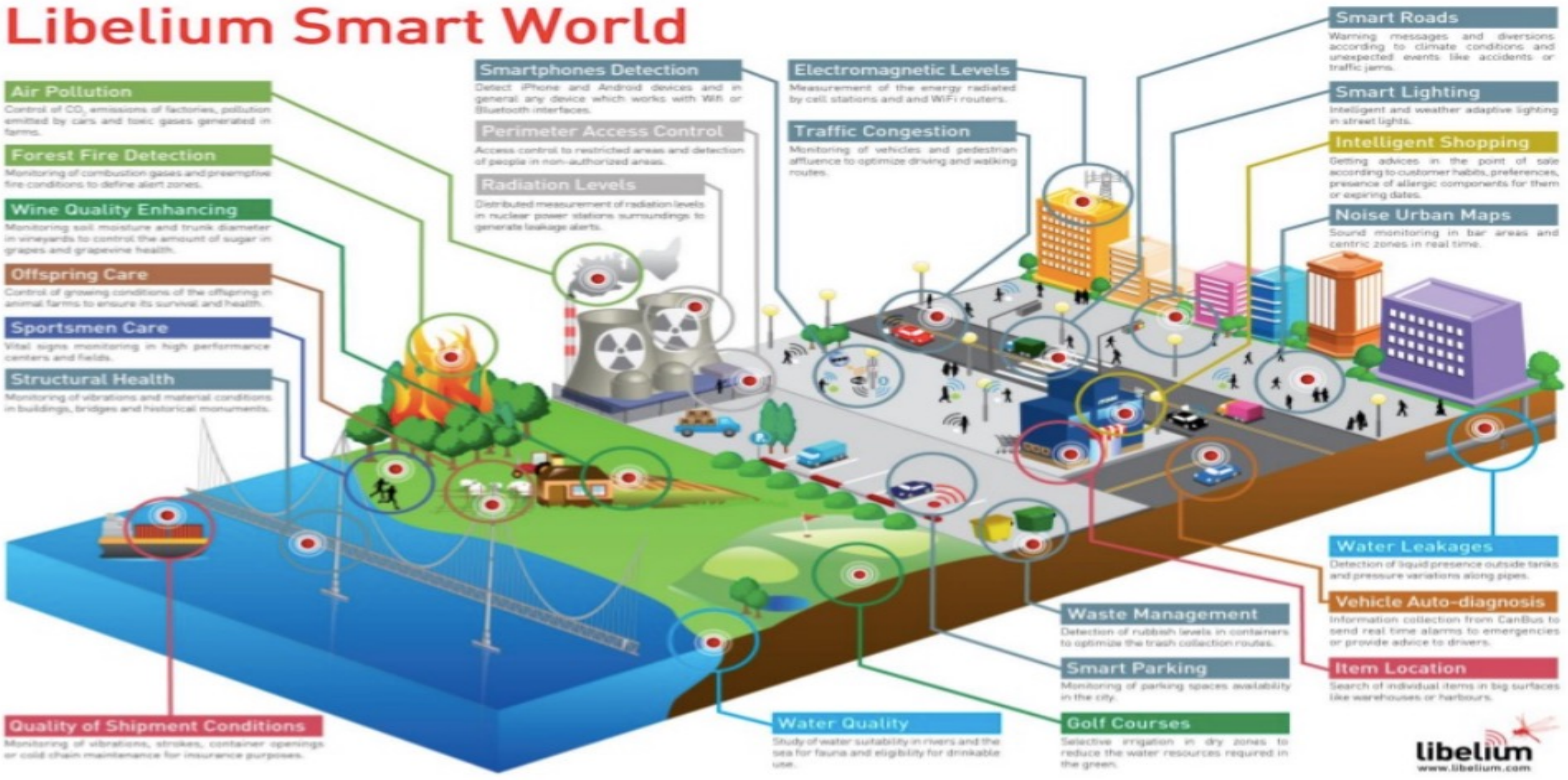
Edge & Fog computing

Thoai Nam

High Performance Computing Lab (HPC Lab)
Faculty of Computer Science and Engineering
HCMC University of Technology

Smart cities

Many applications/services
Many machines
Internet, Intranet: network



http://www.libelium.com/libelium-smart-world-infographic-smart-cities-internet-of-things/

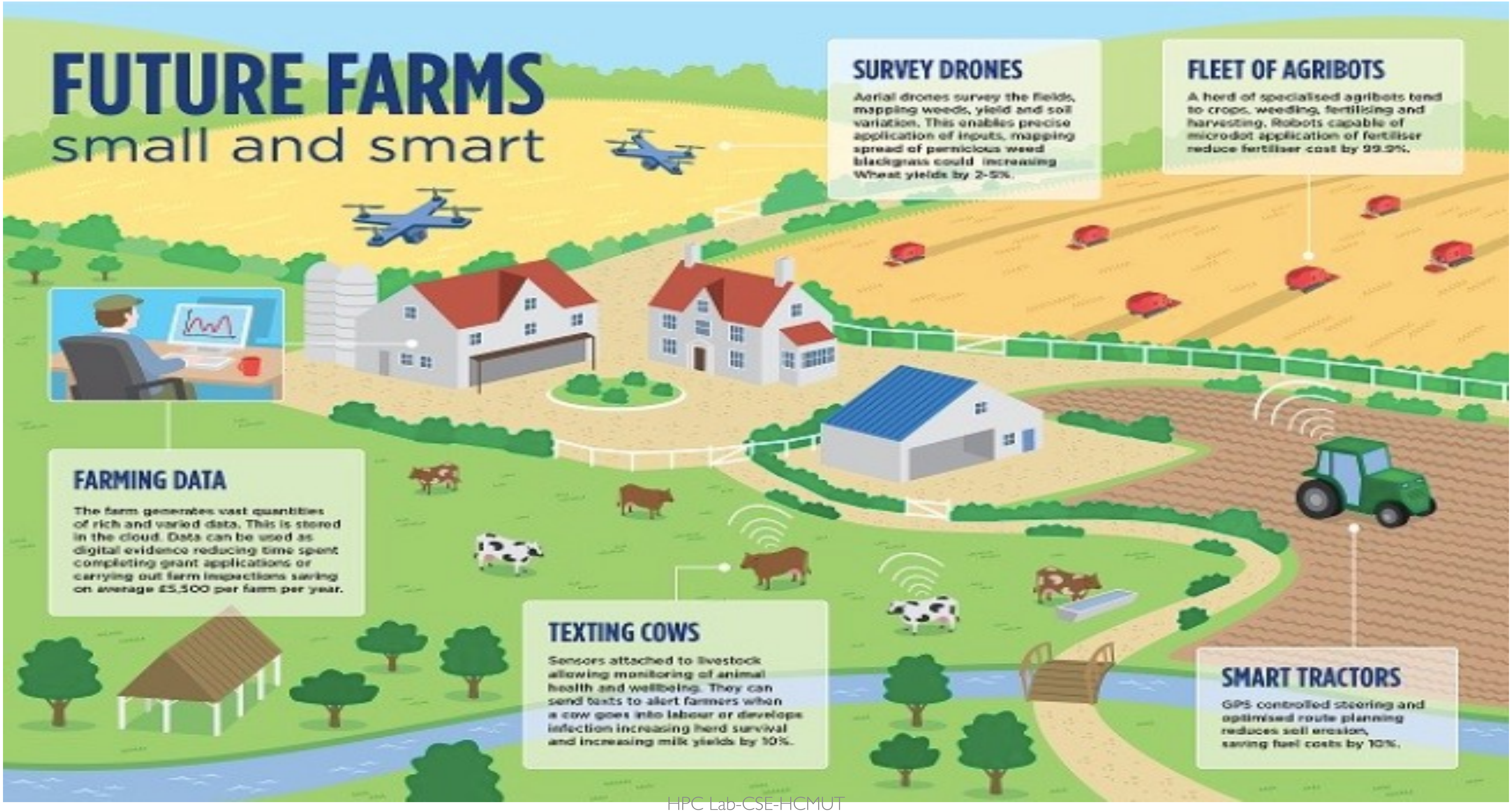
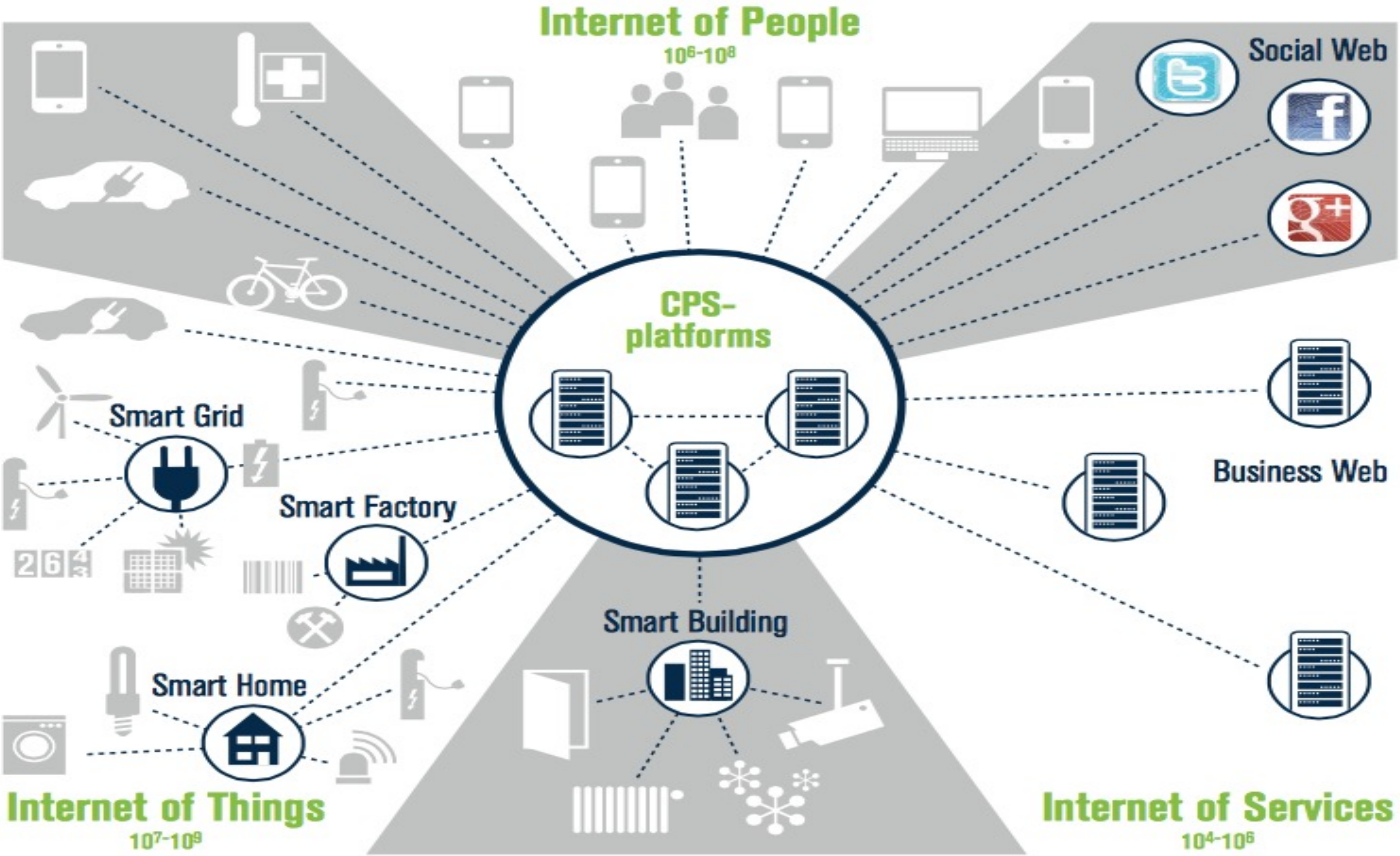
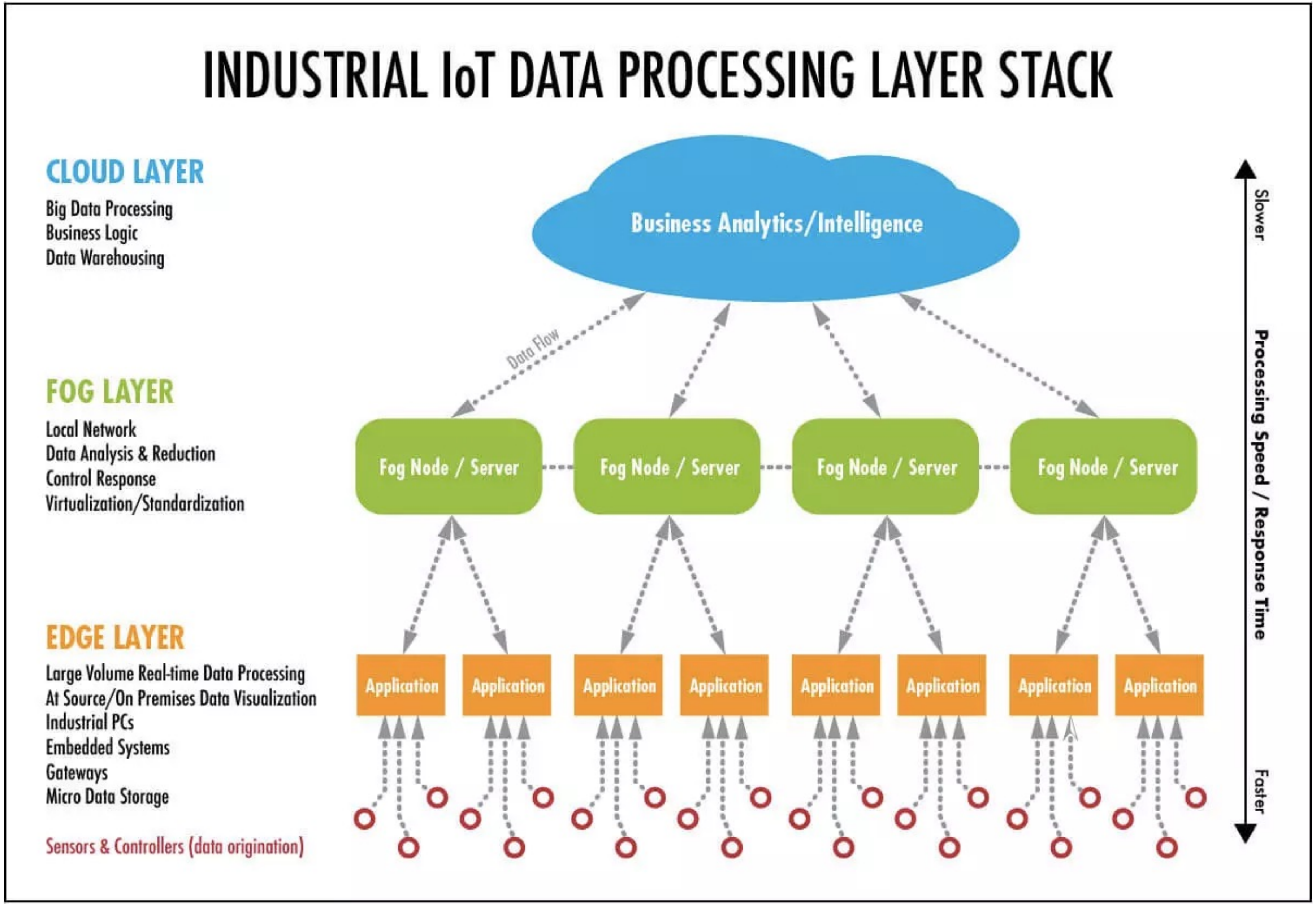

FUTURE FARMS
small and smart
SURVEY DRONES
Aerial drones survey the fields, mapping weeds, yield and soil variation. This enables precise application of inputs, mapping spread of pernicious weed blackgrass could increasing Wheat yields by 2-5%.
FLEET OF AGRIBOTS
A herd of specialised agribots tend to crops, weeding, fertilising and harvesting. Robots capable of microdot application of fertiliser reduce fertiliser cost by 99.9%.
FARMING DATA
The farm generates vast quantities of rich and varied data. This is stored in the cloud. Data can be used as digital evidence reducing time spent completing grant applications or carrying out farm inspections saving on average £5,500 per farm per year.
TEXTING COWS
Sensors attached to livestock allowing monitoring of animal health and wellbeing. They can send texts to alert farmers when a cow goes into labour or develops infection increasing herd survival and increasing milk yields by 10%.
SMART TRACTORS
GPS controlled steering and optimised route planning reduces soil erosion, saving fuel costs by 10%.

Figure 4:
The Internet of Things and Services – Networking people, objects and systems



Source: Bosch Software Innovations 2012



[Source: https://www.winsystems.com/cloud-fog-and-edge-computing-whats-the-difference/]

Edge computing

- Computation takes place at the edge of a device’s network, which is known as edge computing
- A computer is connected with the network of the device, which processes the data and sends the data to the cloud in real-time
- That computer is known as “edge computer” or “edge node”

- Data is processed and transmitted to the devices instantly
- Edge nodes transmit all the data captured or generated by the device regardless of the importance of the data.

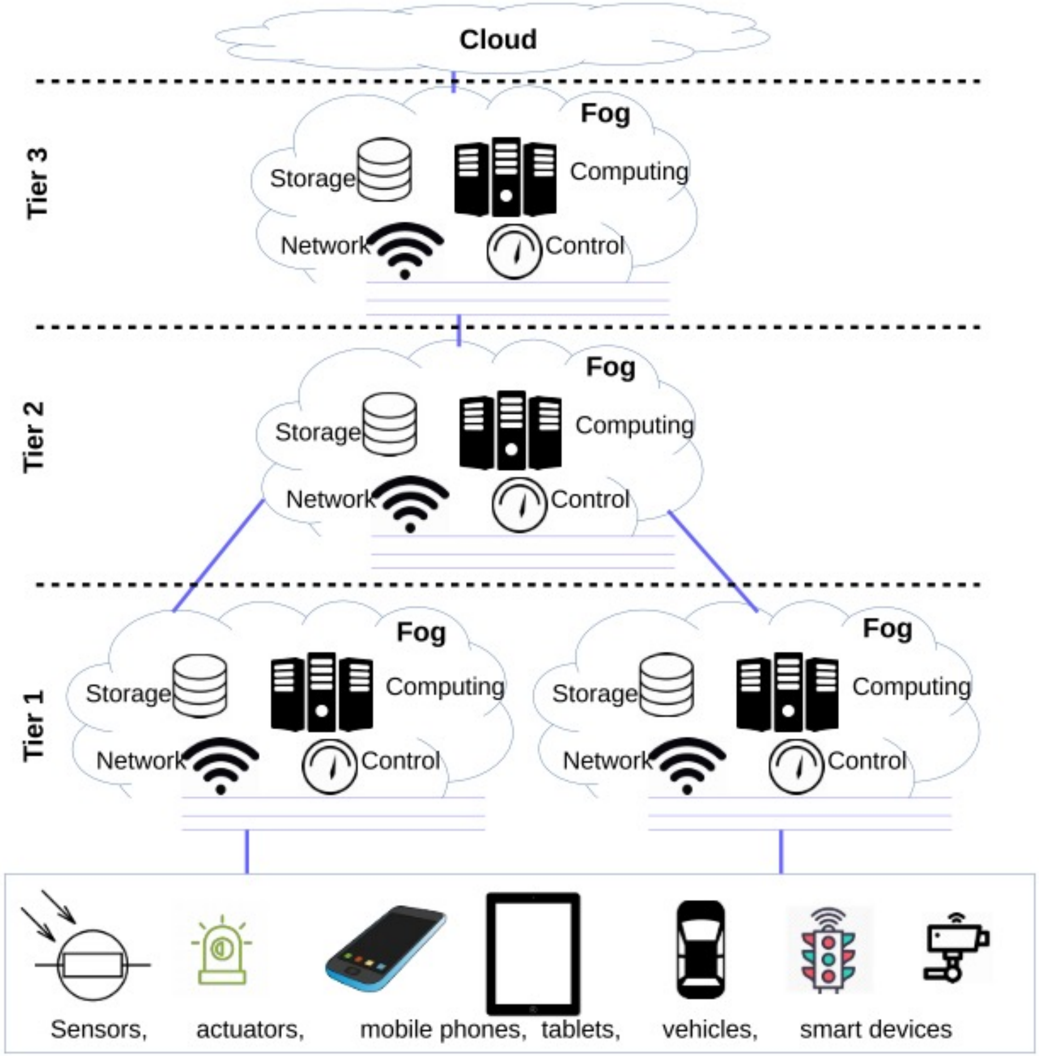
Fog computing

- Fog computing is an extension of cloud computing
- It is a layer in between the edge and the cloud

- When edge computers send huge amounts of data to the cloud, fog nodes receive the data and analyze what's important. Then the fog nodes transfer the important data to the cloud to be stored and delete the unimportant data or keep them with themselves for further analysis. In this way, fog computing saves a lot of space in the cloud and transfers important data quickly.

Fog (including Edge) computing

- Fog technology complements the role of cloud computing and distributes the data processing at the edge of the network, which provides faster responses to application queries and saves the network resources
- Fog computing model
  - Sensors
  - Actuators
  - Fog nodes at T1, T2, T3, etc. levels
  - Cloud
- Benefits of Fog computing
  - Move data to the best place for processing
  - Optimize latency
  - Conserve network bandwidth
  - Collect and secure data



Cloud is the centralized storage situated further from the endpoints than any other type of storage. This explains the highest latency, bandwidth cost, and network requirements. On the other hand, cloud is a powerful global solution that can handle huge amounts of data and scale effectively by engaging more computing resources and server space. It works great for big data analytics, long-term data storage and historical data analysis.

Fog acts as a middle layer between cloud and edge and provides the benefits of both. It relies on and works directly with the cloud handing out data that don't need to be processed on the go. At the same time, fog is placed closer to the edge. If necessary, it engages local computing and storage resources for real-time analytics and quick response to events.

Just like edge, fog is decentralized meaning that it consists of many nodes. However, unlike edge, fog has a network architecture. Fog nodes are connected with each other and can redistribute computing and storage to better solve given tasks.

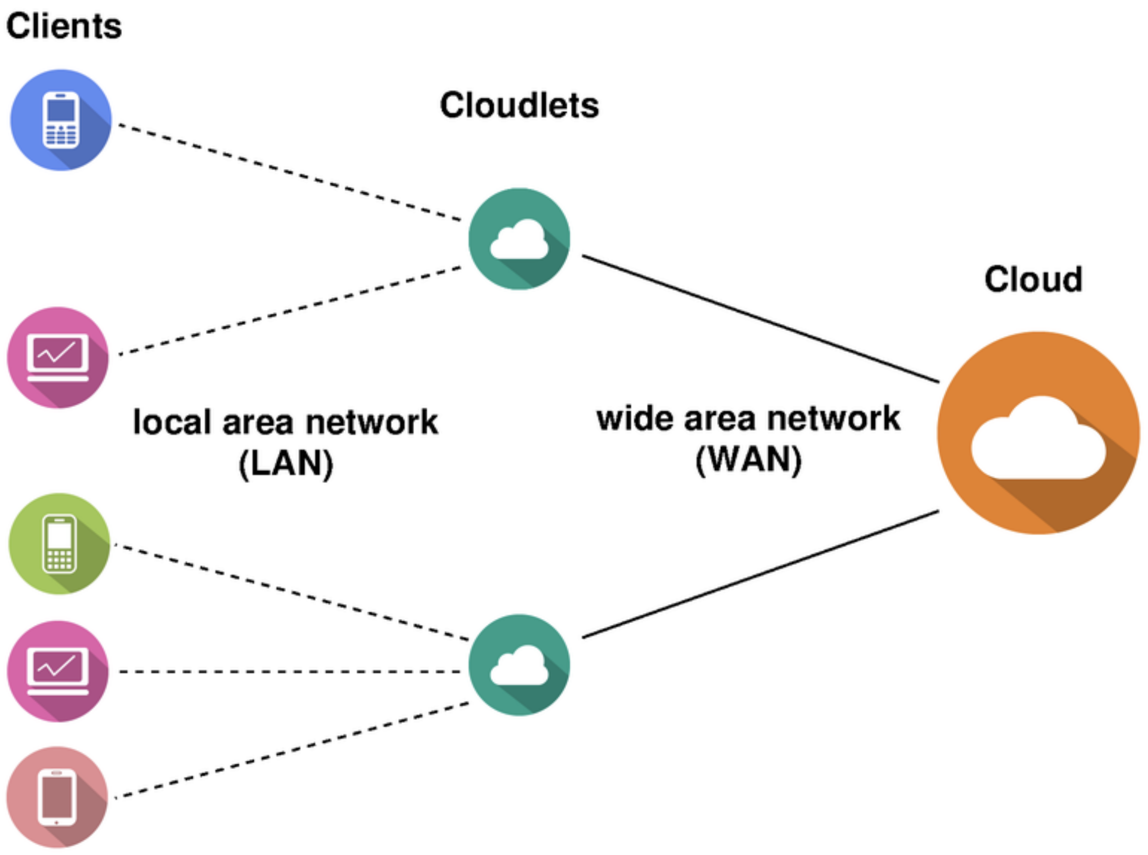
Edge is the closest you can get to end devices, hence the lowest latency and immediate response to data. This approach allows to perform computing and store some (only limited) volume of data directly on devices, applications and edge gateways. It usually has a loosely connected structure where edge nodes work with data independently. This is what differentiates edge from network-based fog.

Here's a cloud vs. fog vs. edge computing comparison chart that gives a quick overview of these and other differences between these approaches.

No.	Edge computing	Fog computing
1	Less scalable than fog computing.	Highly scalable when compared to edge computing.
2	Billions of nodes are present.	Millions of nodes are present.
3	Nodes are installed far away from the cloud.	Nodes in this computing are installed closer to the cloud(remote database where data is stored).
4	Edge computing is a subdivision of fog computing.	Fog computing is a subdivision of cloud computing.
5	The bandwidth requirement is very low. Because data comes from the edge nodes themselves.	The bandwidth requirement is high. Data originating from edge nodes is transferred to the cloud.
6	Operational cost is higher.	Operational cost is comparatively lower.
7	High privacy. Attacks on data are very low.	The probability of data attacks is higher.
8	Edge devices are the inclusion of the IoT devices or client's network.	Fog is an extended layer of cloud.
9	The power consumption of nodes is low.	The power consumption of nodes filter important information from the massive amount of data collected from the device and saves it in the filter high.
10	Edge computing helps devices to get faster results by processing the data simultaneously received from the devices.	Fog computing helps in filtering important information from the massive amount of data collected from the device and saves it in the cloud by sending the filtered data.

Cloudlet

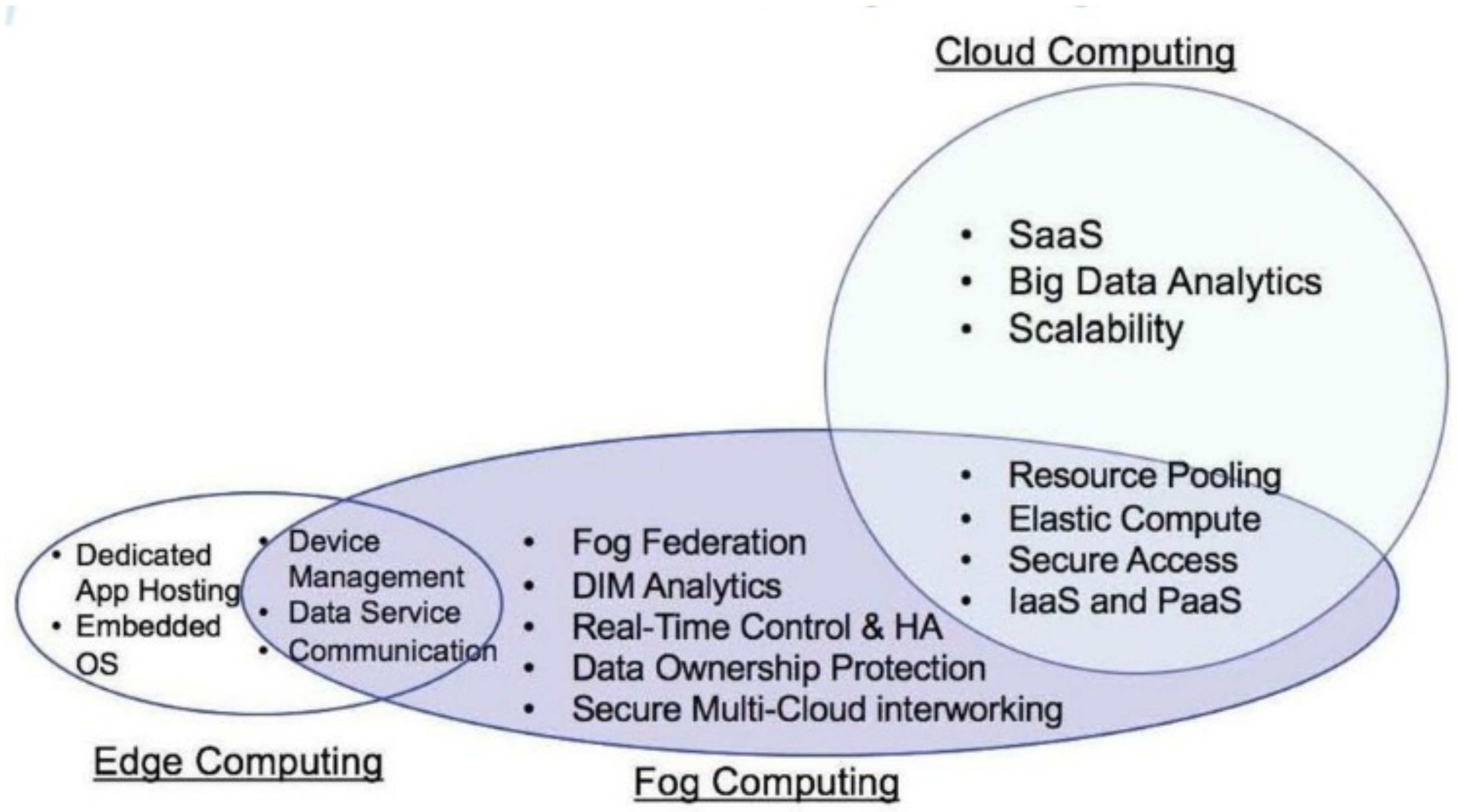
- A cloudlet is a mobility-enhanced small-scale cloud datacenter that is located at the edge of the Internet
- The main purpose of the cloudlet is supporting resource-intensive and interactive mobile applications by providing powerful computing resources to mobile devices with lower latency
- It is a new architectural element that extends today's cloud computing infrastructure
- It represents the middle tier of a 3-tier hierarchy: mobile device - cloudlet - cloud.

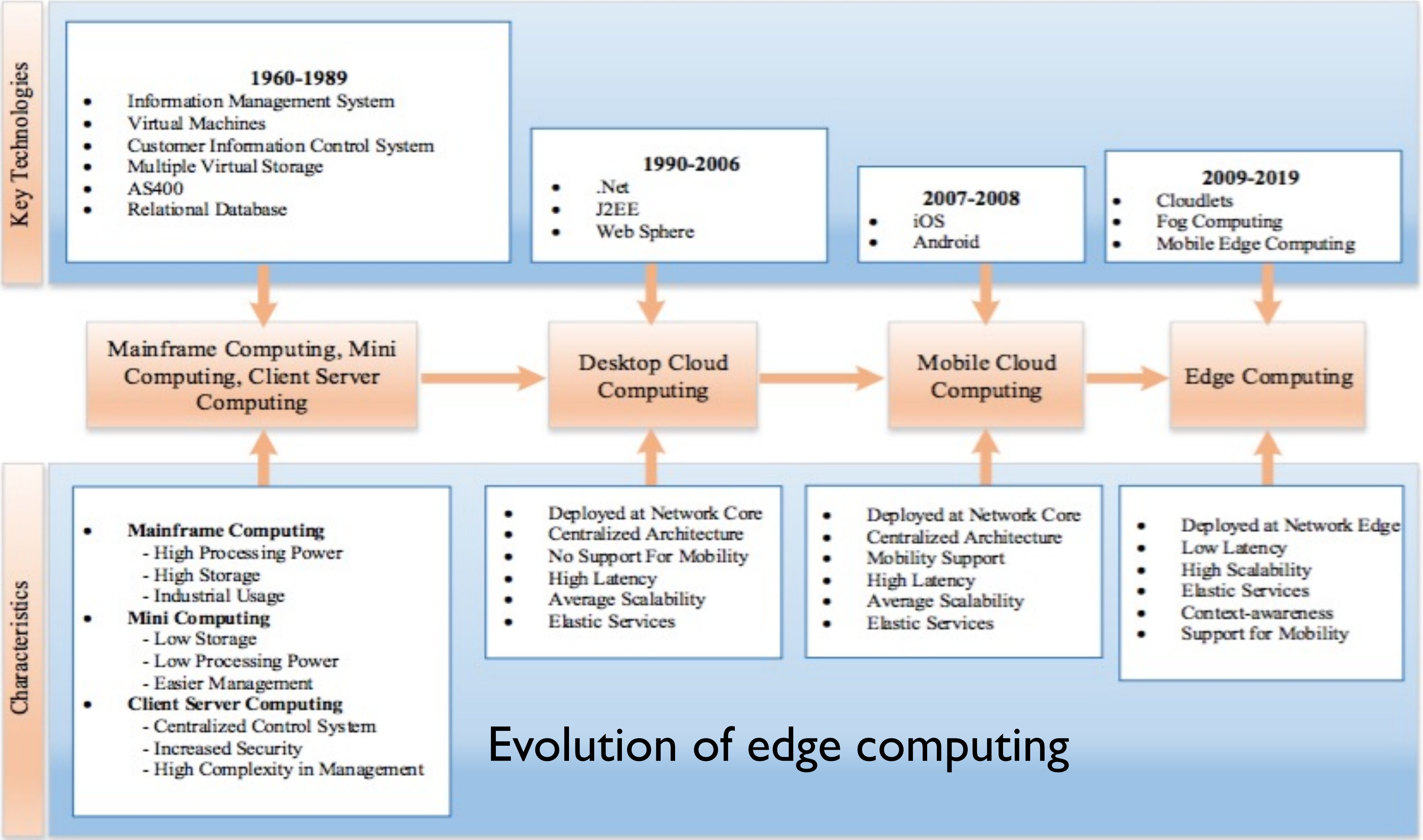


Edge computing paradigms comparison

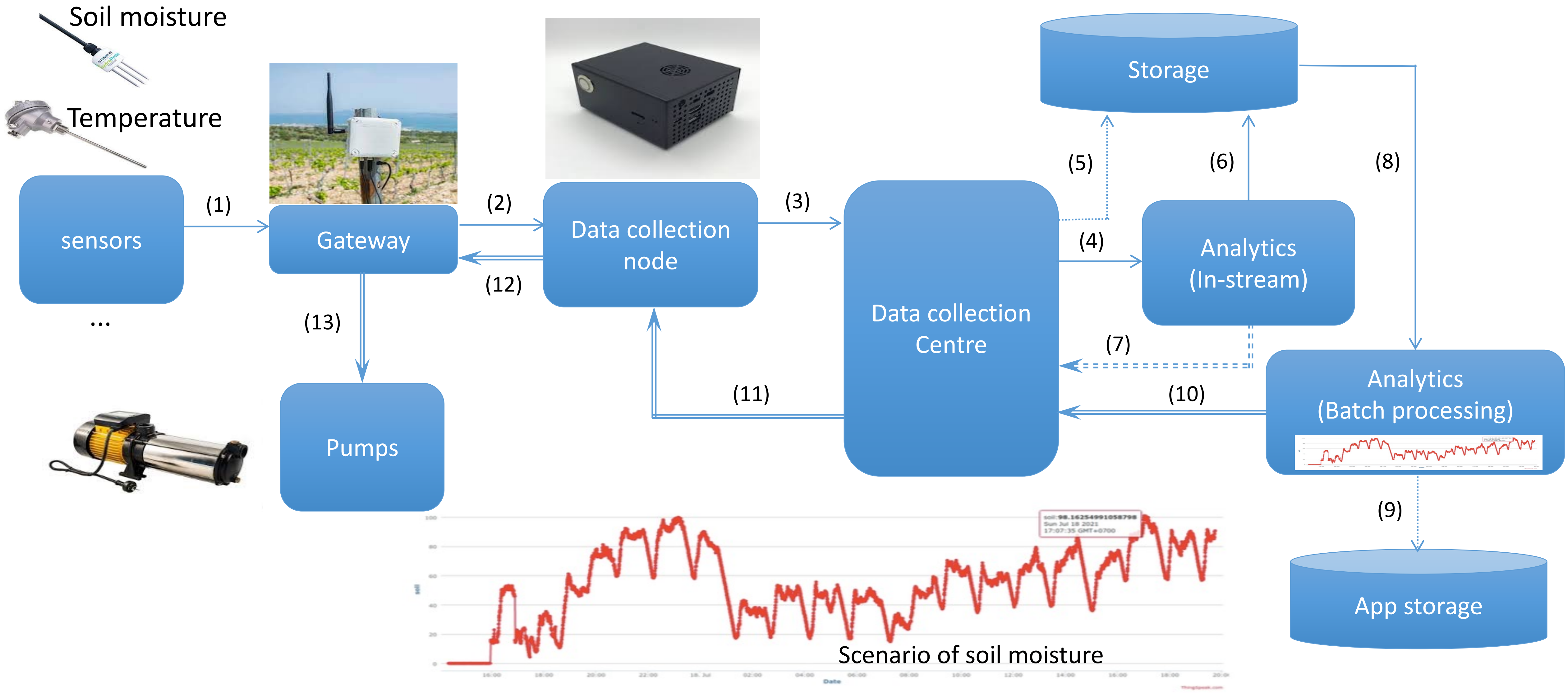
	Cloud computing	Cloudlets	Fog computing	Mobile edge computing
Context-awareness	No	Low	Medium	High
Geo-distribution	Centralized	Distributed	Distributed	Distributed
Latency	High	Low	Low	Low
Mobility support	No/Limited	Yes	Yes	Yes
Distance	Multi hop	Single hop	Single hop/ Multi hop	Single hop
Scalability	Yes	Yes	Yes	Yes
Flexibility	Yes	Yes	Yes	Yes
Deployment cost	High	Low	Low	High

From cloud to frog to edge





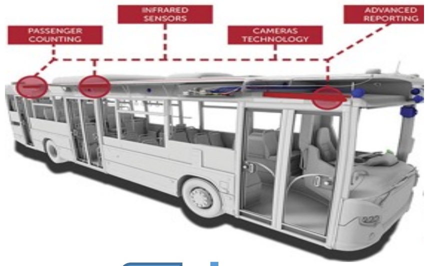
Evolution of edge computing



Smart irrigation

Challenges:

- *Low latency and location awareness*
- *Wide-spread geographical distribution*
- *Very large number of nodes*
- *Predominant role of wireless access*
- *Strong presence of streaming*
- *Real time applications*
- *Heterogeneity*
- *Mobility*

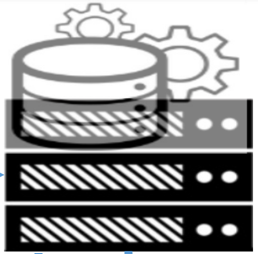


GPS Positioning data
Passenger counting
Camera videos

Cloud
- big data analytics
- long-term storage

Data

Pre-processing



Data

Video processing

Data

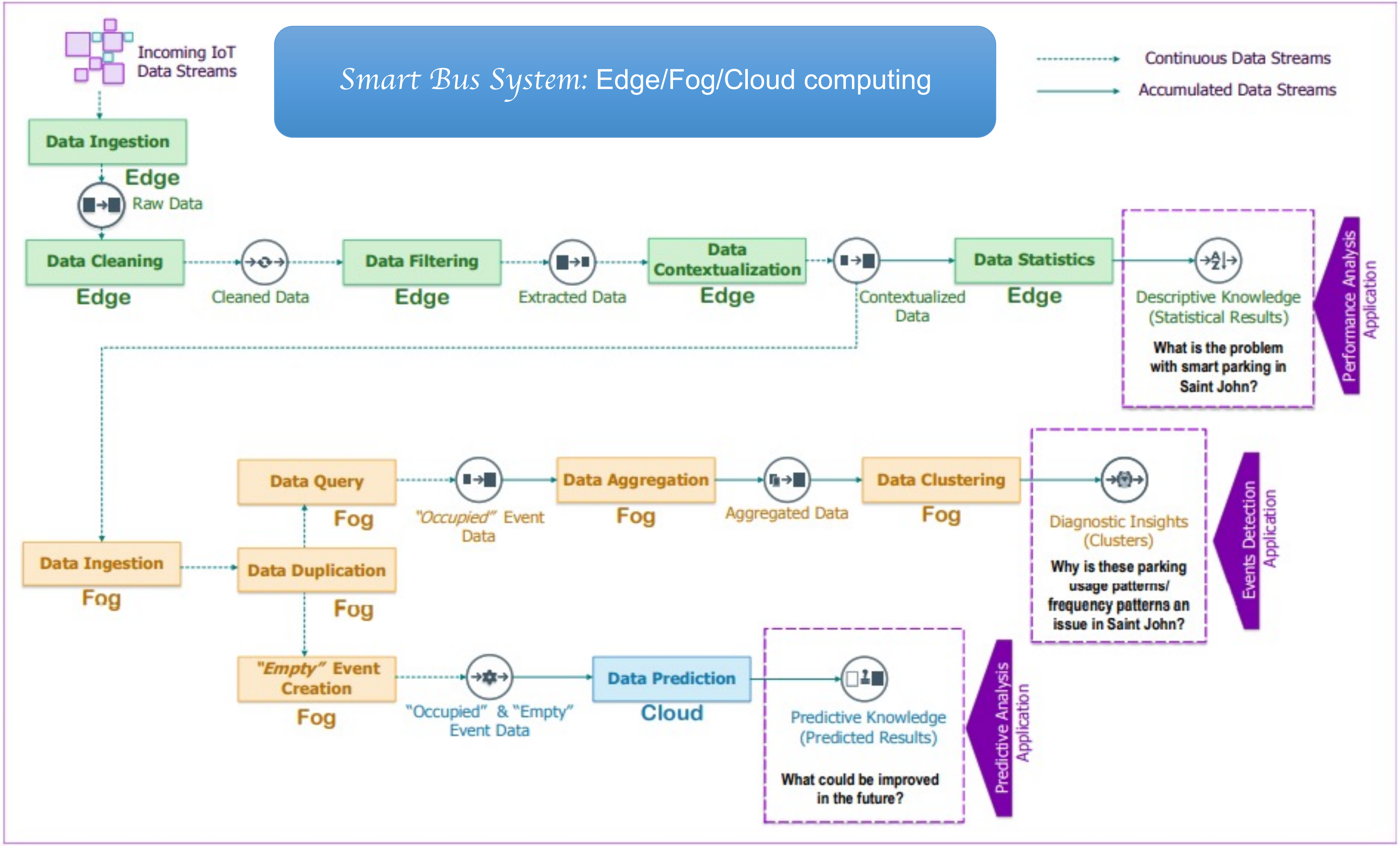
Complex data processing

Data

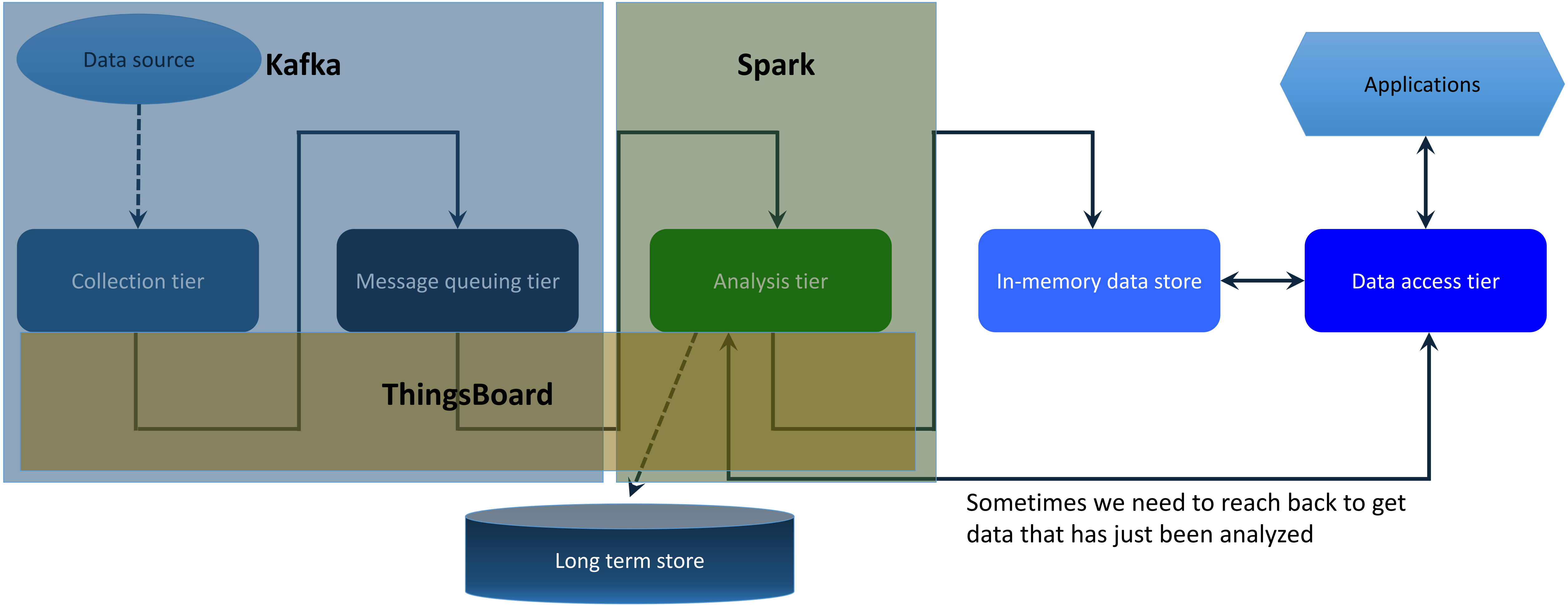
Data

Data

Bus number
Bus position
Available seats


Incoming IoT Data Streams
Smart Bus System: Edge/Fog/Cloud computing
Continuous Data Streams
Accumulated Data Streams
Data Ingestion
Edge
Raw Data
Data Cleaning
Edge
Cleaned Data
Data Filtering
Edge
Extracted Data
Data Contextualization
Edge
Contextualized Data
Data Statistics
Edge
Descriptive Knowledge (Statistical Results)
What is the problem with smart parking in Saint John?
Performance Analysis Application
Data Query
Fog
"Occupied" Event Data
Data Aggregation
Fog
Aggregated Data
Data Clustering
Fog
Diagnostic Insights (Clusters)
Why is these parking usage patterns/ frequency patterns an issue in Saint John?
Events Detection Application
Data Ingestion
Fog
Data Duplication
Fog
"Empty" Event Creation
Fog
"Occupied" & "Empty" Event Data
Data Prediction
Cloud
Predictive Knowledge (Predicted Results)
What could be improved in the future?
Predictive Analysis Application

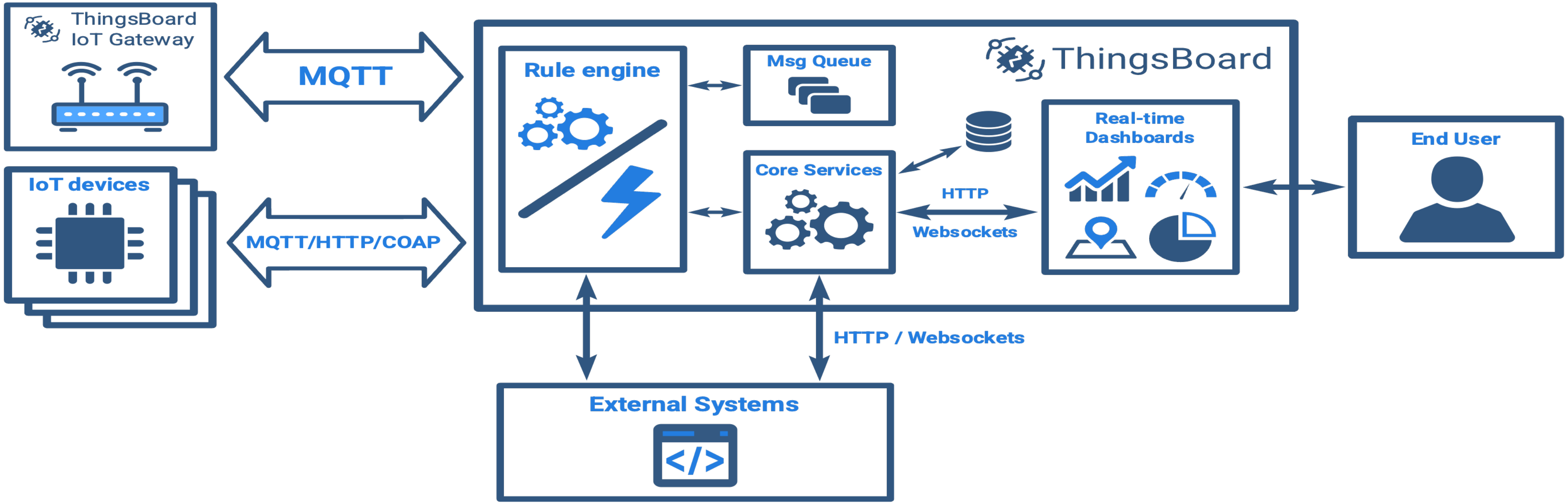
Data streaming architecture: Pub/Sub



Sometimes we need to reach back to get data that has just been analyzed

We want to persist analyzed data for future use

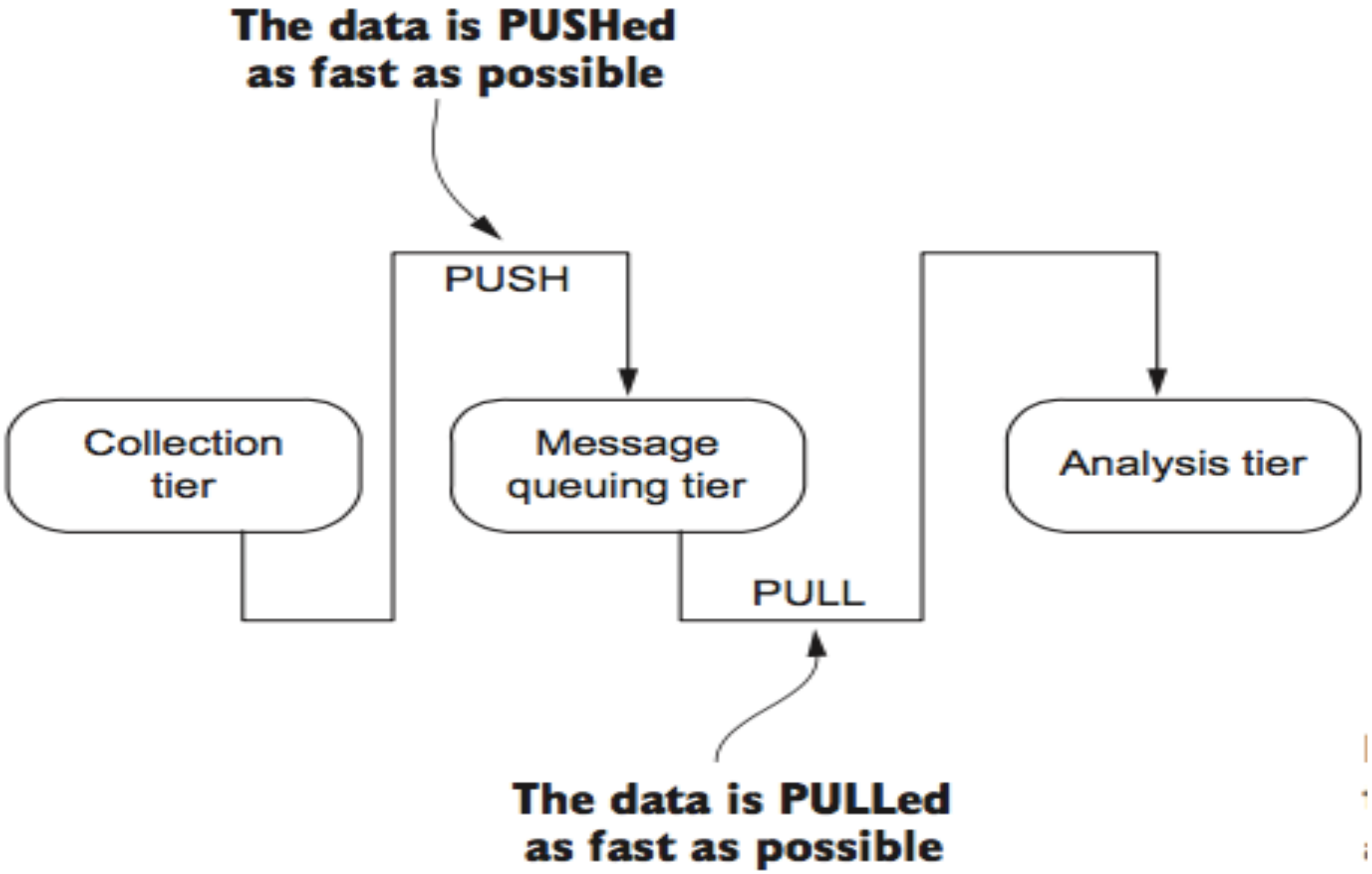
ThingsBoard



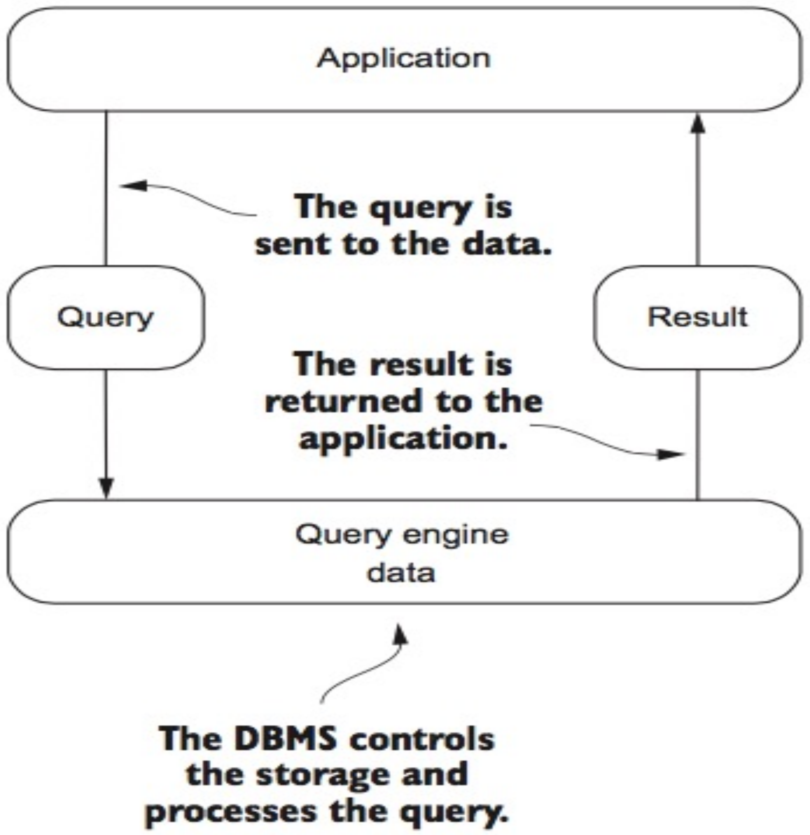
- Collection data node
- Support many protocols
- Rule engine
  - Pre-processing: sampling, filtering, integration
- Message queue + etc.

=> A small scale solution (do not need Kafka + Spark)

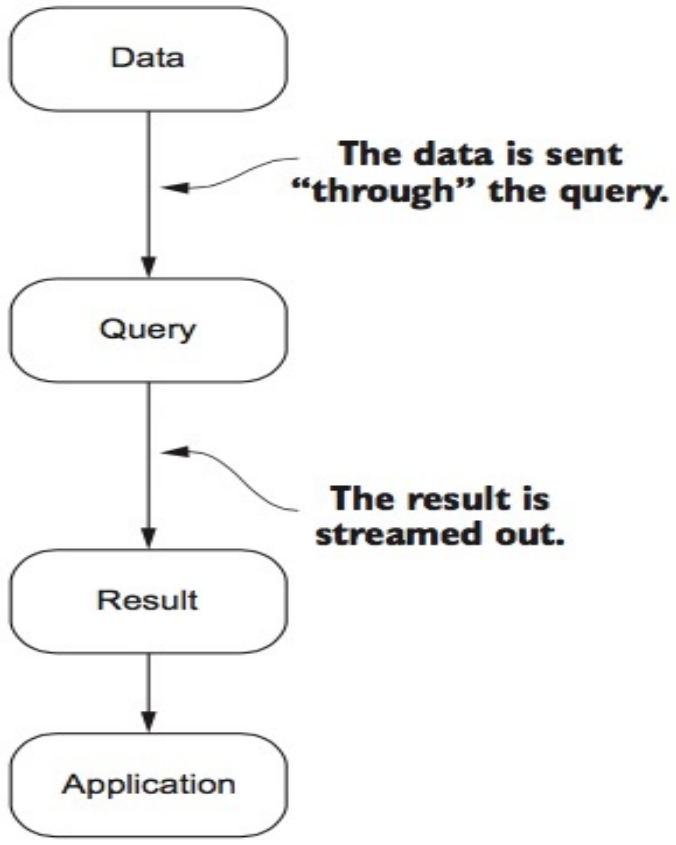
PUSH/PULL


The data is PUSHed
as fast as possible
PUSH
Collection
tier
Message
queuing tier
Analysis tier
PULL
The data is PULLed
as fast as possible

Non-streaming & streaming system



Traditional DBMS



Streaming System

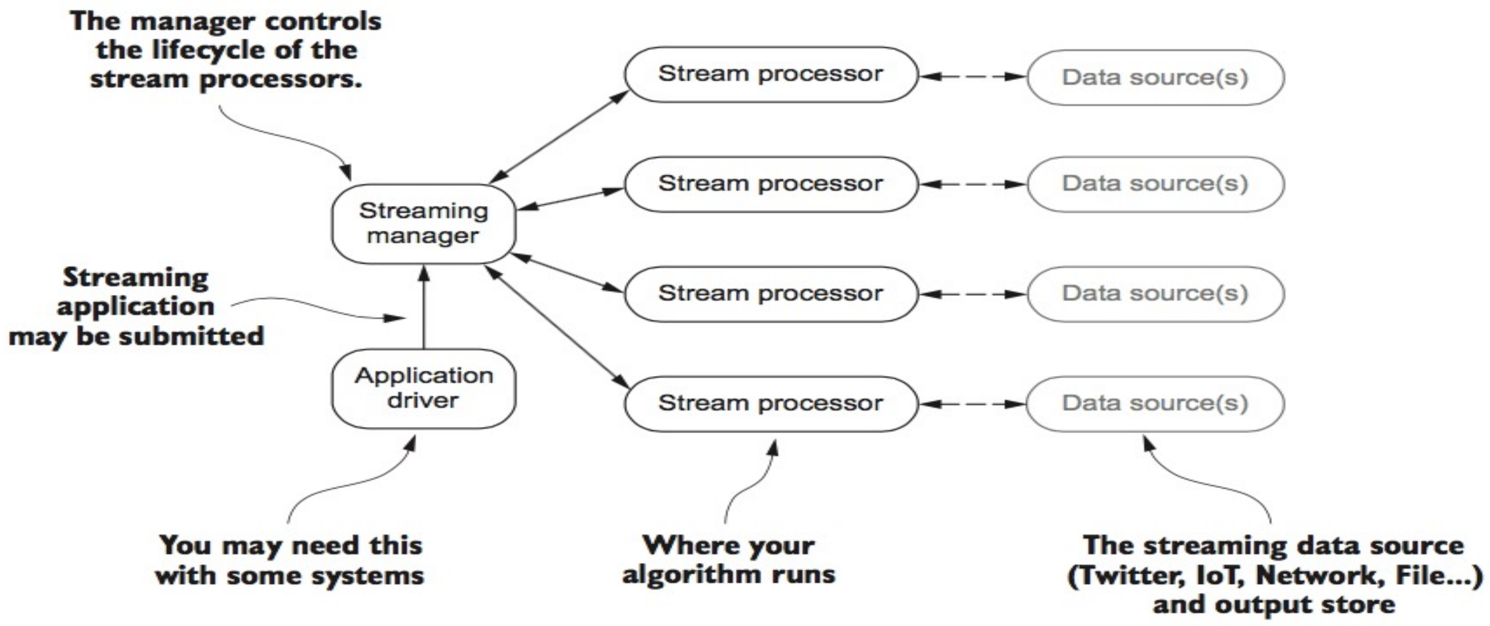
- A traditional DBMS (RDBMS, Hadoop, HBase, Cassandra, and so on):
  - In those non-streaming systems the data is at rest, and we query it for answers
- A streaming system:
  - *In-flight data:* The data is moved through the query
  - *Continuous query model*: the query is constantly being evaluated as new data arrive

Comparison of traditional DBMS to streaming system

	DBMS	Streaming system
Query model	Queries are based on a one-time model and a consistent state of the data. In a one-time model, the user executes a query and gets an answer, and the query is forgotten. This is a pull model.	The query is continuously executed based on the data that is flowing into the system. A user registers a query once, and the results are regularly pushed to the client.
Changing data	During down time, the data cannot change.	Many stream applications continue to generate data while the streaming analysis tier is down, possibly requiring a catch-up following a crash.
Query state	If the system crashes while a query is being executed, it is forgotten. It is the responsibility of the application (or user) to re-issue the query when the system comes back up.	Registered continuous queries may or may not need to continue where they left off. Many times it is as if they never stopped in the first place.

Distributed stream-processing architecture (1)

- Tools: (Apache) Spark Streaming, Storm, Flink, and Samza
  - A component that your streaming application is submitted to; this is similar to how Hadoop Map Reduce works. Your application is sent to a node in the cluster that executes your application
  - Separate nodes in the cluster execute your streaming algorithms
  - Data sources are the input to the streaming algorithms

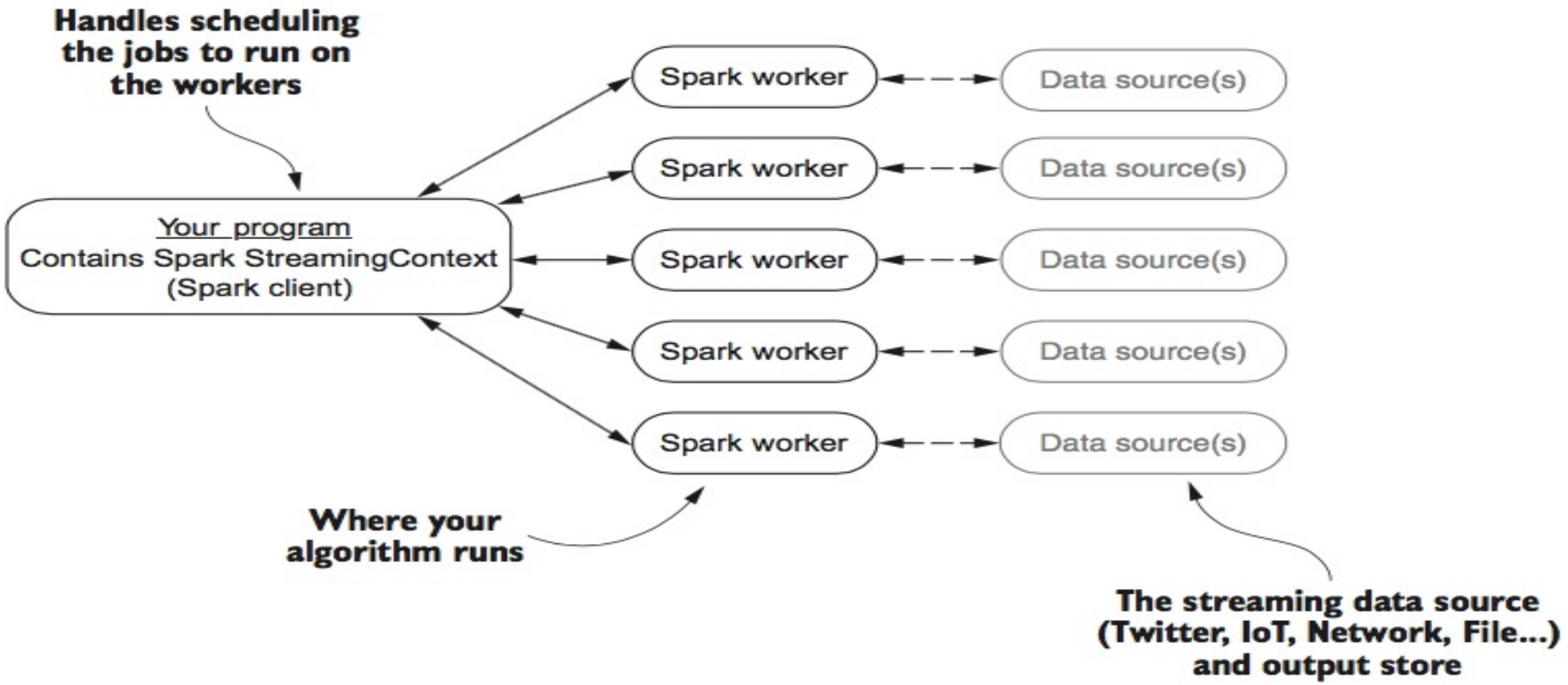


Distributed stream-processing architecture (2)

- *Application driver*: with some streaming systems, this will be the client code that defines your streaming programming and communicates with the streaming manager
- *Streaming manager*: the streaming manager has the general responsibility of getting your streaming job to the stream processor(s); in some cases it will control or request the resources required by the stream processors
- *Stream processor*: the place where your job runs; although this may take many shapes based on the streaming platform in use, the job remains the same: to execute the job that was submitted
- *Data source(s)*: This represents the input and potentially the output data from your streaming job. With some platforms your job may be able to ingest data from multiple sources in a single job, whereas others may only allow ingestion from a single source.

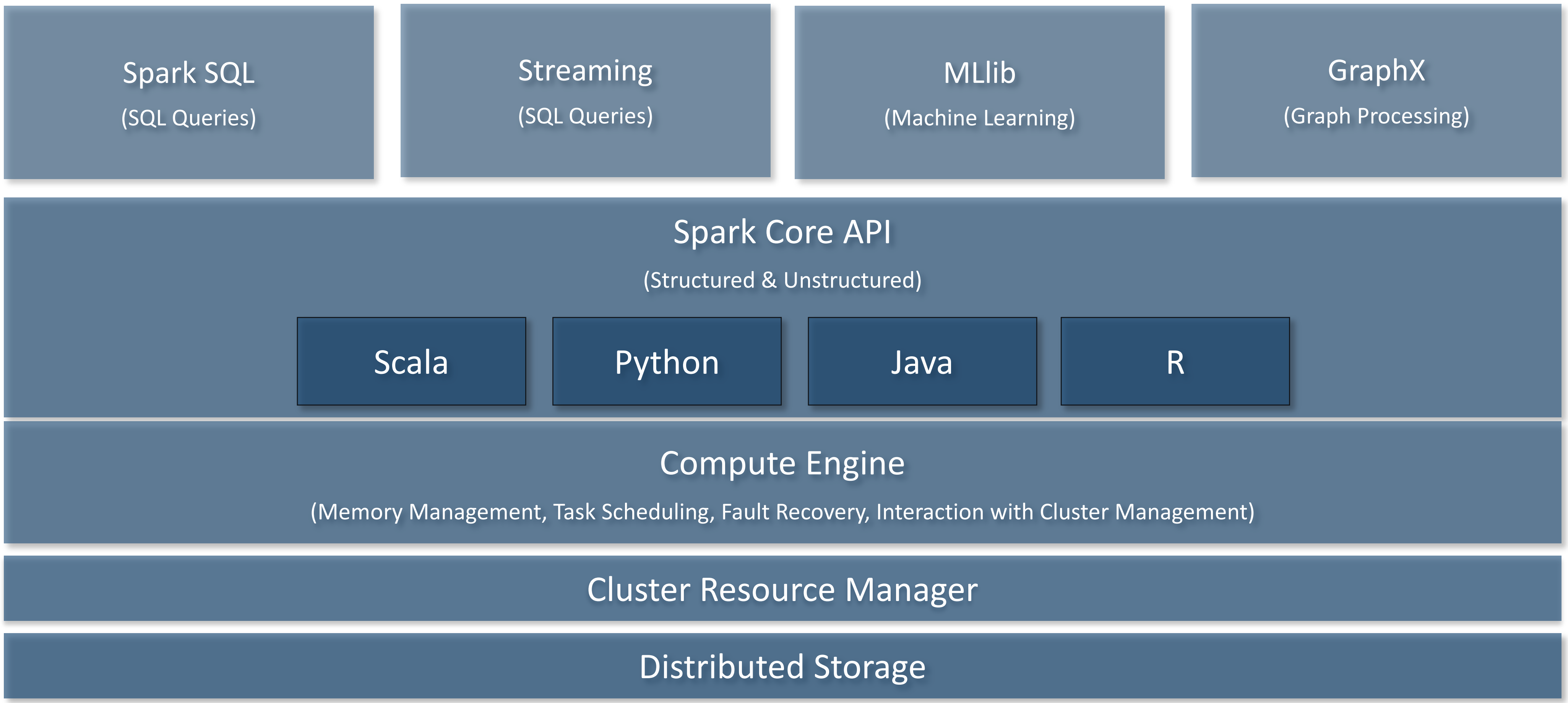
Apache Spark

- The de facto platform for general-purpose distributed computation
- Programming languages: Java, Scala, Python, and R
- Modules:
    - Spark Streaming
    - MLlib (machine learning)
    - SparkR (integration with R)
    - GraphX (for graph processing)



- Spark StreamingContext is the driver
- A *job* in Spark Streaming is the logic of your program that's bundled and passed to the Spark workers
- Spark workers: which run on any number of computers (from one to thousands) and are where your job (your streaming algorithm) is executed. They receive data from an external data source and communicate with the Spark StreamingContext that's running as part of the driver.

Apache Spark



Data analytics

Message delivery semantics

- *At-most-once* - *a* message may get lost, but it will never be processed a second time => **Simple**
- *At-least-once* - *a* message will never be lost, but it may be processed more than once
  - If every time the streaming job receives the same message, it produces the same result => the duplicate-messages situation
- *Exactly-once* - *a* message is never lost and will be processed only once
  - Detect and ignore duplicates

Algorithms for (streaming) data analytics

Streaming data queries

- *Ad-hoc queries* - These are queries asked one time about a stream
  - Ex: What is the maximum value seen so far in the stream? This style of query is the same kind you would execute against an RDBMS
- *Continuous queries:* These are queries that are, in essence, asked about the stream at all times
  - Ex: Determine the maximum value ever seen in the stream emitted every five minutes and generate an alert if it exceeds a given threshold

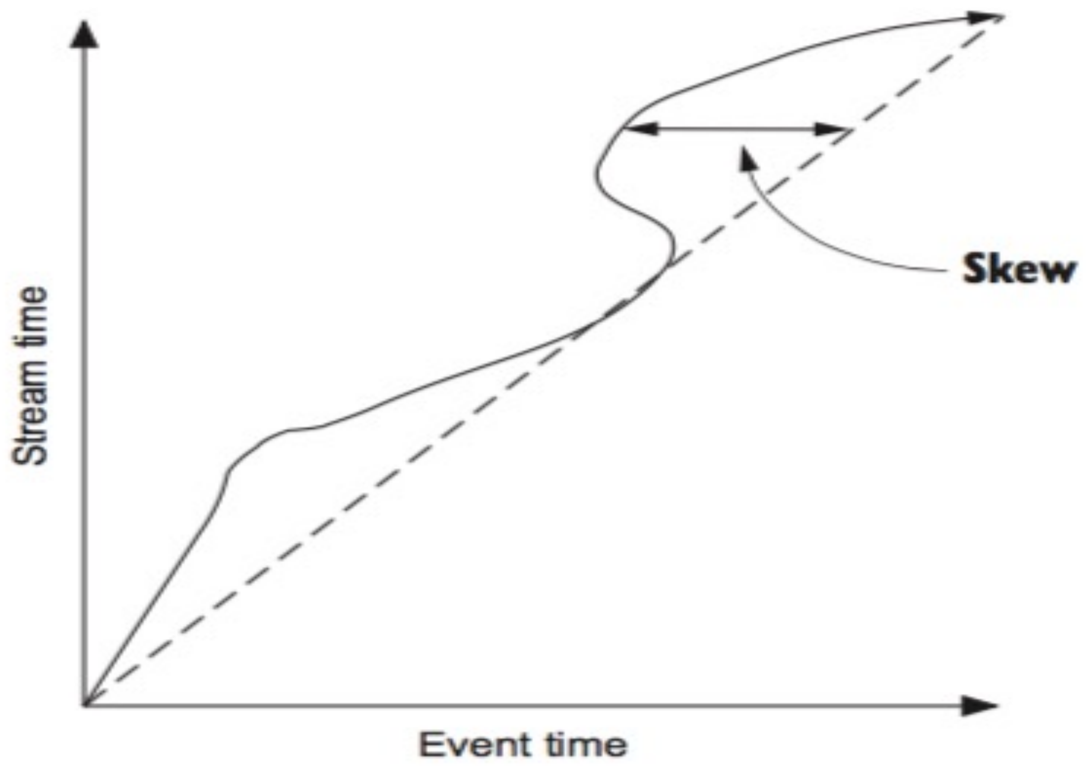
Product	Query language support
Apache Storm	As of version 1.1.0 Apache Storm has had SQL support (http://storm.apache.org/releases/1.1.0/storm-sql.html). As of this writing it is still considered experimental and not ready for production use
Apache Samza	Since version 0.9 of Apache Samza there has been a JIRA open for adding query language support. As of this writing, that JIRA is still open, and Samza does not have any query language support: https://issues.apache.org/jira/browse/SAMZA-390
Apache Flink	Table API supporting SQL-like expressions (http://ci.apache.org/projects/ flink/flink-docs-release-0.9/libs/table.html)
Apache Spark Streaming	SparkSQL/Hive language support (http://spark.apache.org/docs/latest/ sql-programming-guide.html)

Constrains and relaxing

- *One-pass*
  - We must assume that the data is not being archived and that we only have one chance to process it
  - Many traditional data-mining algorithms are iterative and require multiple passes over the data
- *Concept drift*
  - This is a phenomenon that may impact your predictive models. Concept drift may happen over time as your data evolves and various statistical properties of it change
- *Resource constraints*
  - A temporary peak in the data speed or volume => an algorithm may have to drop tuples that can't be processed in time, called *load shedding*
- *Domain constraints*
  - huge collected data => challenges in analytics.

Stream time and event time

- *Stream* time is the time at which an event enters the streaming system: $T_{stream}(e)$
- *Event* time is the time at which the event occurs: $T_{event}(e)$
- $T_{stream}(e) > T_{event}(e)$
- *Time skew* = $T_{stream}(e) - T_{event}(e)$

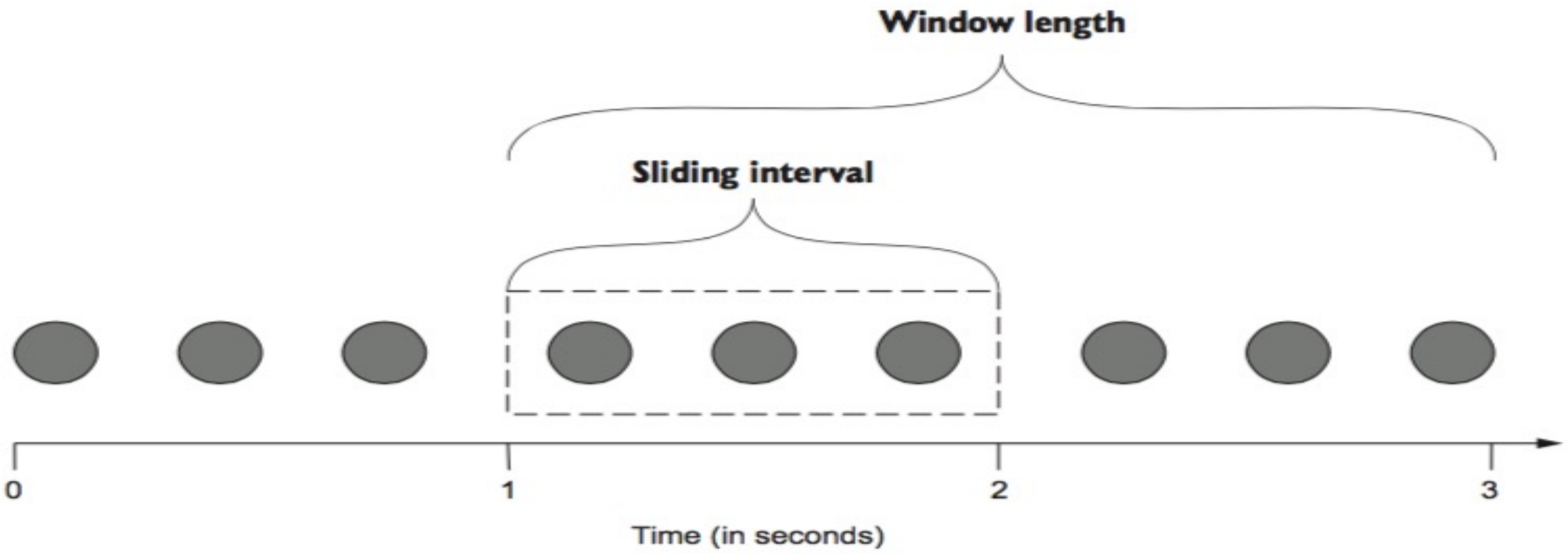


Windows of time

- Due to its size and never-ending nature, the stream processing engine can't keep an entire stream of data in memory
  - Cannot perform traditional batch processing on it
- A *window* of data represents a certain amount of data that we can perform computations on
  - The *trigger* policy defines the rules a stream-processing system uses to notify our code that it's time to process all the data that is in the window
  - The *eviction* policy defines the rules used to decide if a data element should be evicted from the window
  - Both polices are driven by either time or the quantity of data in the window.

Sliding window

- The *sliding* window technique uses eviction and trigger policies that are based on time
- The window length represents the eviction policy—the duration of time that data is retained and available for processing
- The sliding interval defines the trigger policy

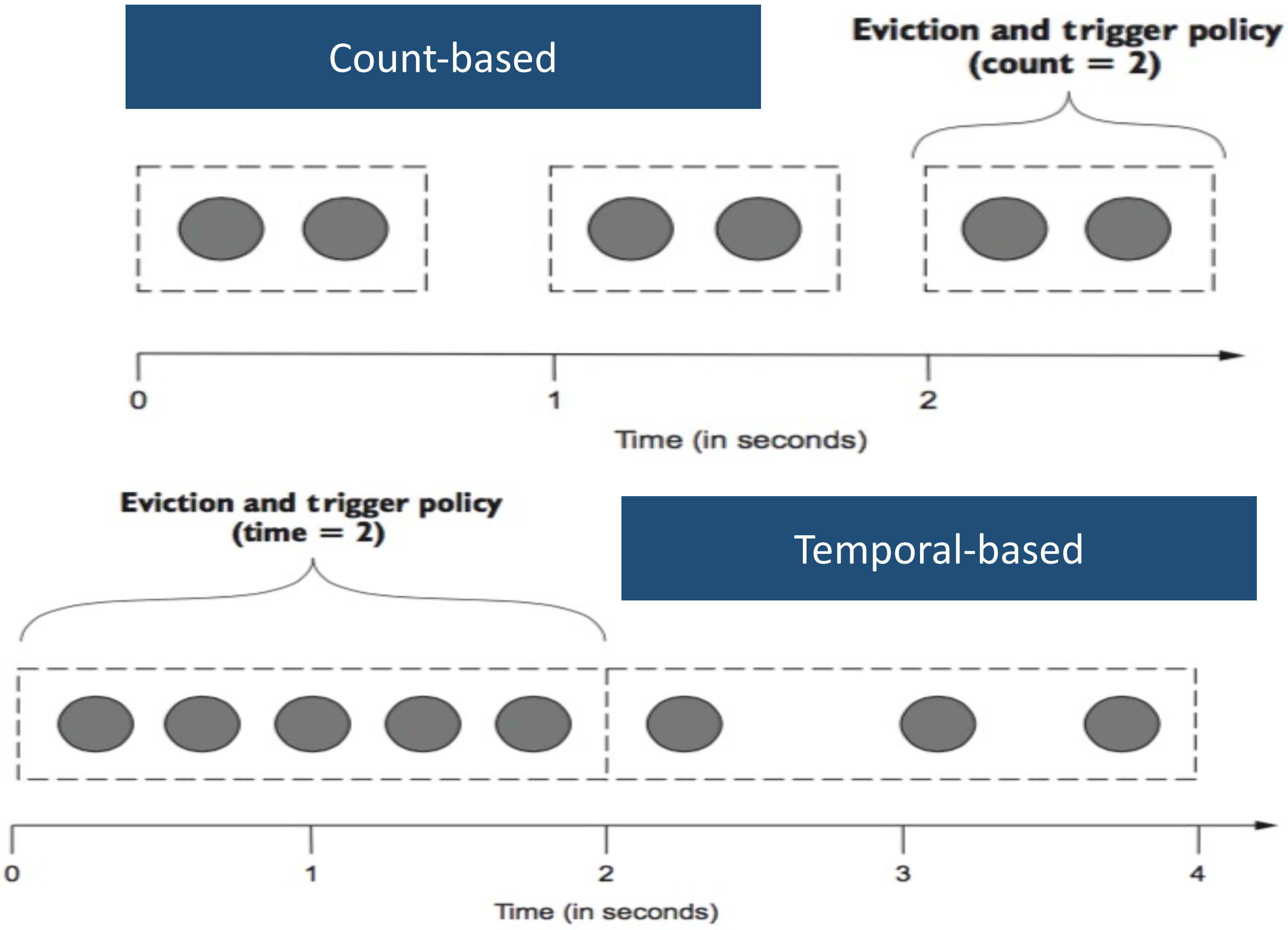


Sliding window support in popular stream-processing frameworks

Framework	Sliding window	Event or stream time	Comments
Spark Streaming	Yes	Stream time	Spark Streaming doesn't allow custom policies.
Storm	No	N/A	Storm doesn't provide native support for sliding windowing, but it could be implemented using timers.
Flink	Yes	Both	Flink allows a user to define a custom policy and trigger policies.
Samza	No	N/A	Samza doesn't provide direct support for sliding windows.

Tumbling window

- The eviction policy is always based on the window being full
- The trigger policy is based on either the count of items in the window or time
  - Count-based
  - Temporal-based



Tumbling window support in popular stream-processing frameworks

Framework	Count	Temporal	Comments
Spark Streaming	No	No	Currently you would need to build this.
Storm	Yes	Yes	Although Storm does not have the native windowing support, we can easily implement this.
Flink	Yes	Yes	Flink has built-in support for both types of tumbling windows.
Samza	No	Yes	Samza does not provide direct support for sliding windows.

Algorithms

- Sampling
  - Since we cannot store the entire stream, one obvious approach is to store a sample
- Membership
  - Has this stream element ever occurred in the stream before?
- Frequency
  - How many times has stream element X occurred?
- Counting distinct elements
  - Count the distinct items in a stream, but remember we are con- strained by memory and don’t have the luxury of storing the entire stream

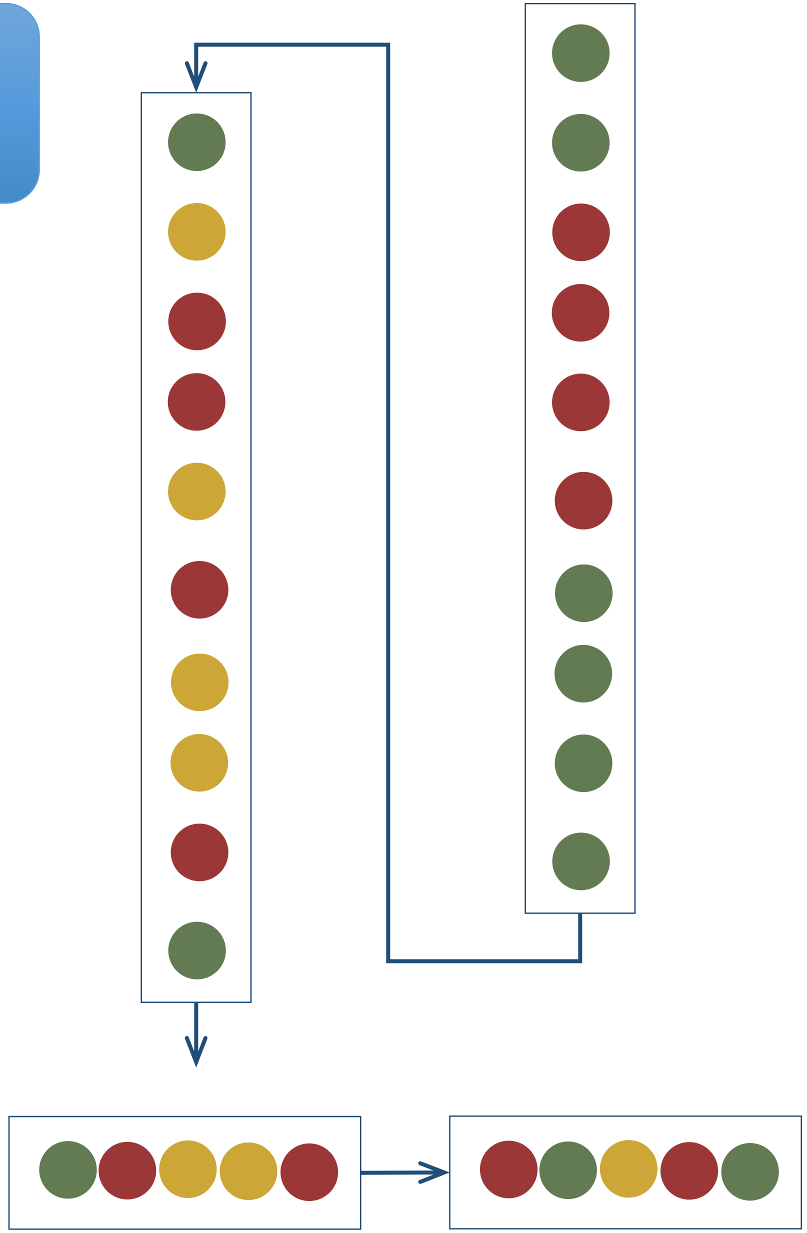
Sampling

Distributed (big data) Sampling

- Fixed-proportion
  - Sample a fixed proportion of elements in the stream (say 1 in 10)
  - Use a hash function that hashes keys of tuples uniformly into 10 buckets
- Fixed-size
  - Maintain a random sample of fixed size over a potentially infinite stream
  - Reservoir sampling

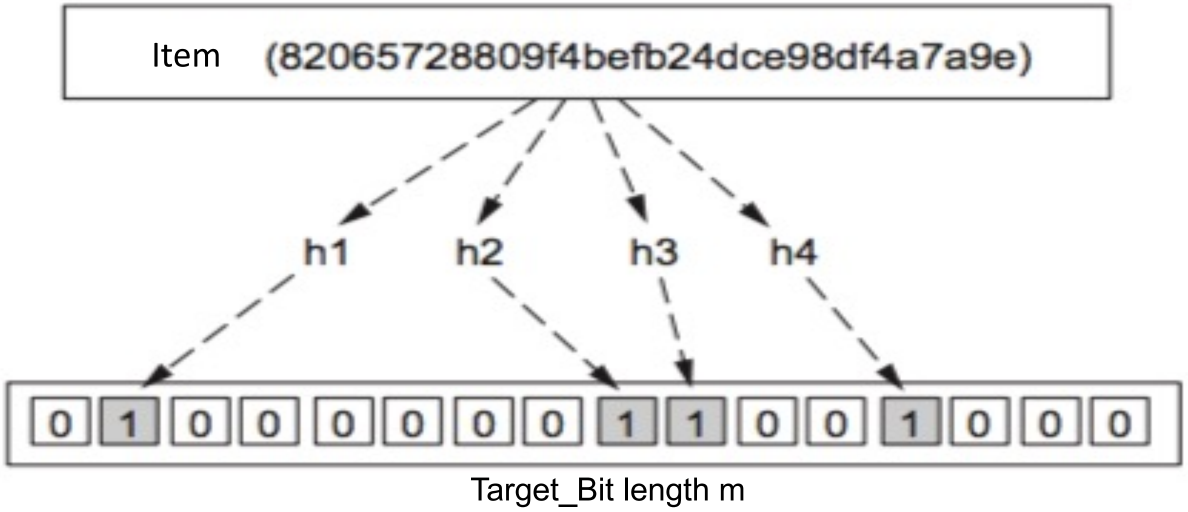
Reservoir sampling

- Store all the first ***s*** elements of the stream to ***S***
- Suppose we have seen ***n-1*** elements, and now the ***n****th* element arrives (***n*** > ***s***)
  - With probability ***s/n***, keep the ***n***th element, else discard it
  - If we picked the ***n***th element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random



Membership

- Has this stream element ever occurred in the stream before?
- Bloom filtering
  - a binary bit array of length *m* (Target_Bit) and is associated with a set of *k* independent hash functions
  - all item *x* and with all hash functions: bit [$h_i(x)$] = 1
  - *MEMBERSHIP of stream element z* = $AND(h_1(z),h_2(z),...,h_k(z))$



Spam mail with Bloom filtering (1)

- y = 1B e-mails/darts
- x = 8B targets/bits
- Probability that a given target won't be hit by any darts (i.e. zero bit)?
- Probability that given dart will not hit a given targets is $\frac{x-1}{x}$
- Probability that none of y darts will hit a given target is $\left(\frac{x-1}{x}\right)^y = \left(1-\frac{1}{x}\right)^{x\frac{y}{x}} \to e^{-\frac{y}{x}}$
- Probability that any given bit will be zero is $e^{-\frac{y}{x}} = e^{-\frac{1}{8}}$
- Probability that given bit will be 1 is $1 - e^{-\frac{1}{8}} = 0.1175$
- Slightly less than 1/8=0.125

Spam mail with Bloom filtering (2)

- S has m members, array has n bits and there are k hash functions
  - Targets *x=n*
  - Number of darts *y=km*
- We want proportion of 0 be large
  - So non-S will hash to zero at least once
  - Choose k to be n/m or less
    - Probability of 0 is $e^{-\frac{y}{x}} = e^{-\frac{km}{n}} = 0.37 = 37\%$
    - Probability of 1 is $1 - e^{-\frac{km}{n}}$
    - Probability of false positive: $\left(1 - e^{-\frac{km}{n}}\right)^k$

Spam mail with Bloom filtering (3)

- In the previous example
  - Fraction on 1’s is 0.1175
  - Also the probability of false positive

- Use two different hash functions
  - 2B darts on 8B targets
  - Probability of zero is $e^{-\frac{1}{4}}$
  - False positive: $\left(1 - e^{-\frac{1}{4}}\right)^2$=0.0493

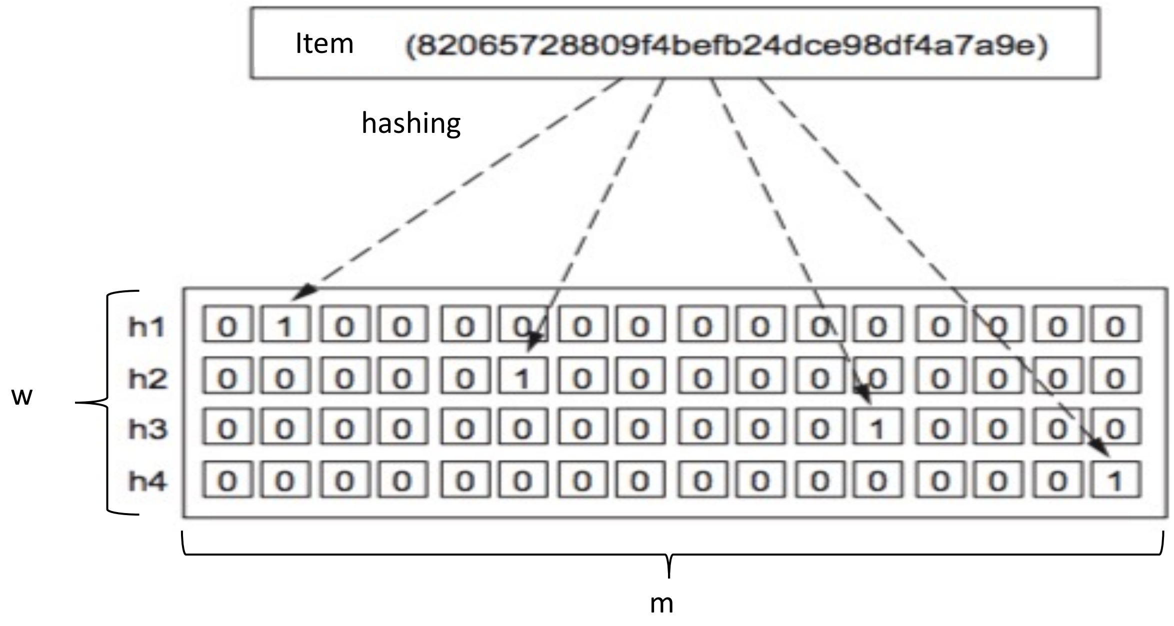
Frequency

- How many times has stream element X occurred?
- *Count-Min Sketch*
  - *A point query*: a particular stream element
  - ☐*A range query*: frequencies in a given range.
  - *An inner product query*: the join size of two sketches; Ex: we may use this to provide a summarization to this question: What products were viewed after an ad was served?

Count-Min sketch

- *w* numeric arrays, often called *counters*, the length of each is defined by the length *m*

  (m << n items)
- Each array is indexed starting at 0 and has a range of {0...*m* – 1}
- Each counter must be associated with a different hash function (h_1, h_2, h_3,...) , which must be pairwise independent



- Count first and compute the minimum next :
  - Count step: hash the item value using the hash function for each respective row and then increment the count for the cell the value hashes to by 1
  - Min step: The minimum value from the w cells represents the approximate count for the number of times the item was viewed
- This algorithm will never undercount, but could overcount
  - A width of 8 and a count of 128 (a 2-dimensional array of 8 x 128) the relative error was approximately 1.5%, and the probability of the relative error being 1.5% is 99.6%

Counting distinct elements

- Count the distinct items in a stream, but remember we are constrained by memory and do not have the luxury of storing the entire stream
- *Bit-pattern-based*
  - Observation of patterns of bits that occur at the beginning of the binary value of each element of the stream. Using the bit pattern - more specifically, the leading zeros in the binary representation of a hash of the stream element - the cardinality is determined
  - LogLog, HyperLogLog, and HyperLogLog++
- *Order statistics-based*
  - The algorithms in this class are based on order statistics, such as the smallest values that appears in a stream
  - MinCount and Bar-Yossef

Flajolet-Martin algorithm

- h(a) is hashed to a bit string
- Number of zeros at the end is tail lengths of h(a)
- Let *r* be maximum tail length seen so far
- Estimate number of different elements as 2^r

The Flajolet-Martin algorithm

- Hash element to a sufficiently long bit string
  - More possible hash values then elements
  - 64 bits for URLs (264 $values$)
- Whenever number of distinct elements increases
  - Number of different hash-values increases
  - The probability of "special/unusual" hash-value increases
- Unusual value
  - Ending in many 0's

- Probability that any given element has tail length at least r is 2^{-r}
- For m distinct element in the stream, the probability that none of them has at least r is

$$(1-2^{-r})^m = (1-2^{-r})^{2^{-r} m 2^r} \approx e^{-m2^{-r}}$$

- Estimate of m
  - For $m \gg 2^r$, $e^{-m2^{-r}}$ is small, at least one has r zeroes
  - For $m \ll 2^r$, $e^{-m2^{-r}}$ is large, no elements have r zeroes
  - Estimate the cardinality of M as $2^r/\Phi$, where $\Phi \approx 0.77351$.