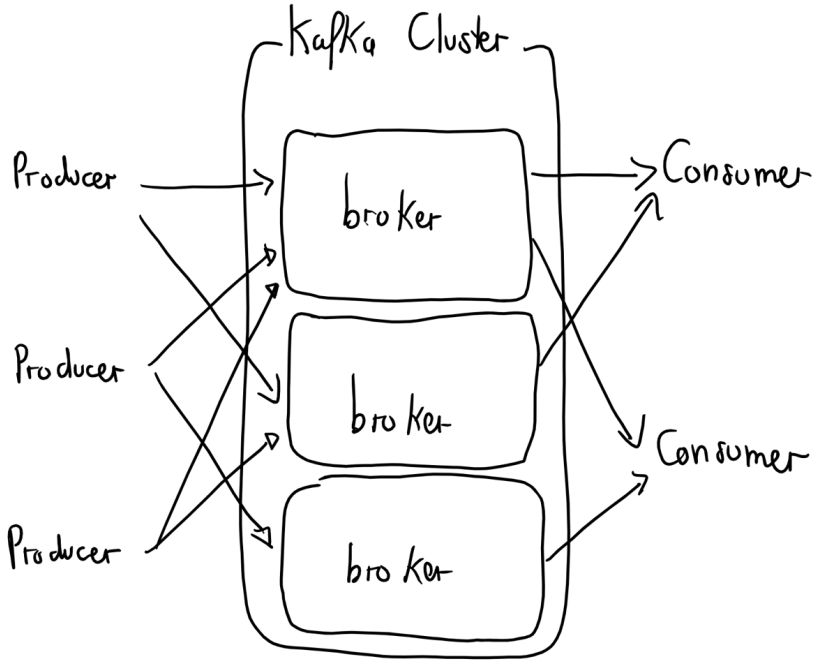
Approach 1: drive faster!

Throughput = 150 people per hour (1 car every 1/150 of an hour)



Approach 2: build more lanes!

Throughput: 300 people per hour (3 cars every 1/100 of an hour)



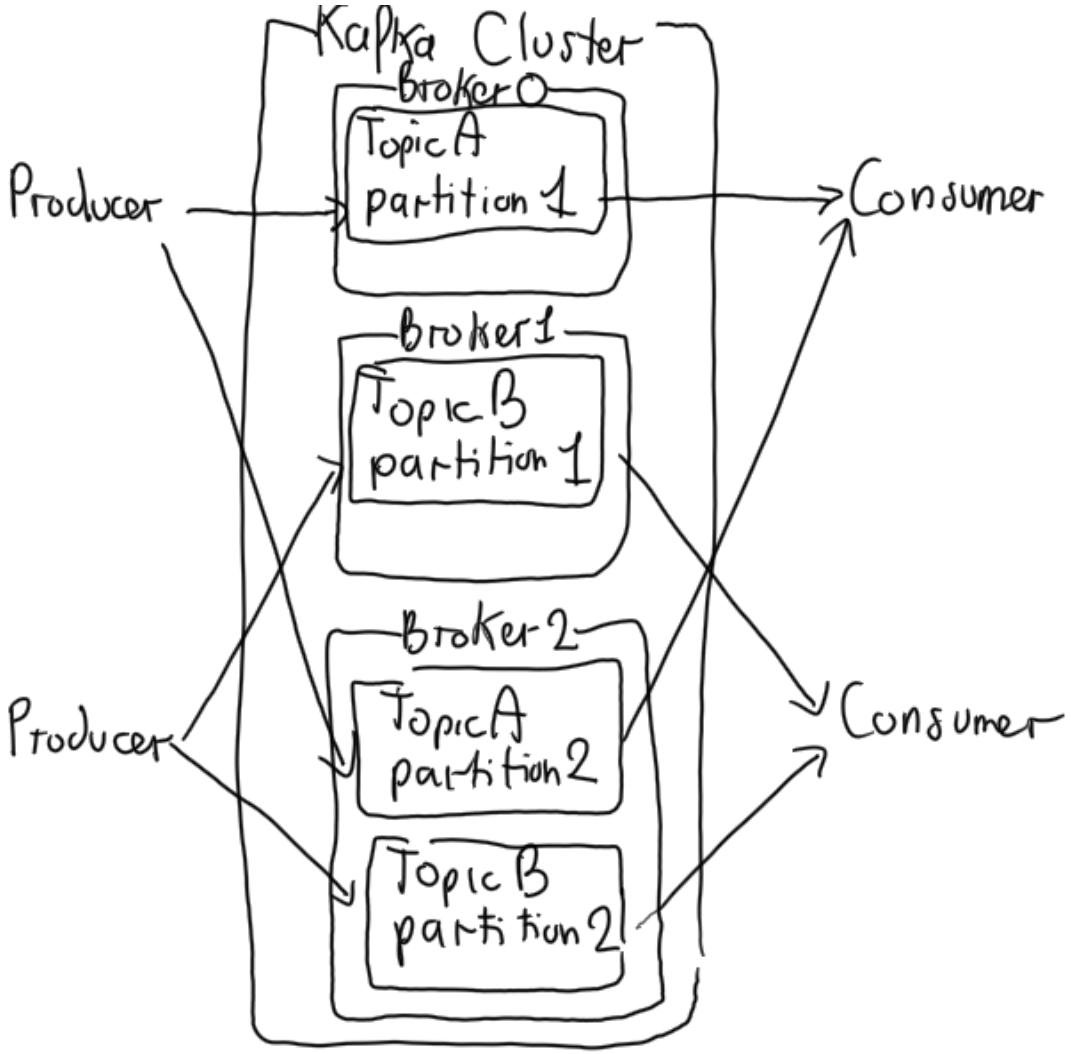
More on brokers

POLITECNICO MILANO 1863

- Brokers receive and store messages when they are sent by the Producers
- A production Kafka cluster will have three or more Brokers
  - Each can handle hundreds of thousands, or millions, of messages per second
- Typically, a Broker manages multiple Partitions

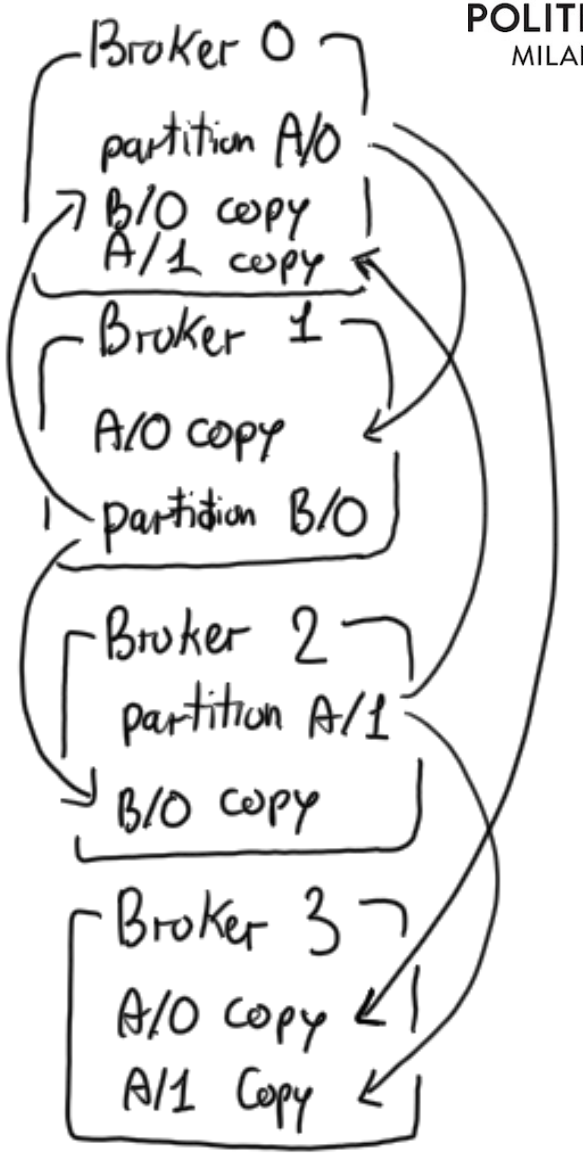
Reconciling the two views

- Topics are **partition**ed across brokers
- **Producers shard messages** over the partitions of a certain topic
- Typically, the message **key determines** which **partition** a message is assigned to



Fault Tolerance via a Replicated Log

- Kafka maintains replicas of each partition on other Brokers in the cluster
  - Number of replicas is configurable
- One Broker is the leader for that Partition
  - All writes and reads go to and from the leader
  - Other Brokers are followers
- Replication provides fault tolerance in case a Broker goes down



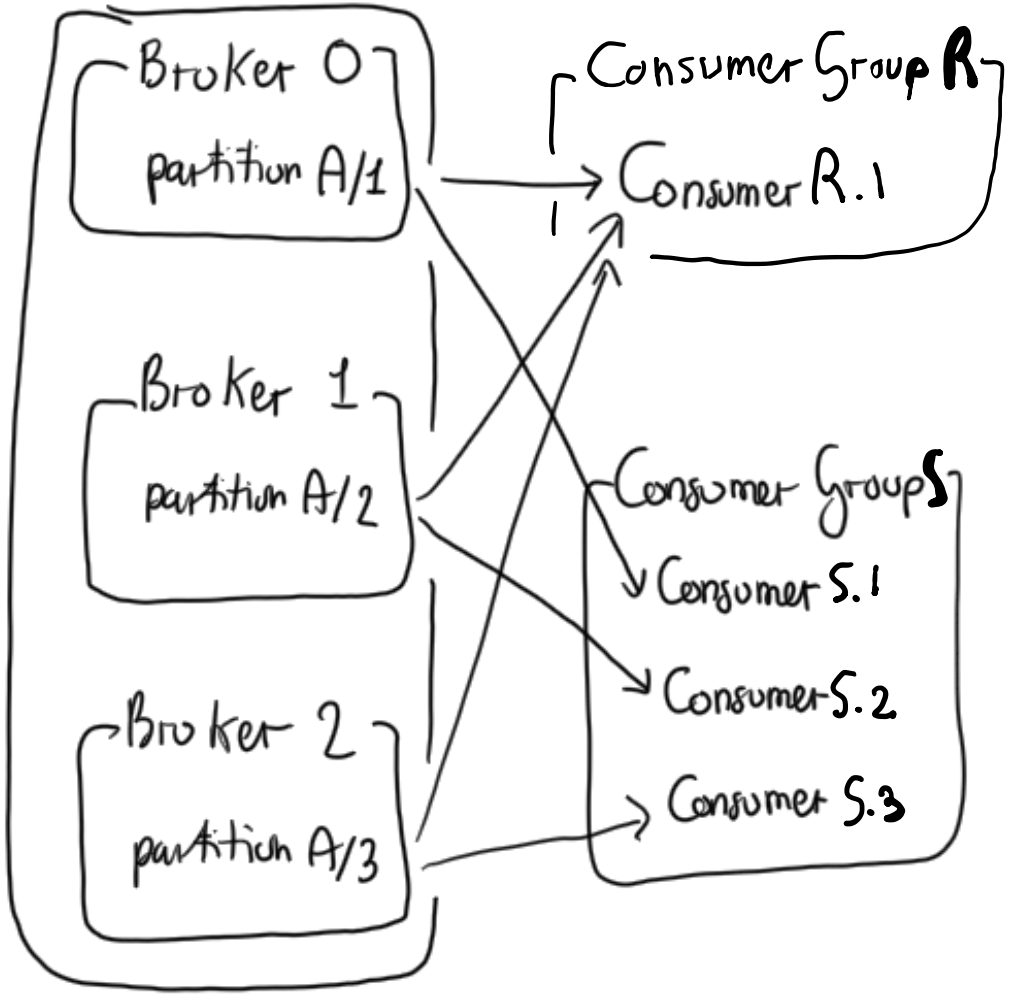
Producers, fault tolerance and durability

POLITECNICO MILANO 1863

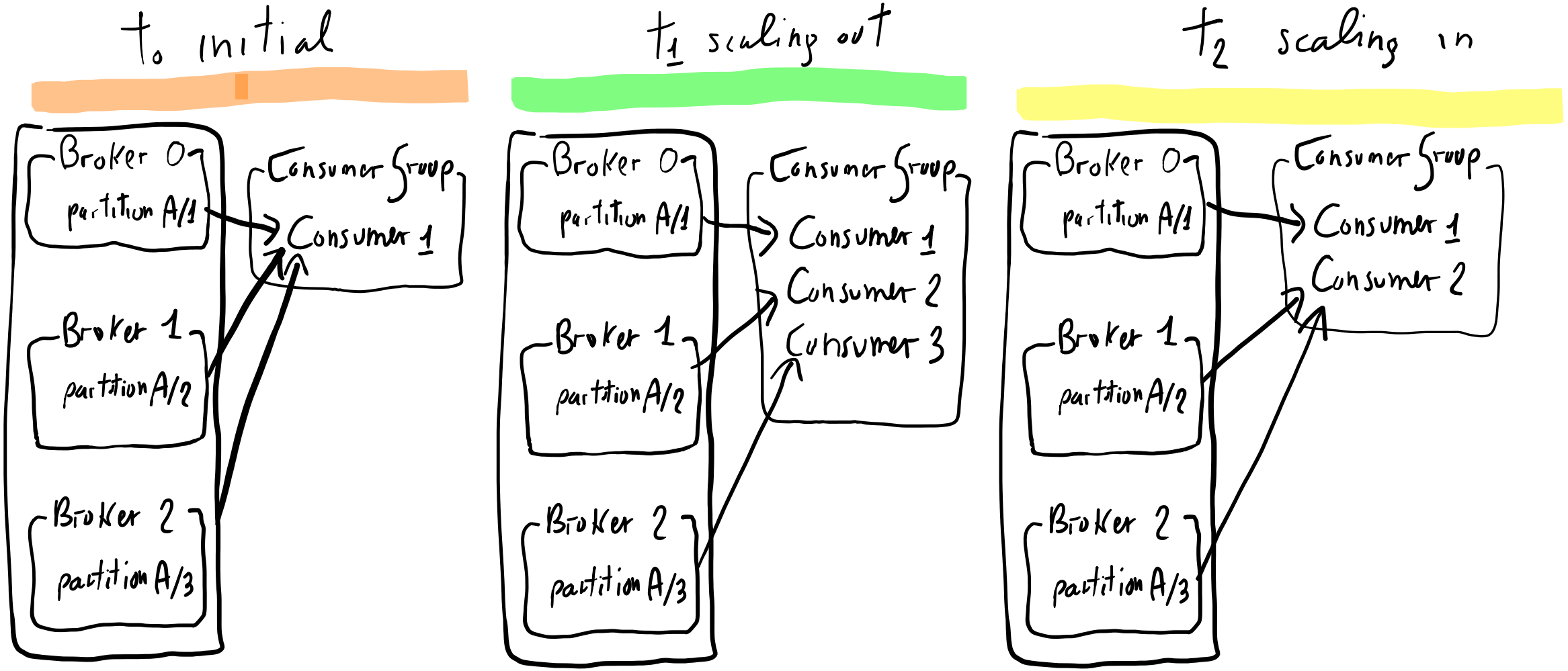
- Producers can control durability by requiring the leader a number of acknowledgments the before considering the request complete.
  - **acks=0**
    Producer will not wait for any acknowledgment from the broker
  - **acks=1**
    Producer will wait until the leader has written the record to its local log
  - **acks=all**
    Producer will wait until all insync replicas have acknowledged receipt of the record

Topic partitioning invites distributed consumption

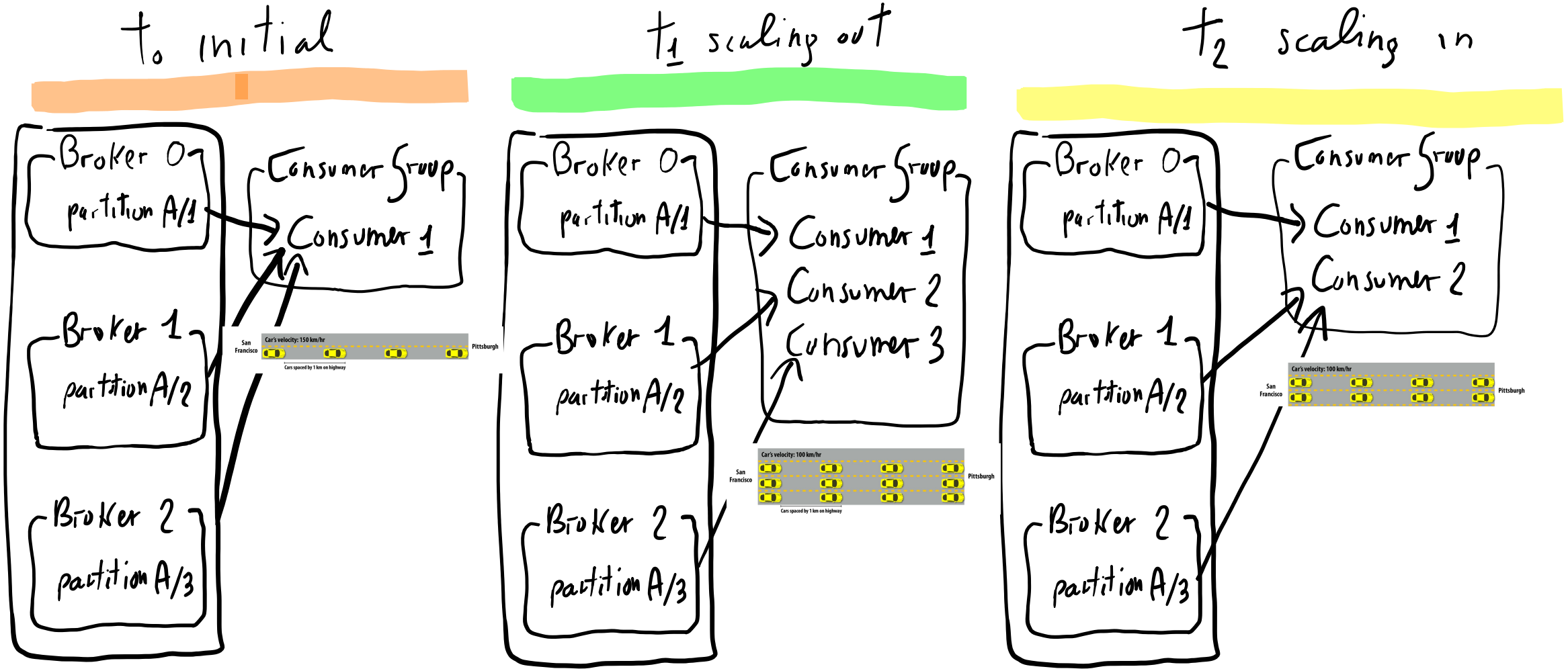
- Different Consumers can read data from the same Topic
  - By default, each Consumer will receive all the messages in the Topic
- Multiple Consumers can be combined into a **Consumer Group**
  - Consumer Groups provide scaling capabilities
  - Each Consumer is assigned a subset of Partitions for consumption



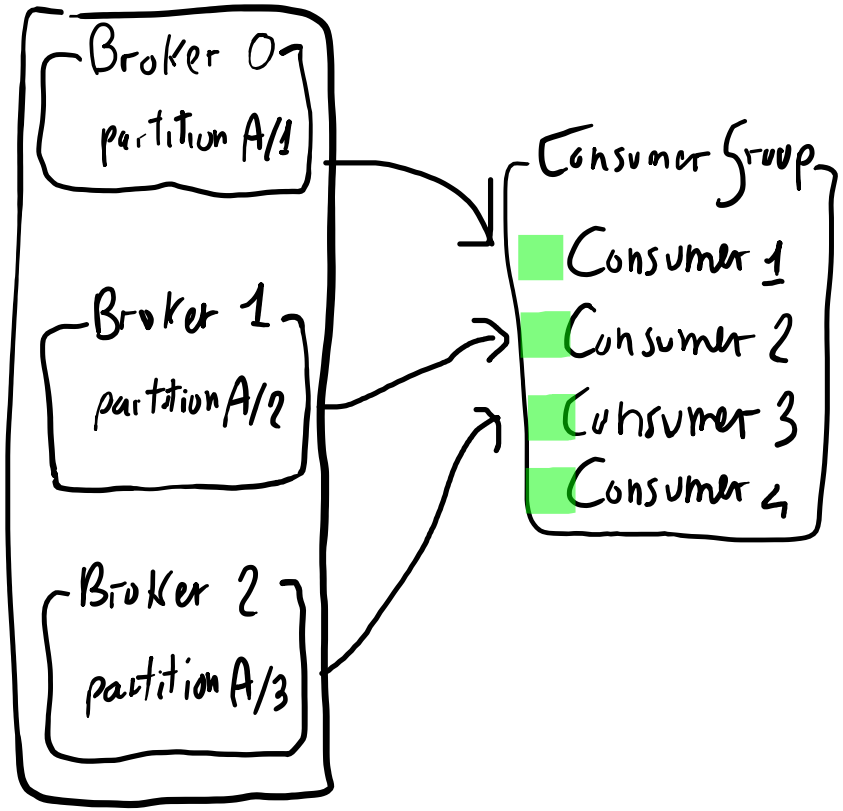
Consumer Group and scalability

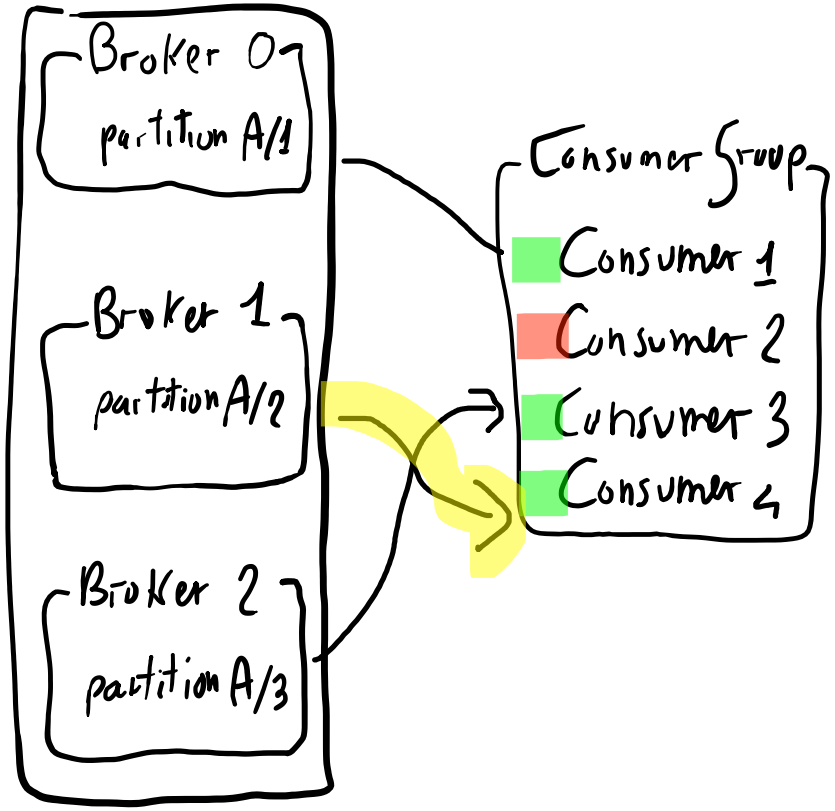


Once again, notice the similarity



Consumer Group and fault tollerance





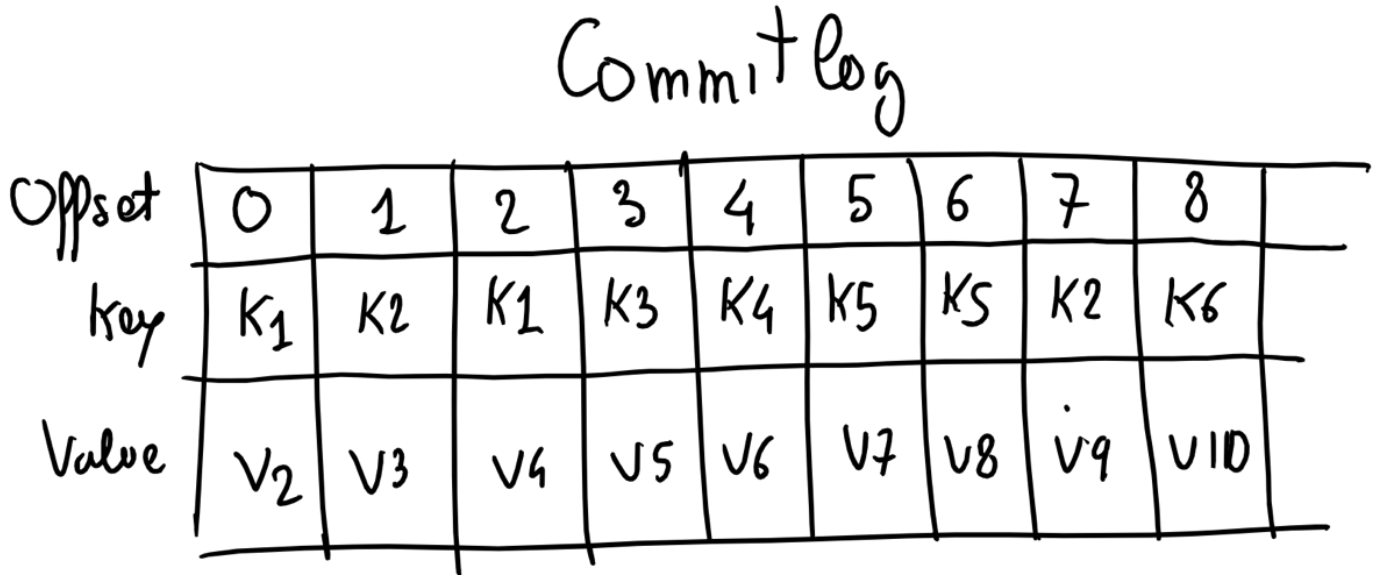
Quiz

POLITECNICO
MILANO 1863

- Provide the correct relationship — 1:1, 1:N, N:1, or N:N –
  - Broker to Partition — ?
  - Key to Partition — ?
  - Producer to Topic — ?
  - Consumer Group to Topic — ?
  - Consumer (in a Consumer Group) to Partition — ?

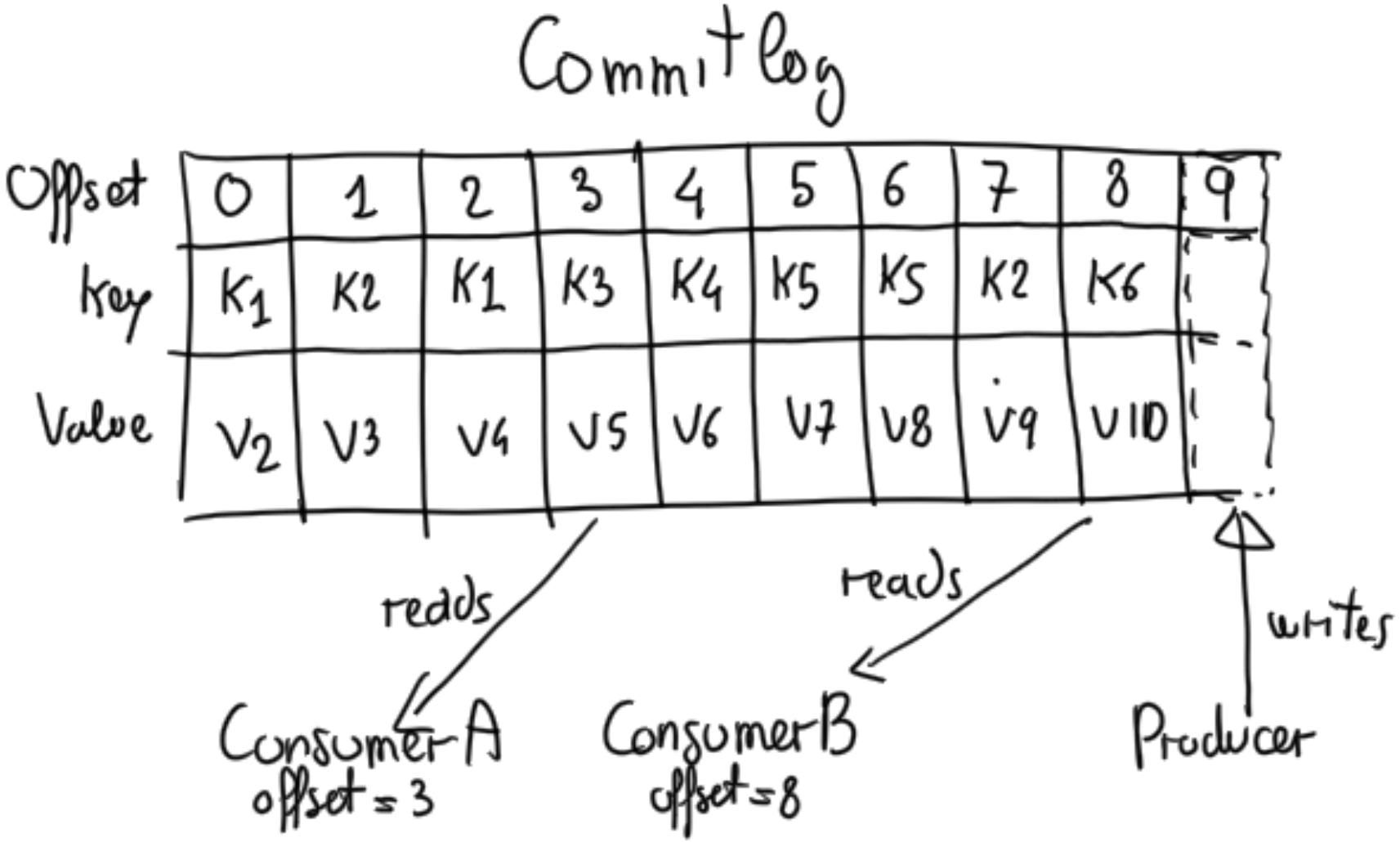
A Physical View of topic partition

- Each Partition is stored on the Broker's disk as one or more log files
- Each message in the log is identified by its **offset** number



A Physical View of topic partition (cont.)

- Messages are always appended
- Consumers can consume from different offset
- Brokers are single thread to guarantee consistency



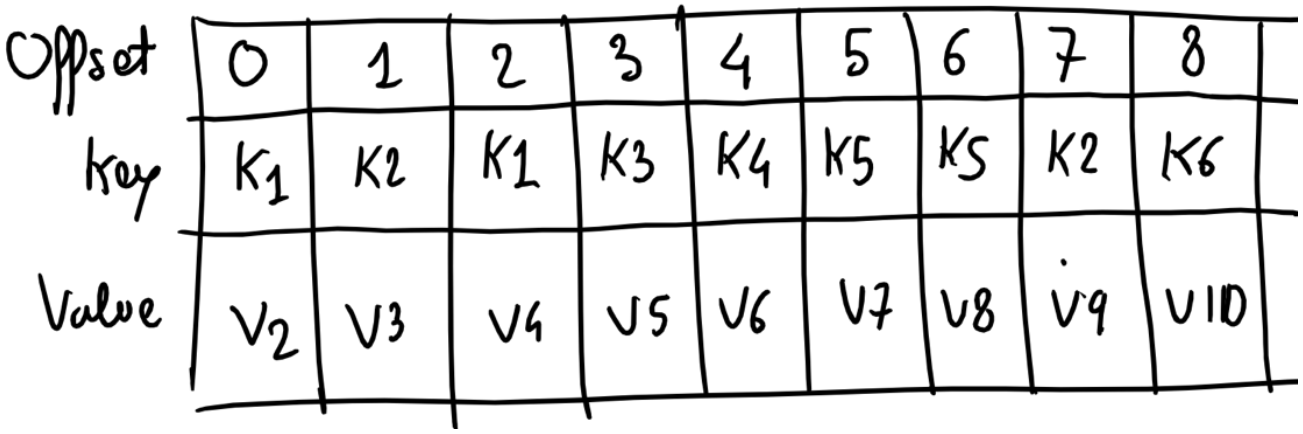
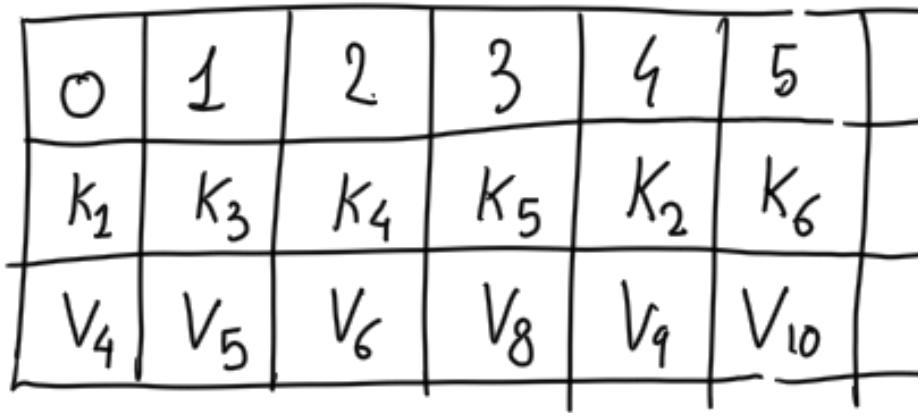
Log Retention

POLITECNICO MILANO 1863

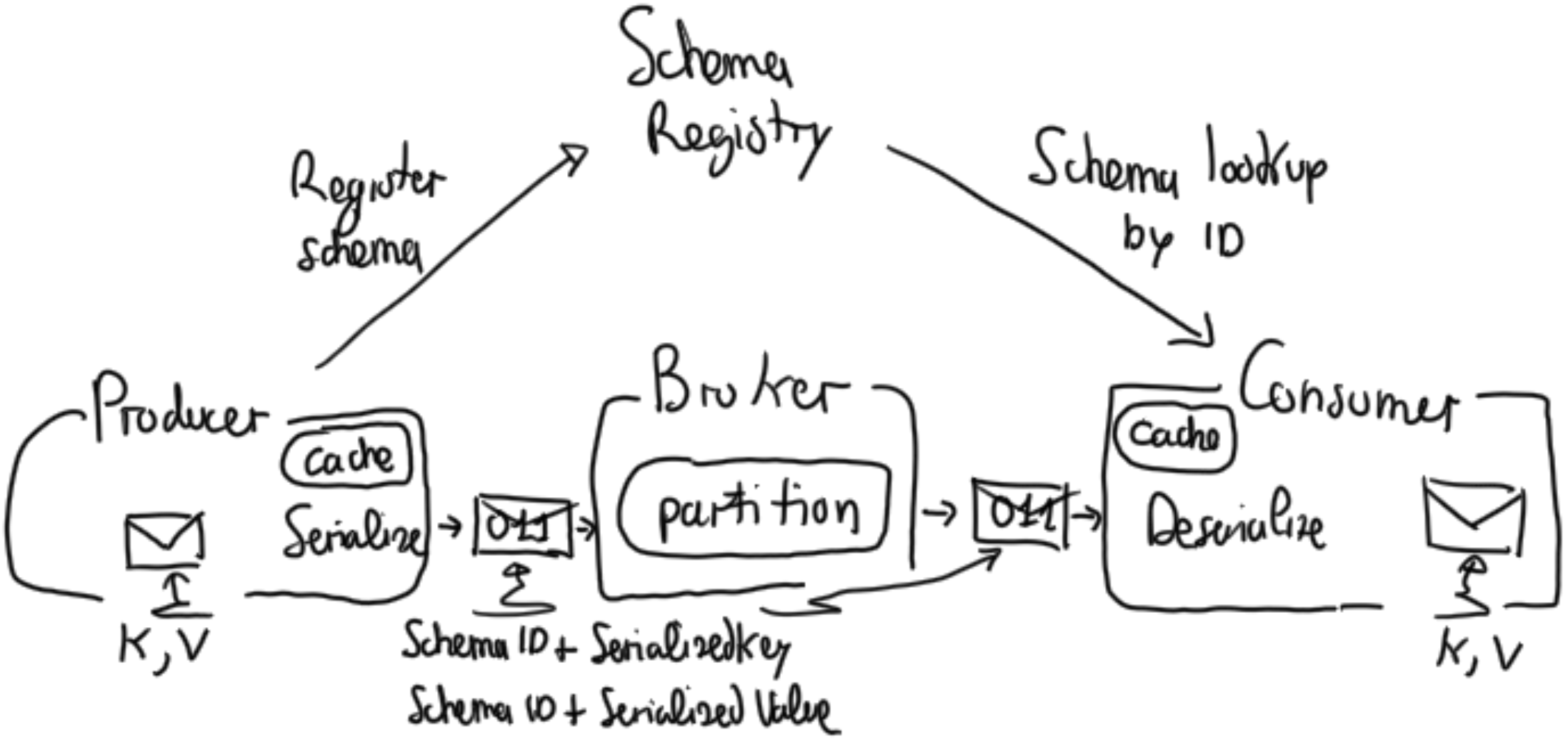
- Duration default: messages will be retained for seven days
- Duration is configurable per Broker by setting
  - a time period
  - a size limit
- Topic can override a Broker's retention policy
- When cleaning up a log
  - the default policy is delete
  - An alternate policy is compact

Log compaction

A compacted log retains at least the last known message value for each key within the Partition

Before compaction	After compaction
	

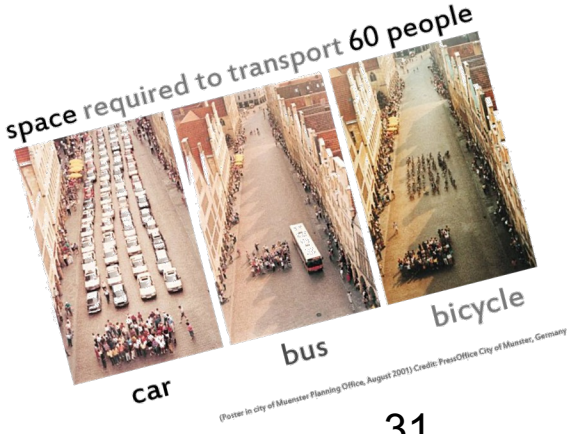
Avro and Schema Registry at work



Avro

POLITECNICO MILANO 1863

- Avro is an Apache open source project
- Provides data serialization into a binary format
  - so stores data efficiently
- Data is defined with a self-describing schema allowing for
  - code generation for serializers and de-serializers in multiple languages
  - type checking at write time



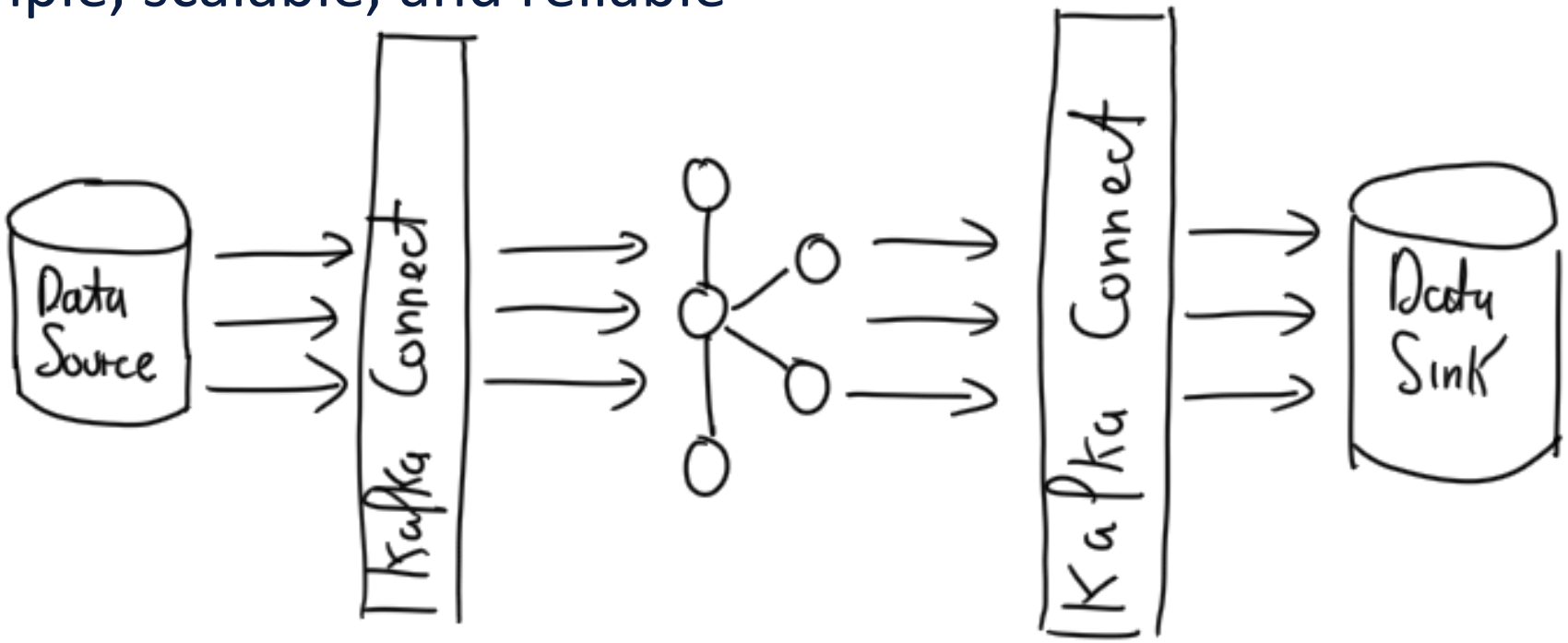
Schema Registry

POLITECNICO MILANO 1863

- Sending the Avro schema with each message would be inefficient
- The Schema Registry
  - is a Confluent component
  - provides centralized management of schemas
  - At registration time each schema is given an ID
- The mapping schema/ID is stored in a special Kafka topic

Kafka Connect

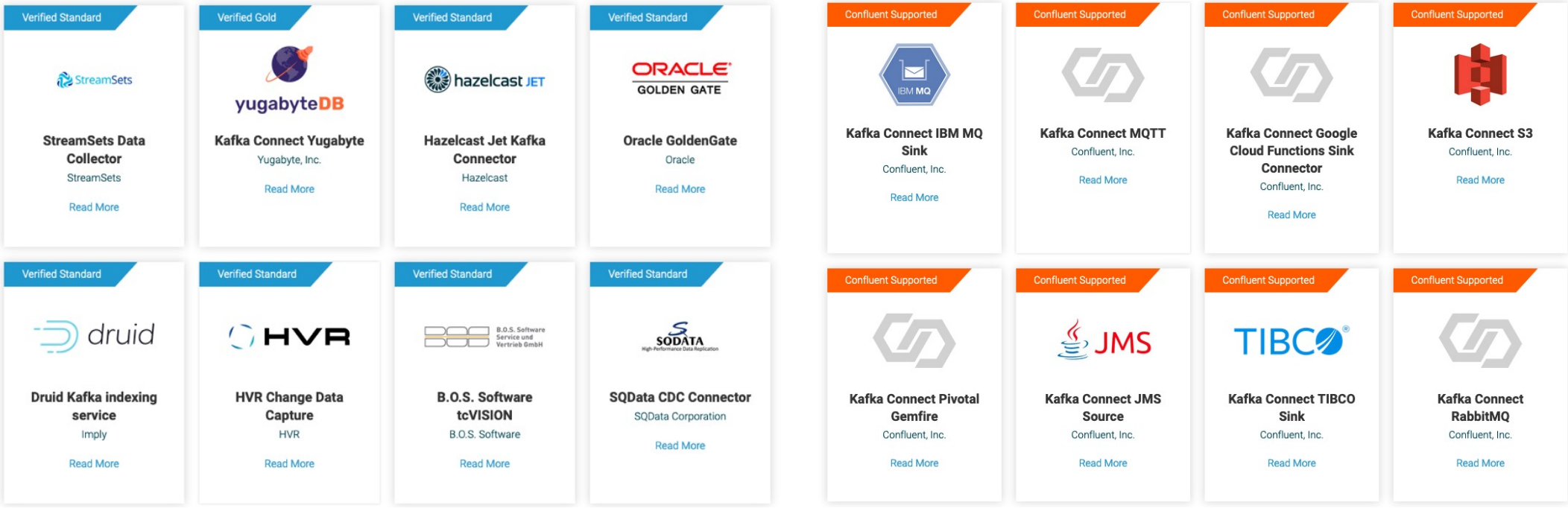
- It is an open source framework for streaming data between Apache Kafka and other data systems
- It is simple, scalable, and reliable



Off-The-Shelf Connectors

POLITECNICO MILANO 1863

https://www.confluent.io/product/connectors/

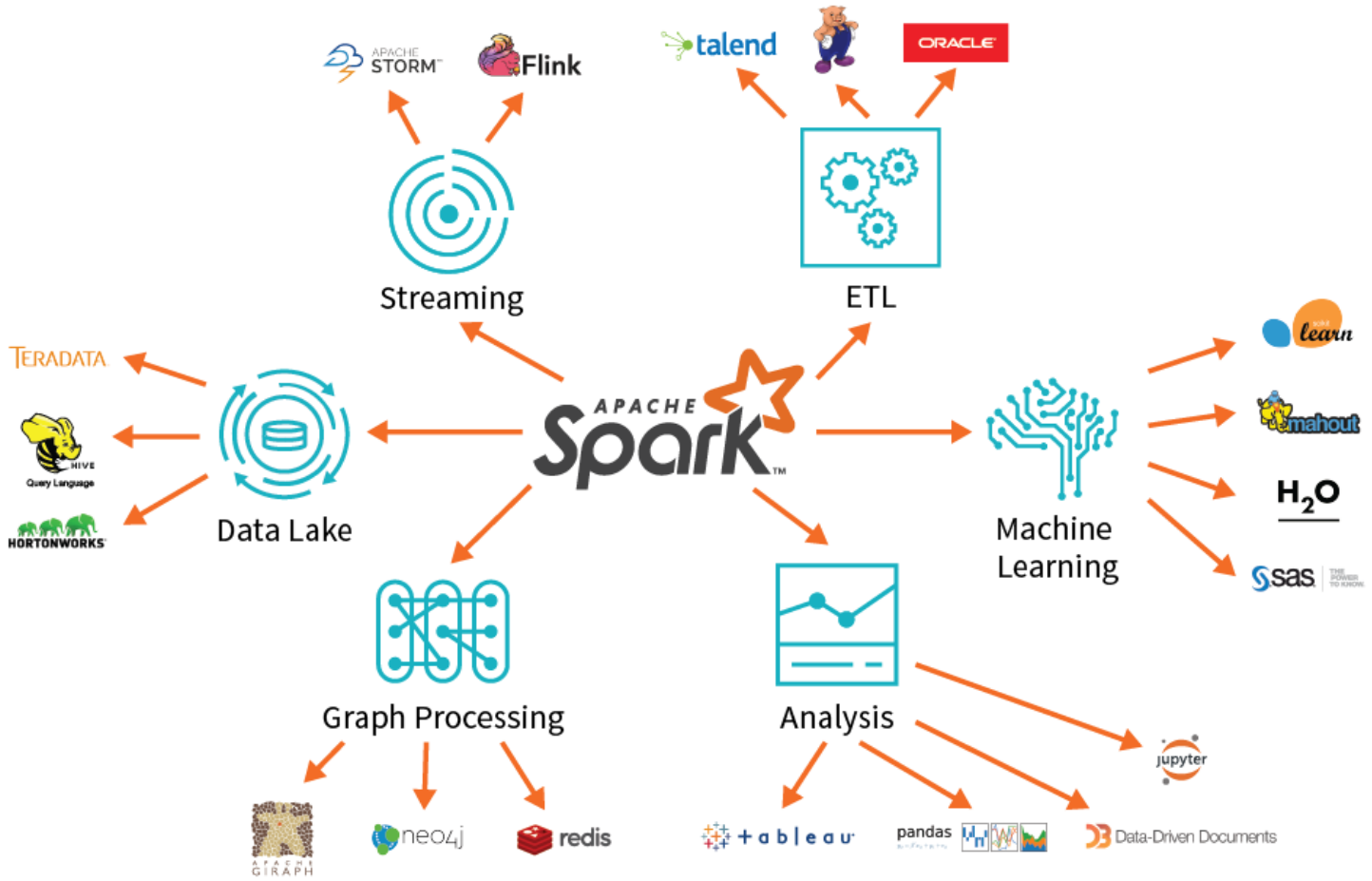


Taming
Continuous massive flows
than you cannot stop
with
Data Stream Mng. Systems

Apache Spark

POLITECNICO MILANO 1863

Unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing



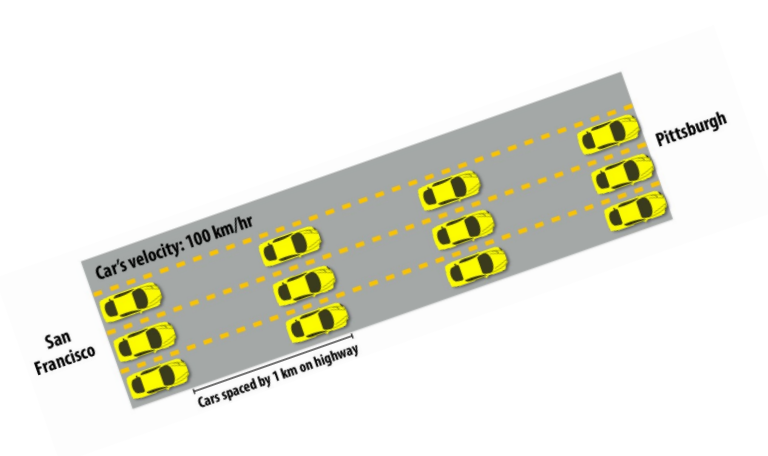
RDD

- Definition
  - ***R**esilient*: Fault-tolerant
  - ***D**istributed*: Computed across multiple nodes
  - ***D**ataset*: Collection of partitioned data
- Characteristics
  - Immutable once constructed
  - Track lineage information

- Logical view: single dataset

RDD

- Physical view: 1 or more partitions

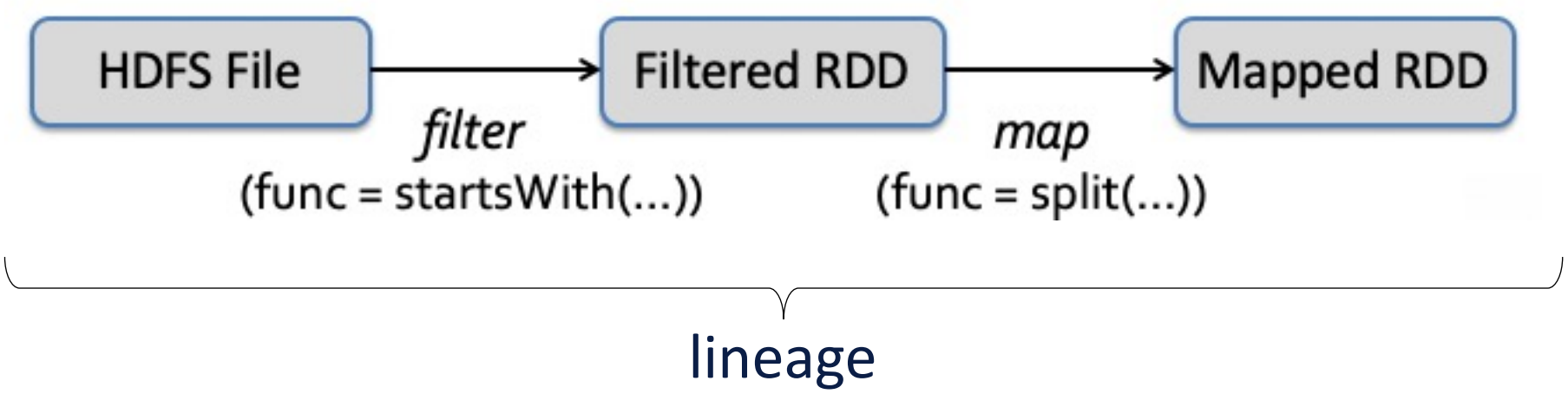


Transformations - logical view

POLITECNICO MILANO 1863

- RDD are immutable, transformations create new RDD

```
msgs = (textFile.filter(lambda s: s.startsWith("ERROR"))
                .map(lambda s: s.split("\t")[2]))
```



Actions – logical view

POLITECNICO MILANO 1863

- Actions either return a result or write to disc
- For example:
  - The number of records in the case of `count()`
  - An array of objects in the case of `collect()` or `take(n)`

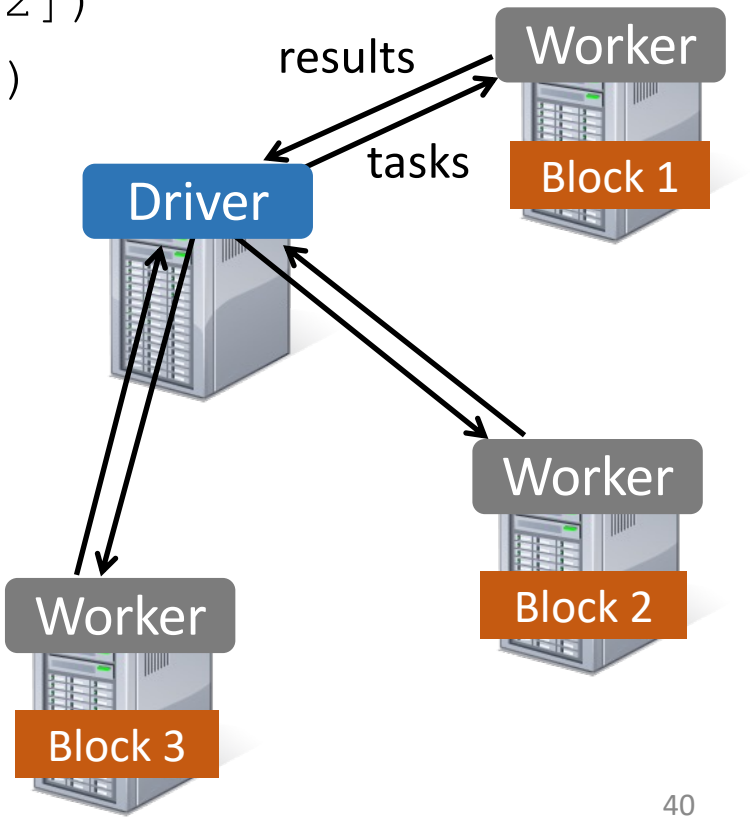
Actions – system view

Base RDD

```
lines = sc.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
message  filter(lambda s: "mysql" in s).count()
```

Transformed RDD

Action



Laziness By Design

POLITECNICO
MILANO 1863

- Fundamental to Apache Spark are the notions that
  - Transformations are **LAZY**
  - Actions are **EAGER**

- NOTE: We saw this play out in the previous slide when we run multiple transformations back-to-back, and no job was triggered until the first action was requested

DataFrame

POLITECNICO MILANO 1863

- Definition
  - Immutable Data with named columns
  - Built on RDDs

- Characteristics
  - User-friendly API
  - Uniform APIs across languages (Scala, Java, Python, R, and SQL)
  - Improved performance via optimizations (Tungsten and Catalyst)

DataFrame code snippet

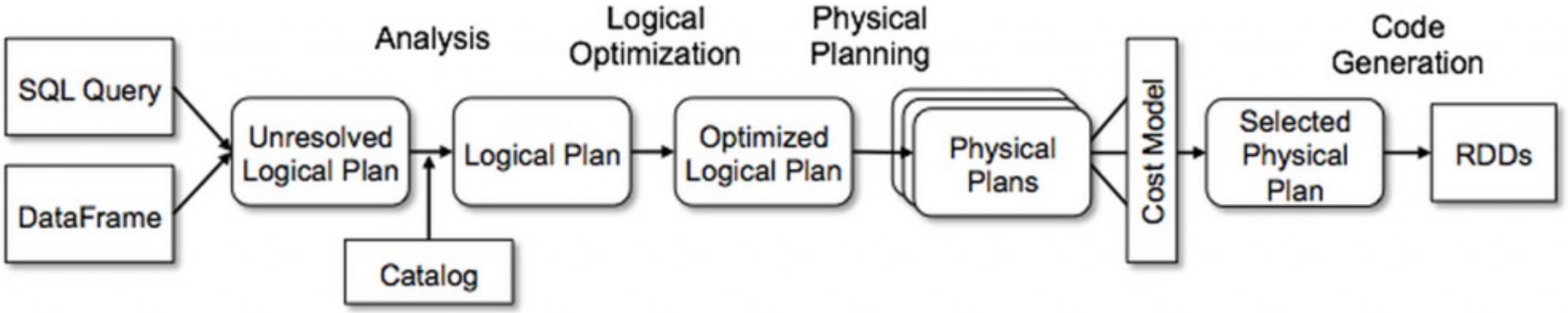
POLITECNICO MILANO 1863

- An Apache Spark code snippet using SQL and DataFrames to query and join different data sources

```
# Read a CSV
userDF = spark.read.csv(" … /userData.csv")
# ... or Use DataFrame APIs and register a temp view
middleageSmokers = userDF.filter(col("smoker")=="Y").filter(col("age")>40)
middleageSmokers.createOrReplaceTempView("middleageSmokers")
# ... read a CSV and register a temp view
spark.read.json(" … /part-00000.json.gz").createOrReplaceTempView("iot_stream")
# ... execute SQL query or ...
spark.sql("SELECT avg(calories_burnt) FROM iot_stream JOIN middleageSmokers ON …")
```

Under the Hood: Catalyst

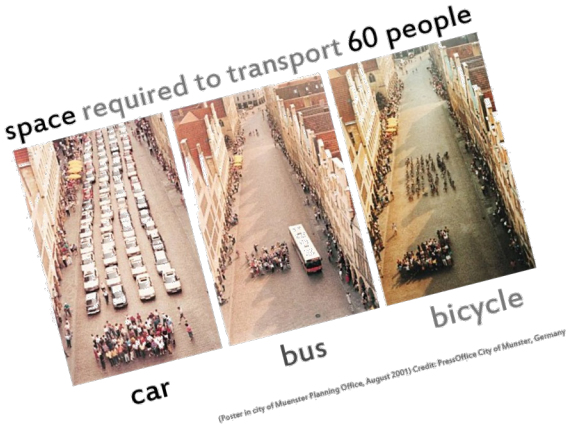
- Generates optimal RDD code from the user request



Under the Hood: Tungsten Binary Format

POLITECNICO MILANO 1863

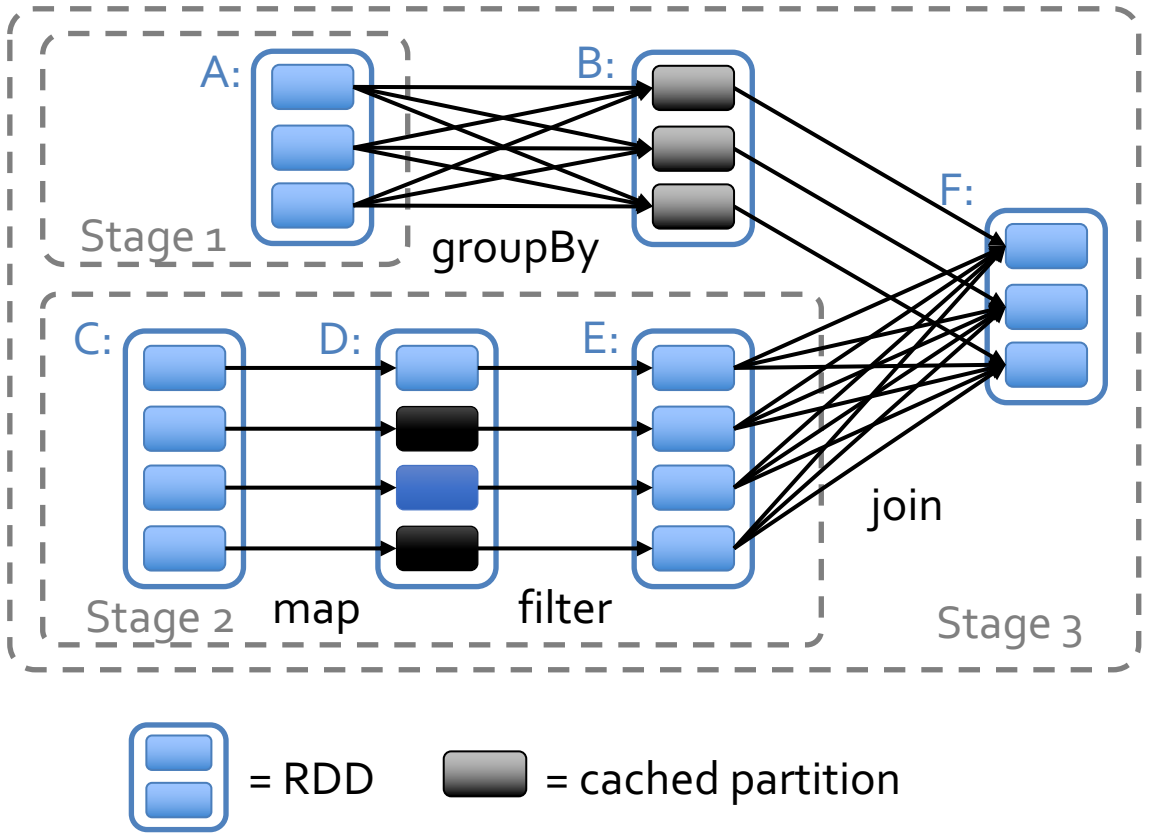
- Off-heap, compact and efficient in-memory storage row format



Spark at work – Jobs, Stages and Tasks

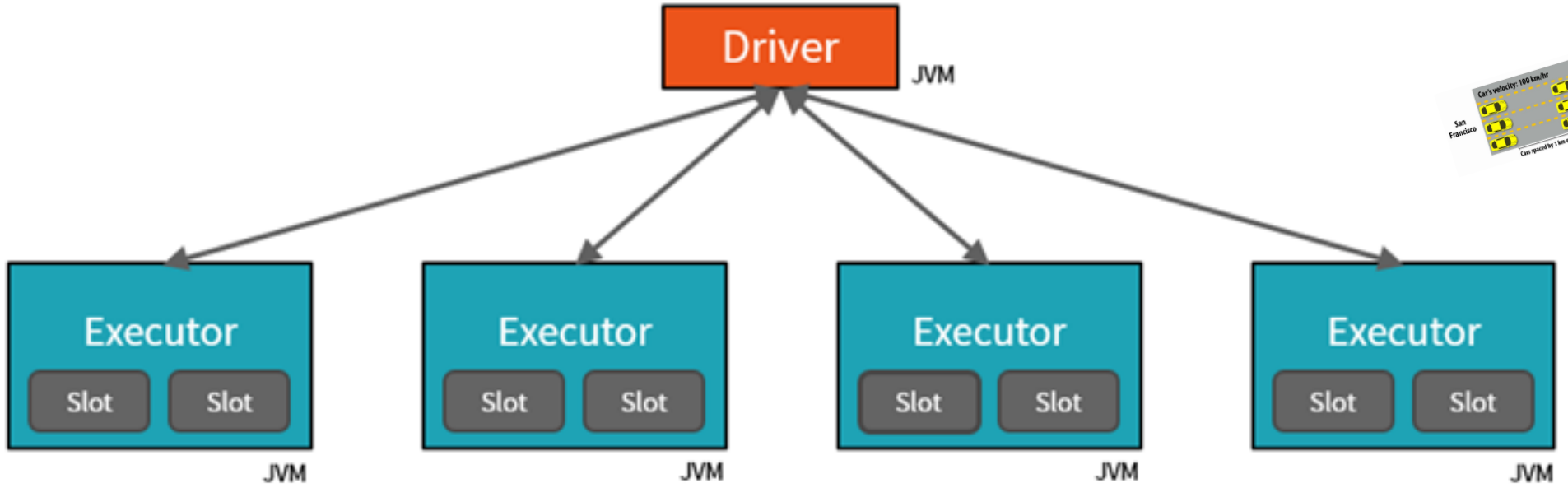
POLITECNICO MILANO 1863

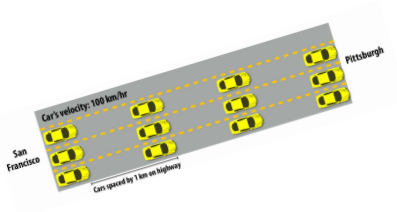
- The **DAG Scheduler** is the scheduling layer of Spark
- It transforms a logical execution plan into a physical one (a **Job**) considering data locality and partitioning (to avoid shuffles)
- Each **Job** is a DAG of **Stages** which are broken down into **Tasks**



Spark at work - components

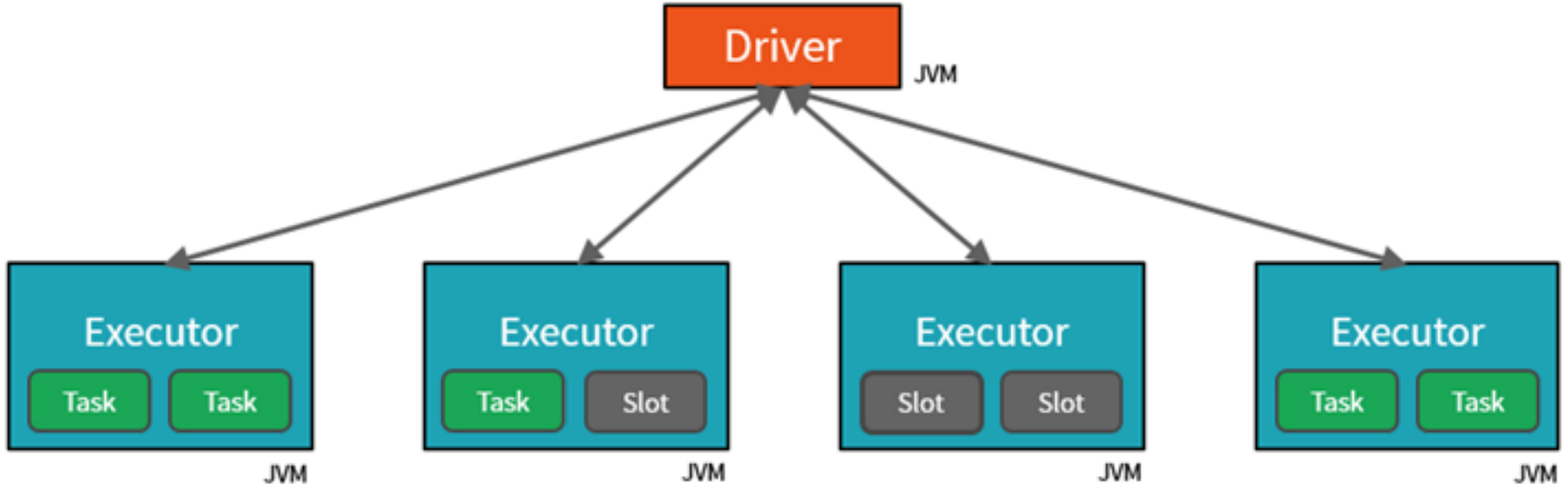
- A Spark cluster includes
  - One **Driver** running the user application
  - Multiple **Executors** (typically, one instance per node) with multiple **Slots** (typically, one per core/CPU)





Spark at work – runtime view

- The **Driver** assigns **Tasks** to Executors' **Slots** for parallel execution



- The Driver also makes sure that all tasks of a Stage are executed before starting any task of the next stage

Connecting Spark to kafka