THE UNIVERSITY of EDINBURGH
informatics

# STORM AND LOW-LATENCY PROCESSING

# Low latency processing

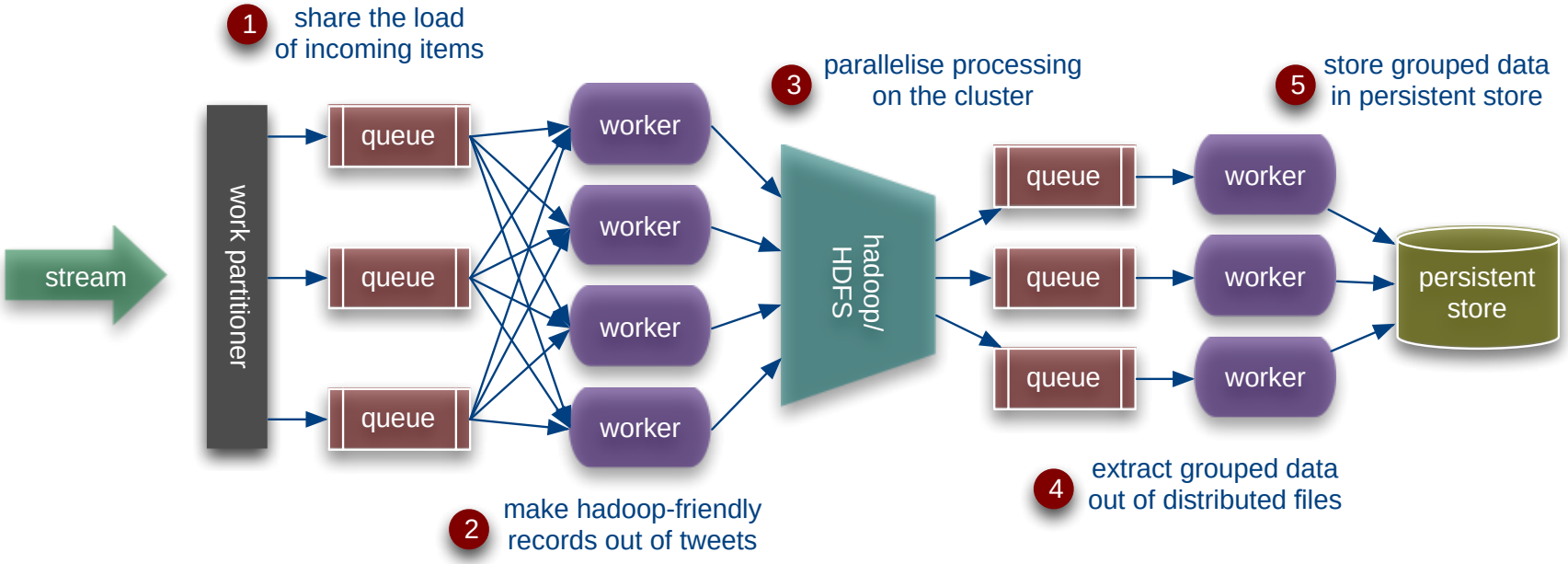
- Similar to data stream processing, but with a twist
  - Data is streaming into the system (from a database, or a network stream, or an HDFS file, or …)
  - We want to process the stream in a distributed fashion
  - And we want results as quickly as possible
- Not (necessarily) the same as what we have seen so far
  - The focus is not on summarising the input
  - Rather, it is on "parsing" the input and/or manipulating it on the fly

# The problem

- Consider the following use-case
- A stream of incoming information needs to be summarised by some identifying token
  - For instance, group tweets by hash-tag; or, group clicks by URL;
  - And maintain accurate counts
- But do that at a massive scale and in real time
- Not so much about handling the incoming load, but using it
  - That's where latency comes into play
- Putting things in perspective
  - Twitter's load is not that high: at 15k tweets/s and at 150 bytes/tweet we're talking about 2.25MB/s
  - Google served 34k searches/s in 2010: let's say 100k searches/s now and an average of 200 bytes/search that's 20MB/s
  - But this 20MB/s needs to filter PBs of data in less than 0.1s; that's an EB/s throughput

# A rough approach

- Latency
  - Each point 1 − 5 in the figure introduces a high processing latency
  - Need a way to transparently use the cluster to process the stream




- Bottlenecks
  - No notion of locality
    - Either a queue per worker per node, or data is moved around
  - What about reconfiguration?
    - If there are bursts in traffic we need to shutdown, reconfigure and redeploy

# Storm

- Started up as backtype; widely used in Twitter
- Open-sourced (you can download it and play with it!
  - http://storm-project.net/
- On the surface, Hadoop for data streams
  - Executes on top of a (likely dedicated) cluster of commodity hardware
  - Similar setup to a Hadoop cluster
    - Master node, distributed coordination, worker nodes
    - We will examine each in detail
- But whereas a MapReduce job will finish, a Storm job—termed a topology—runs continuously
  - Or rather, until you kill it

# Storm vs. Hadoop

| Storm | Hadoop |
|---|---|
| Real-time stream processing | Batch processing |
| Stateless | Stateful |
| Master/Slave architecture with ZooKeeper based coordination. The master node is called as **nimbus** and slaves are **supervisors**. | Master-slave architecture with/without ZooKeeper based coordination. Master node is **job tracker** and slave node is **task tracker**. |
| A Storm streaming process can access tens of thousands messages per second on cluster. | Hadoop Distributed File System (HDFS) uses MapReduce framework to process vast amount of data that takes minutes or hours. |
| Storm topology runs until shutdown by the user or an unexpected unrecoverable failure. | MapReduce jobs are executed in a sequential order and completed eventually. |
| **distributed and fault-tolerant** | **distributed and fault-tolerant** |
| No Single Point of Failure. If nimbus / supervisor dies, restarting makes it continue from where it stopped, hence nothing gets affected. | JobTracker as Single Point of Failure. If it dies, all the running jobs are lost. |

# Application Examples

- **Twitter** − Twitter is using Apache Storm for its range of "Publisher Analytics products". "Publisher Analytics Products" process each and every tweets and clicks in the Twitter Platform. Apache Storm is deeply integrated with Twitter infrastructure.
- **NaviSite** − NaviSite is using Storm for Event log monitoring/auditing system. Every logs generated in the system will go through the Storm. Storm will check the message against the configured set of regular expression and if there is a match, then that particular message will be saved to the database.
- **Wego** − Wego is a travel metasearch engine located in Singapore. Travel related data comes from many sources all over the world with different timing. Storm helps Wego to search real-time data, resolves concurrency issues and find the best match for the end-user.

# Storm topologies

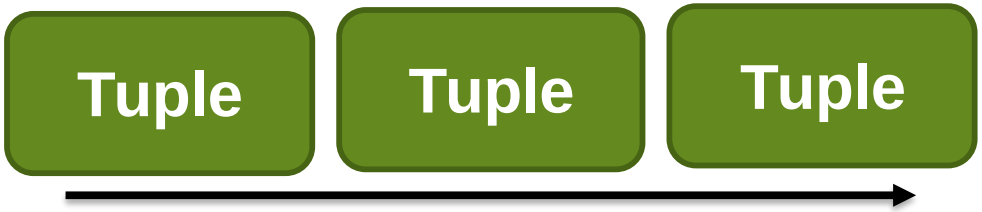
- A Storm topology is a graph of computation
  - Graph contains nodes and edges
  - Nodes model processing logic (i.e., transformation over its input)
  - Directed edges indicate communication between nodes
  - No limitations on the topology; for instance one node may have more than one incoming edges and more than one outgoing edges
- Storm processes topologies in a distributed and reliable fashion

# Tuple

Tuple

- An ordered list of elements
- E.g., <tweeter, tweet>
  - <“Jon”, “Hello everybody”>
  - <“Jane”, “Look at these cute cats!”>
- E.g., <URL, clicker-IP, date, time>
  - <www.google.com,101.201.301.401,4/4/2016,10:35:40>
  - <www.google.com,101.231.311.101,4/4/2016,10:35:43>

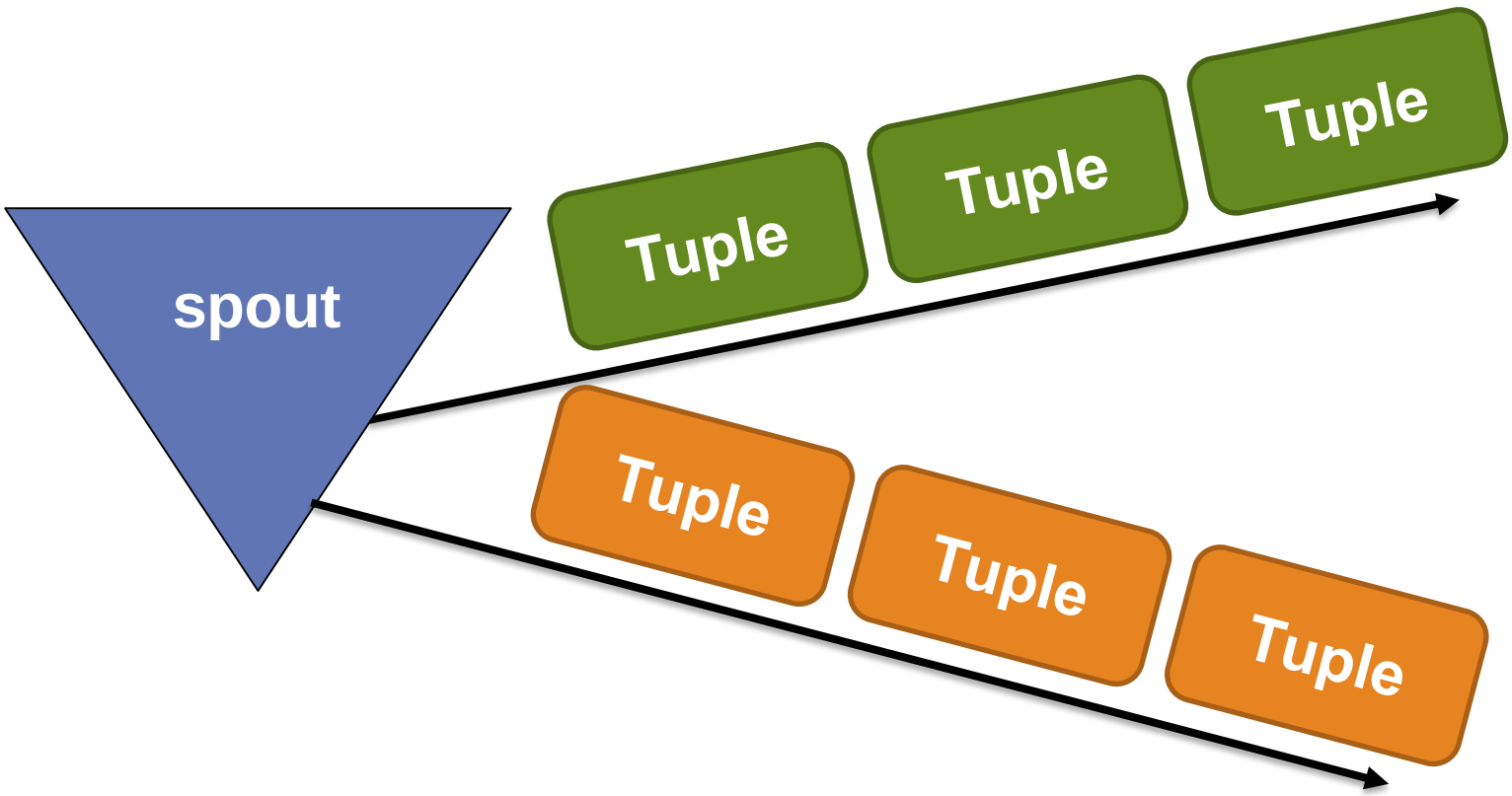
# Stream




- Potentially unbound sequence of tuples
- Twitter Example:
  - <“Jon”, “Hello everybody”>, <“Jane”, “Look at these cute cats!”>, <“James”,”I like cats too.”>, …
- Website Example
  - <www.google.com,101.201.301.401,4/4/2016,10:35:40>,<www.google.com,101.231.311.101,4/4/2016,10:35:43>,…

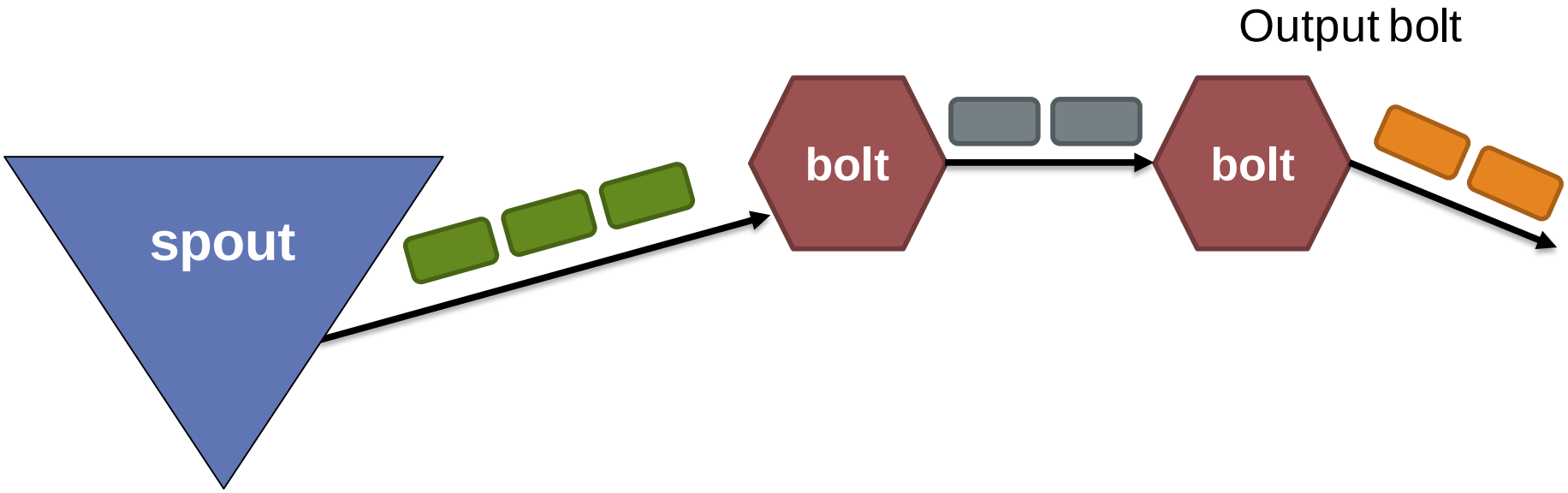
# Spout

- A Storm entity (process) that is a source of streams
- Often reads from a crawler or database

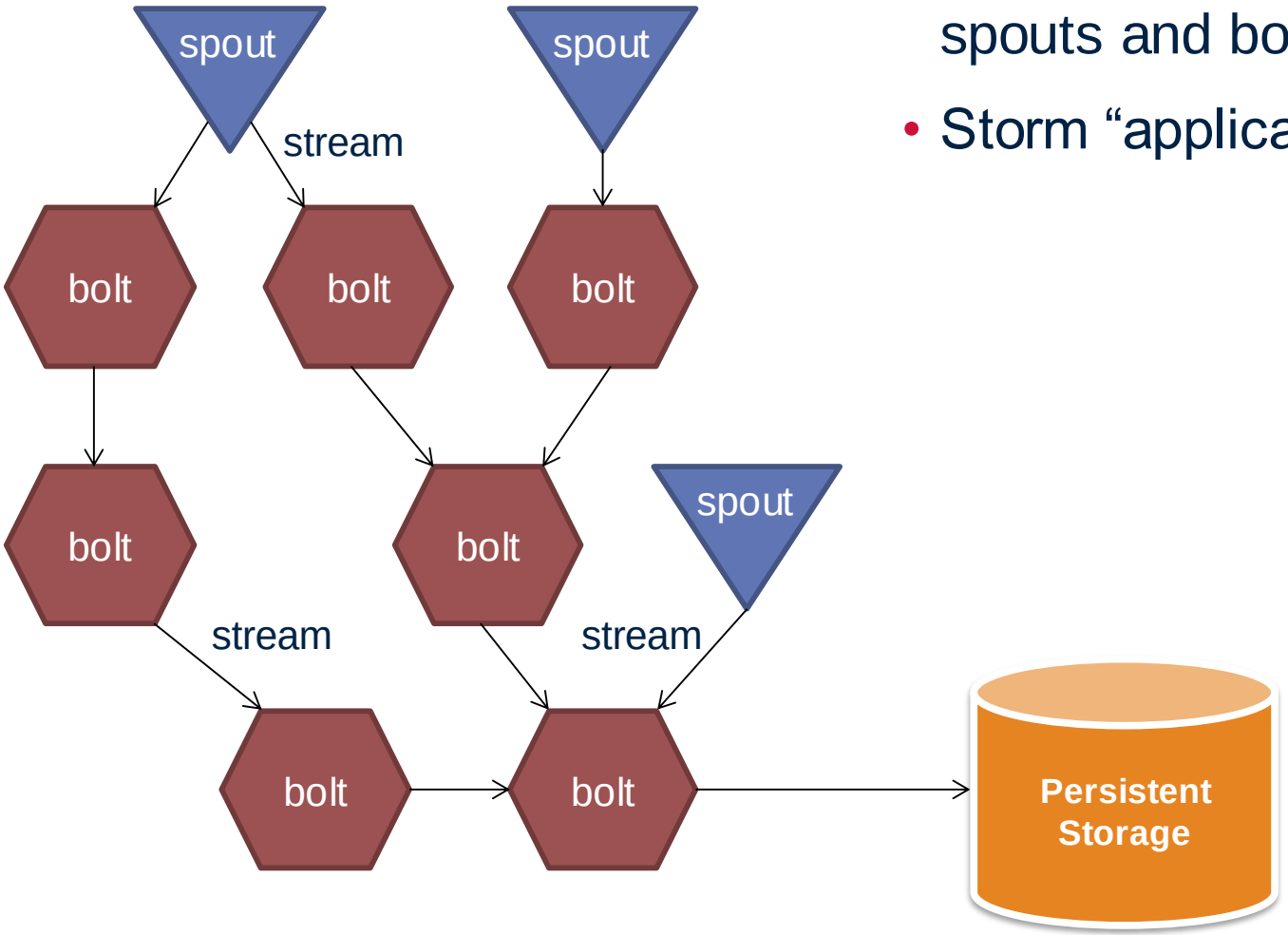


# Bolt

- A Storm entity (process) that
  - Processes input streams
  - Outputs more streams for other bolts

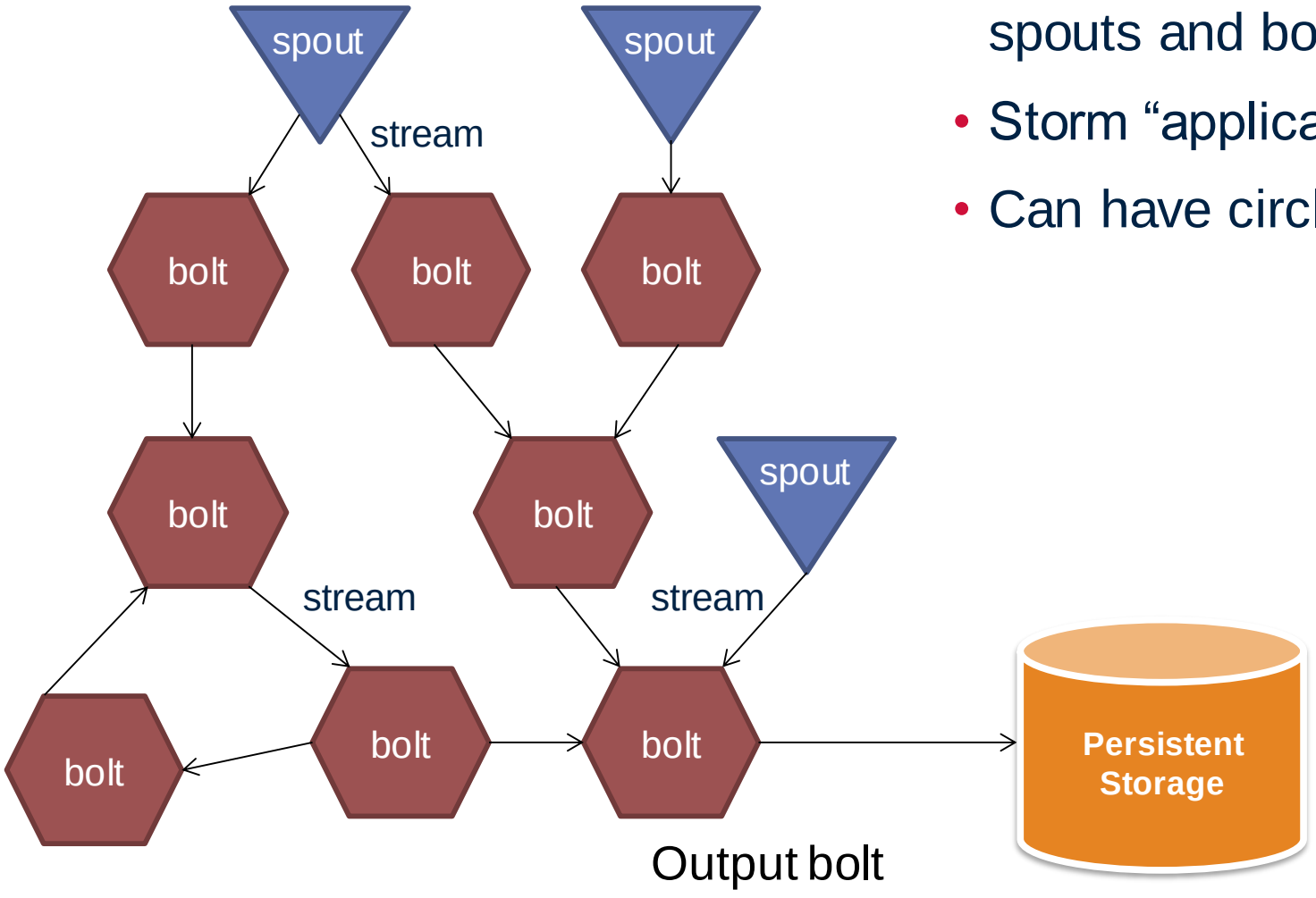


# Topology




- Directed graph of spouts and bolts
- Storm “application”

# Topology



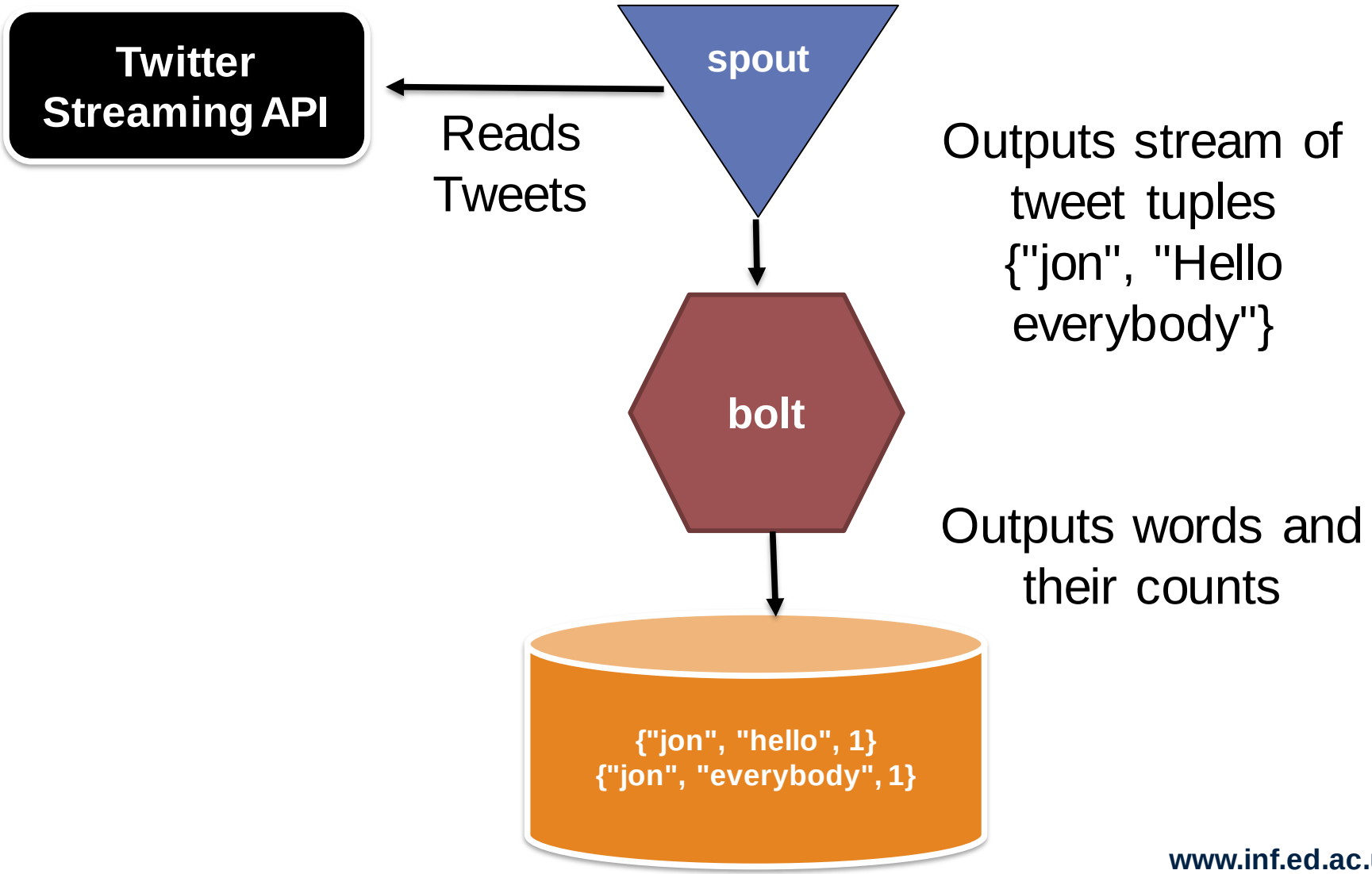

- Directed graph of spouts and bolts
- Storm “application”
- Can have circles

# Types of Bolts

- **Filter**: forward only tuples which satisfy a condition
- **Joins**: When receiving two streams A and B, output all pairs (A,B) which satisfy a condition
- **Apply/Transform**: Modify each tuple according to a function
- …

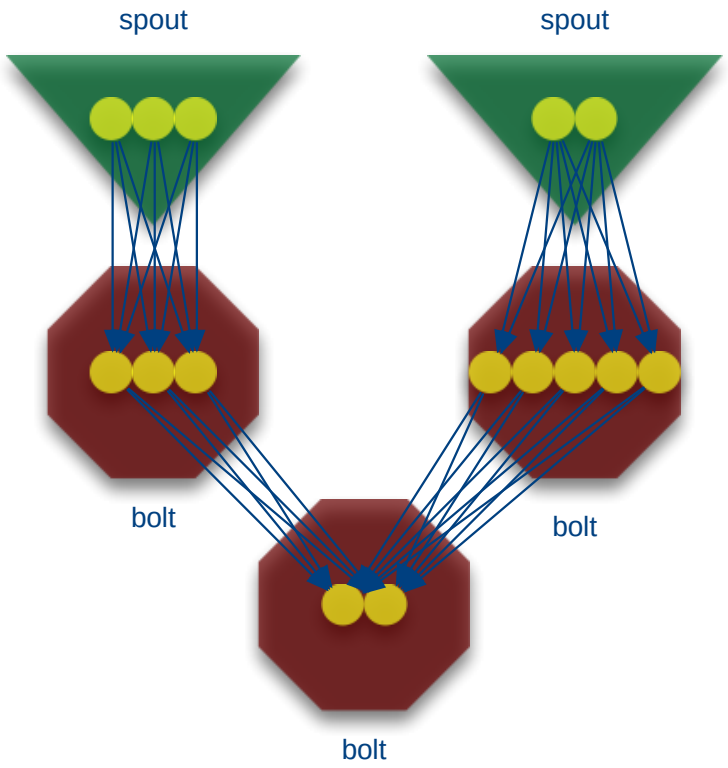
- Bolts need to process a lot of data
  - Need to make them fast

# Topology Example

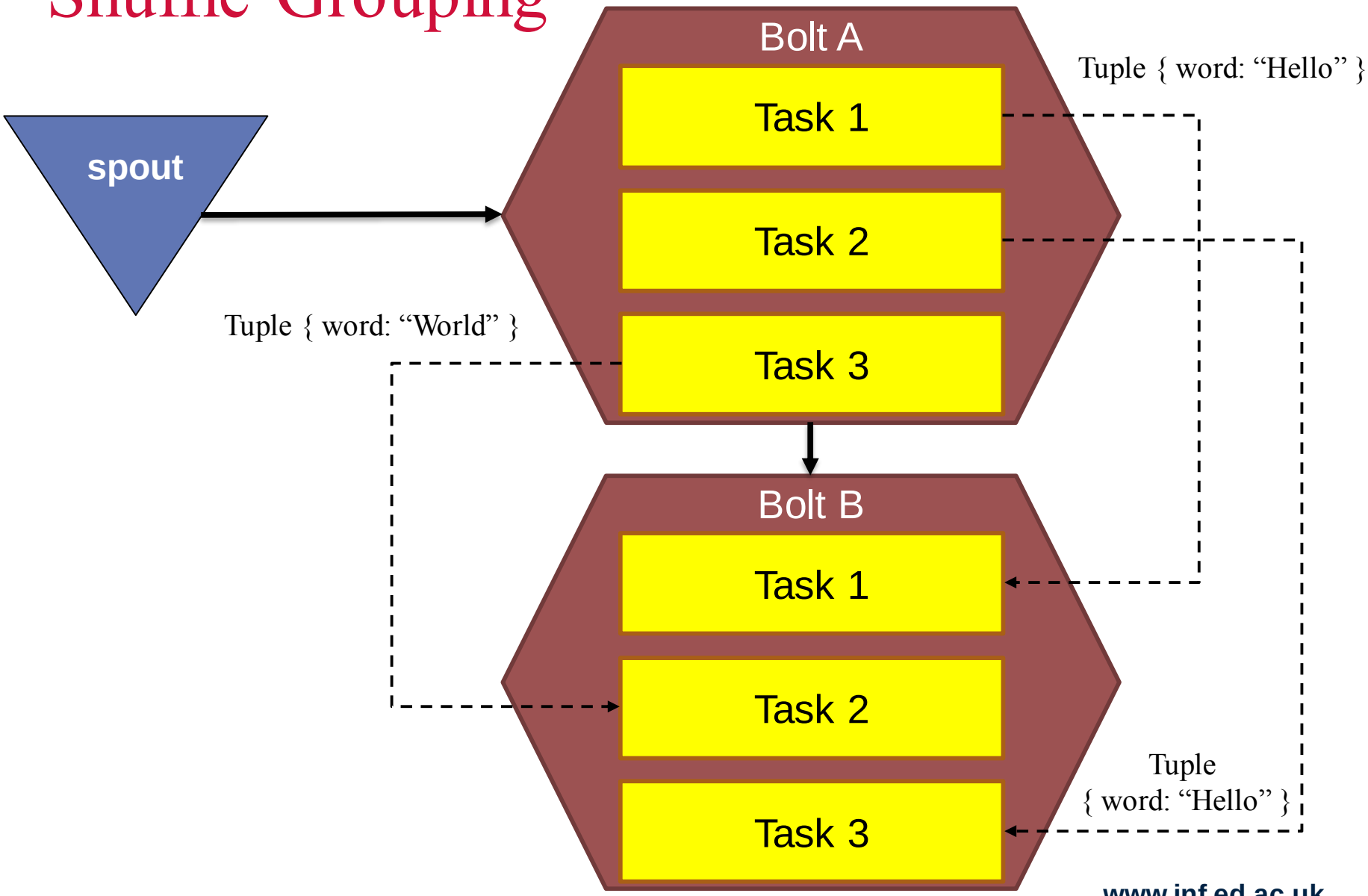


# From topology to processing: stream groupings

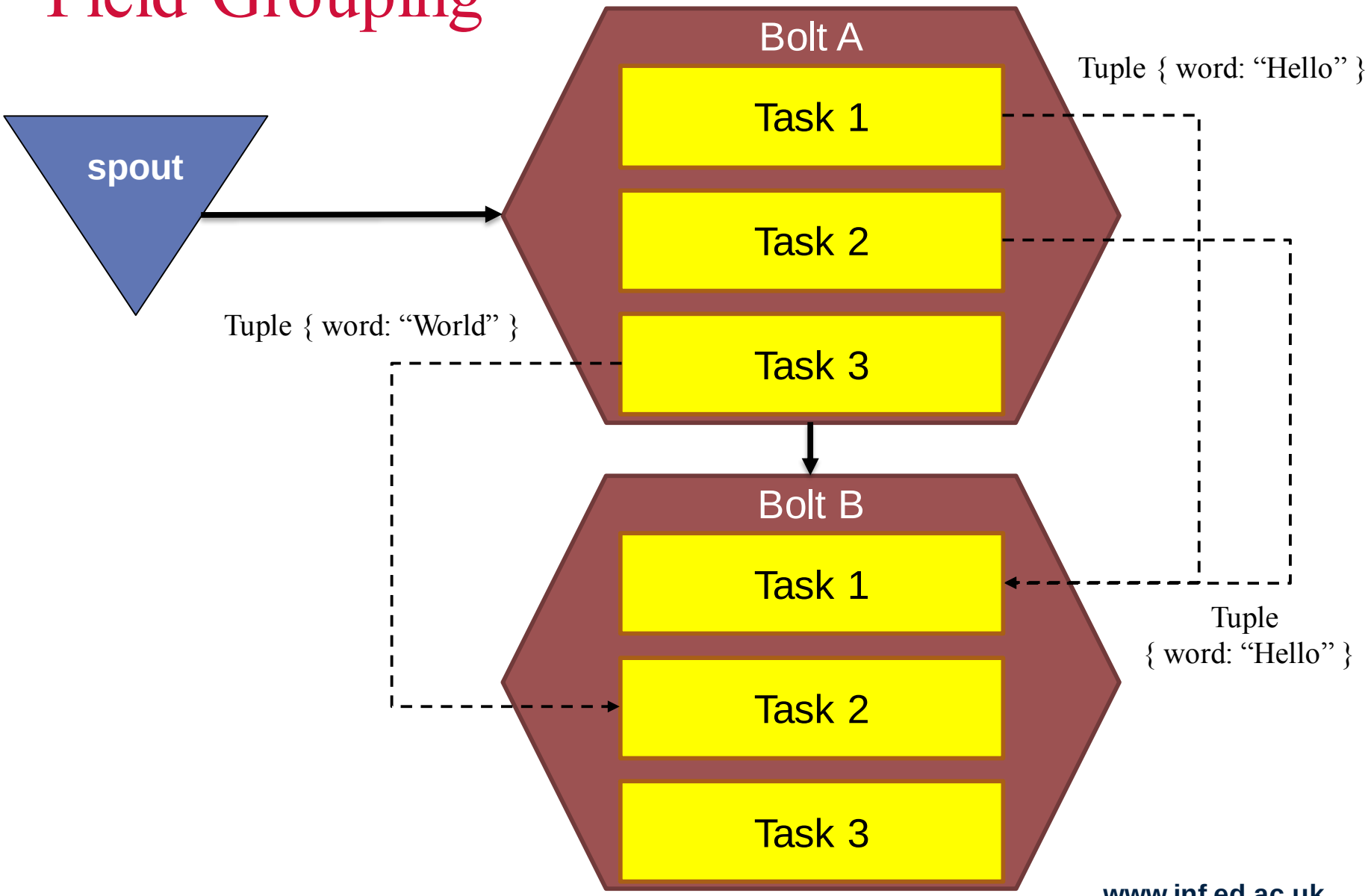
- Spouts and bolts are replicated in tasks, each task executed in parallel by a worker
  - User-defined degree of replication
  - All pairwise combinations are possible between tasks
- When a task emits a tuple, which task should it send to?
- Stream groupings dictate how to propagate tuples
  - Shuffle grouping: round-robin
  - Field grouping: based on the data value (e.g., range partitioning)



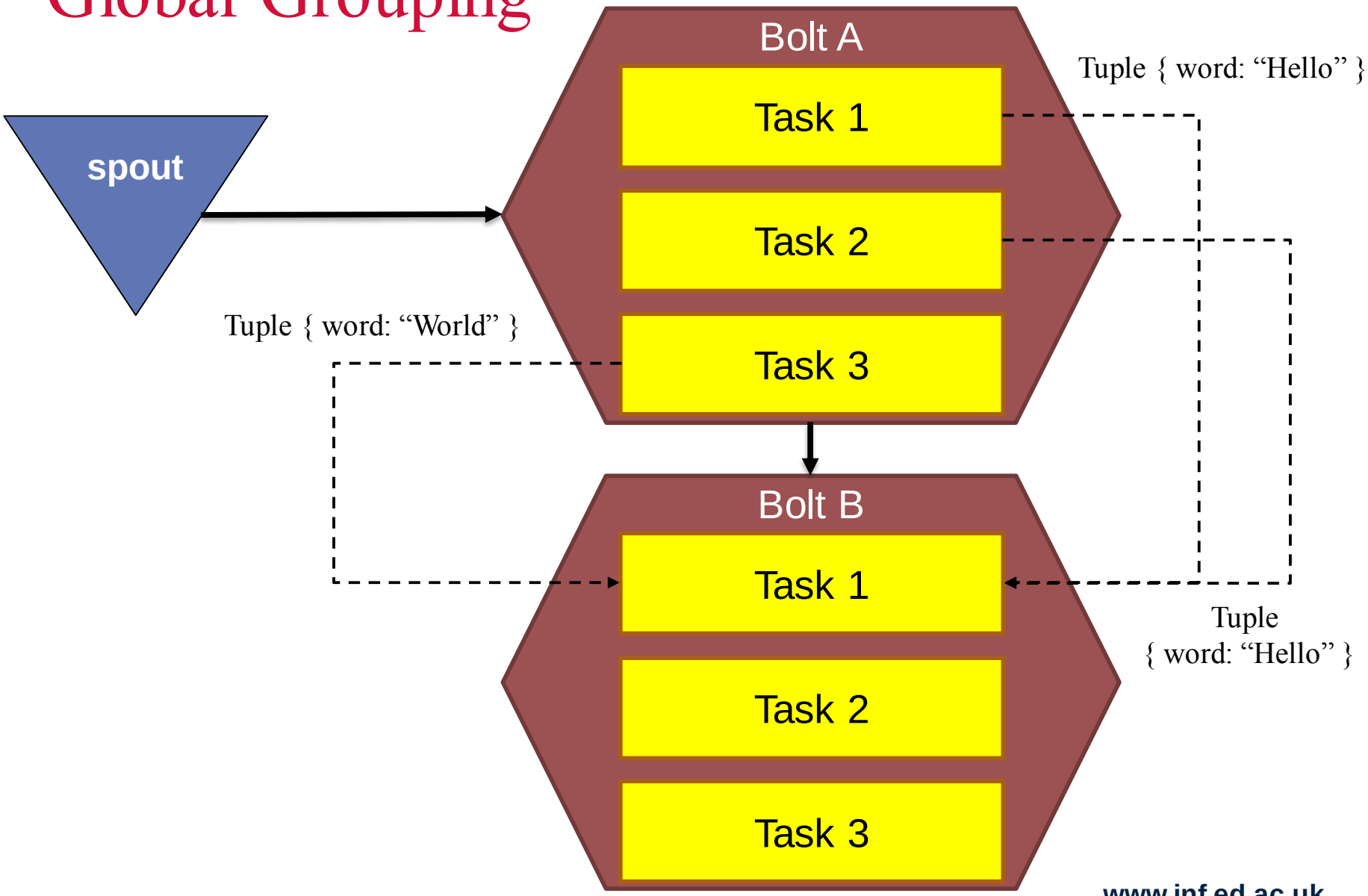
# Shuffle Grouping



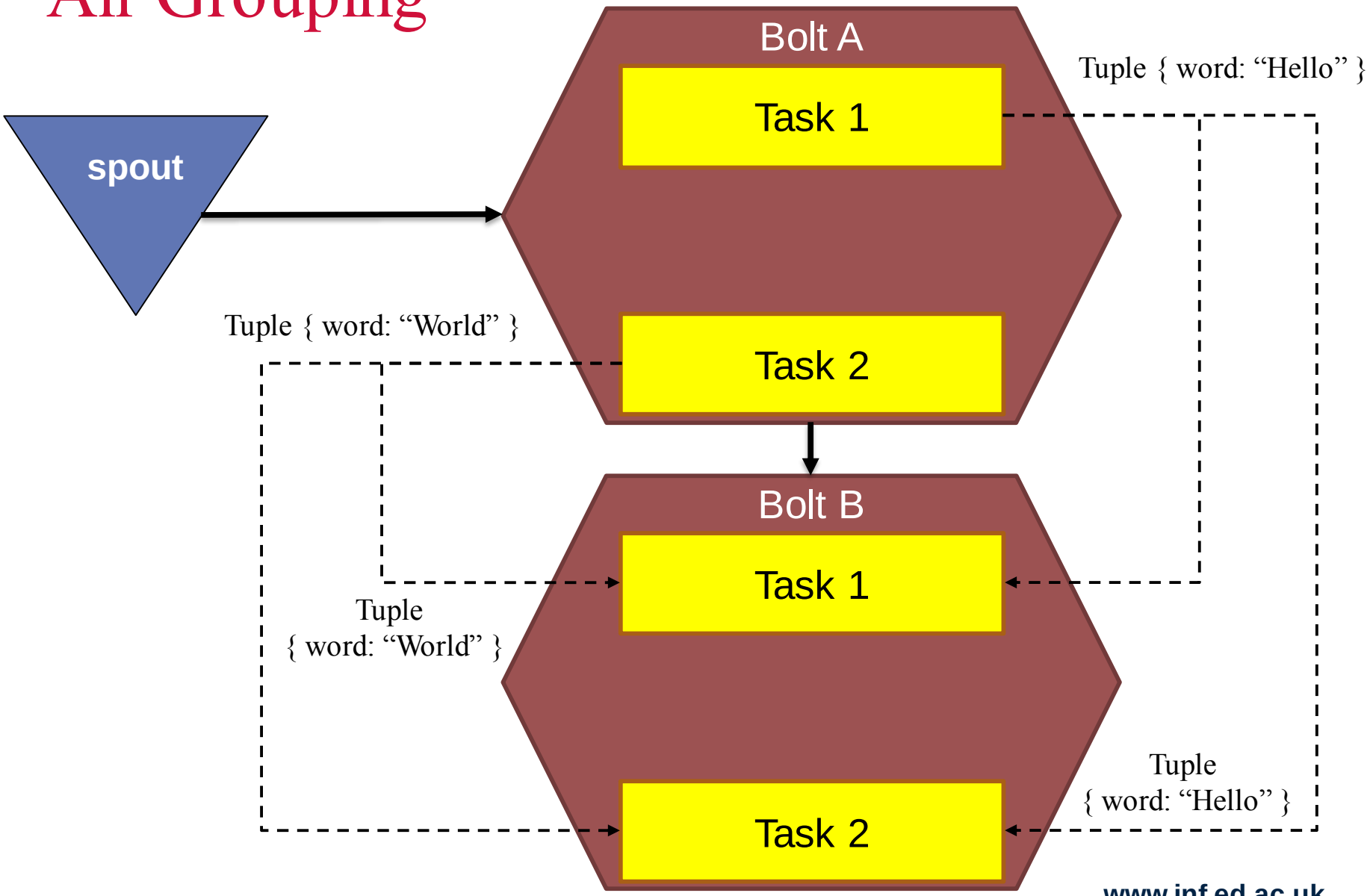
# Field Grouping



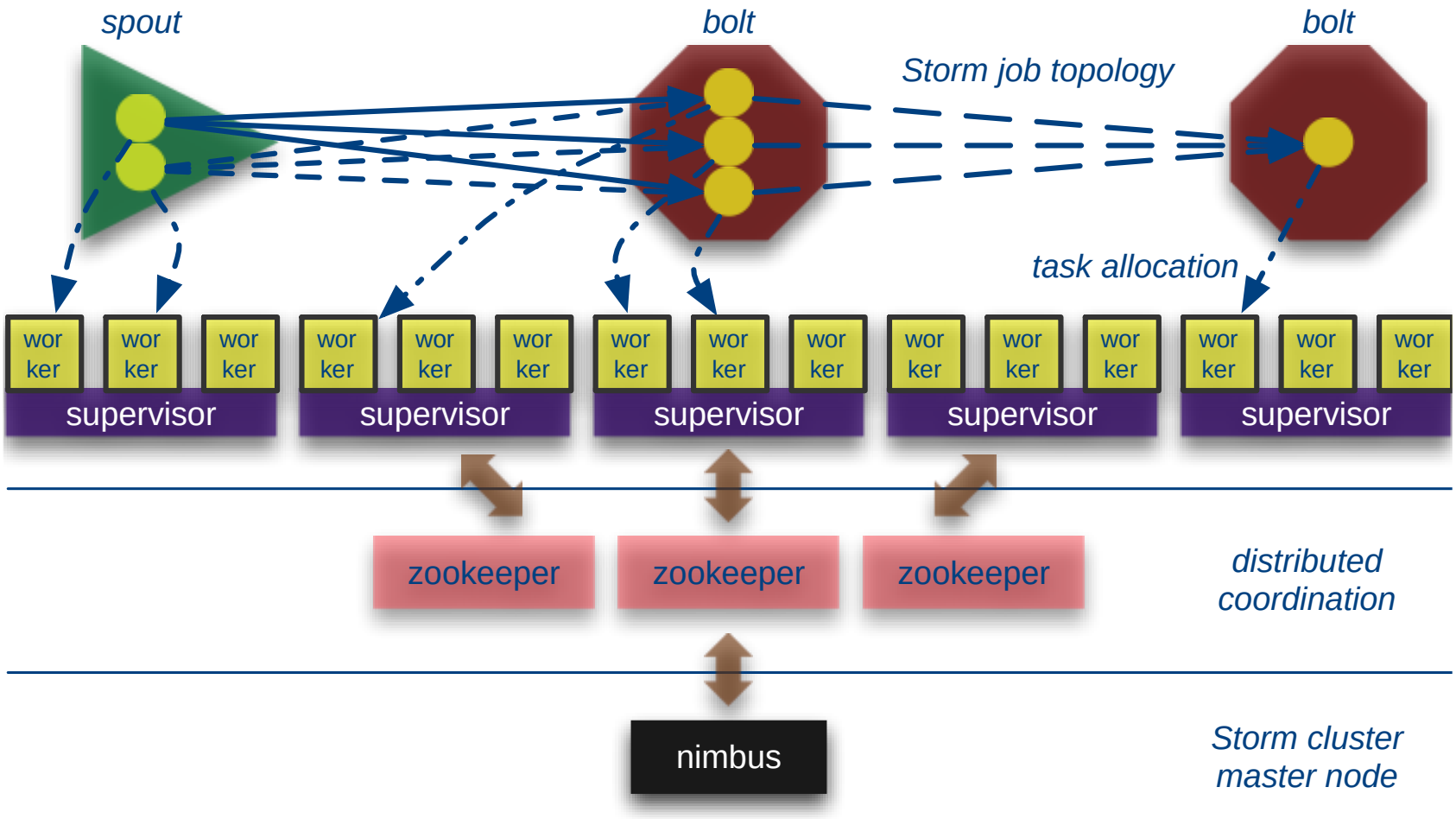
# Global Grouping



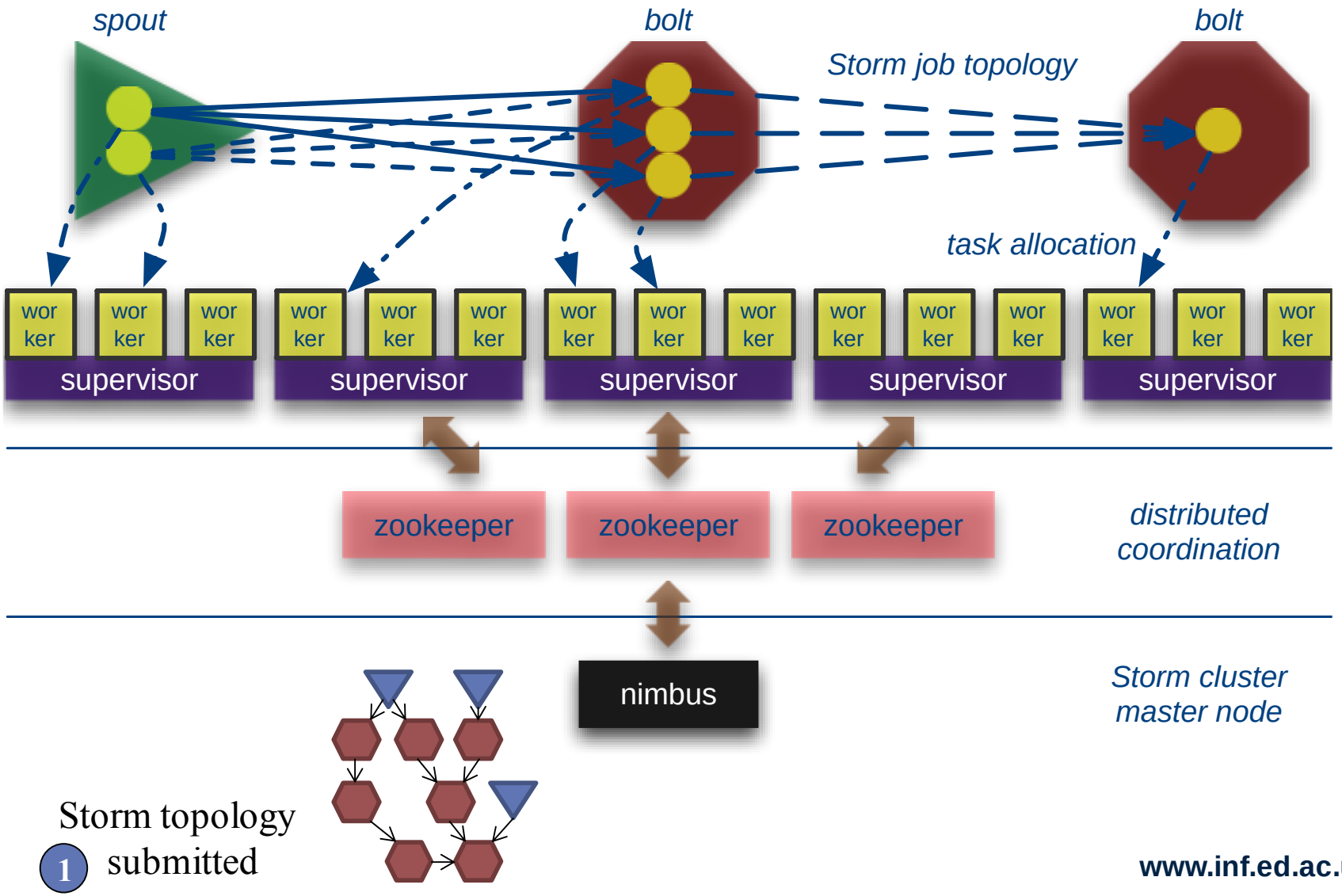
# All Grouping



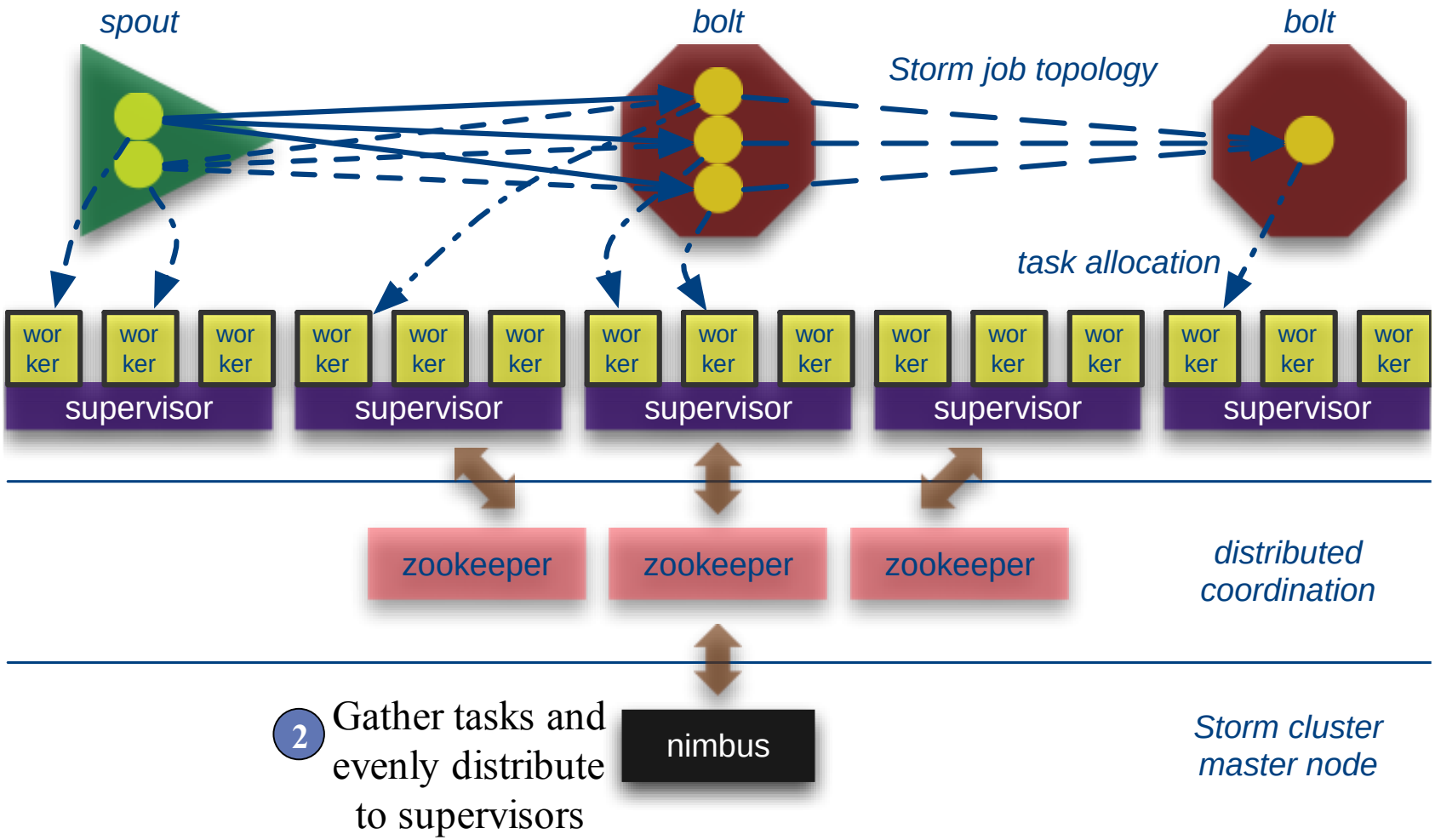
# Storm Architecture



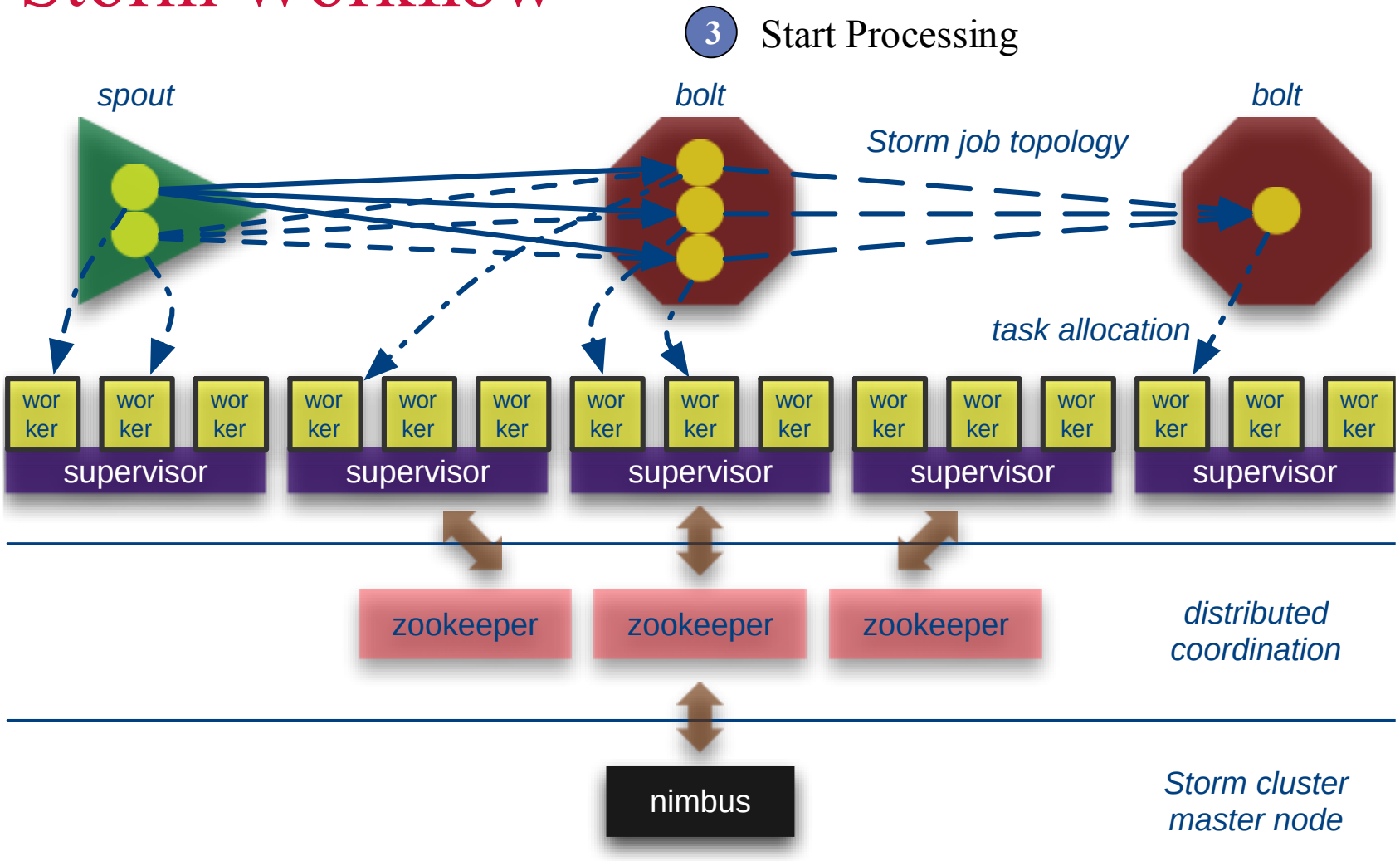
# Storm Workflow



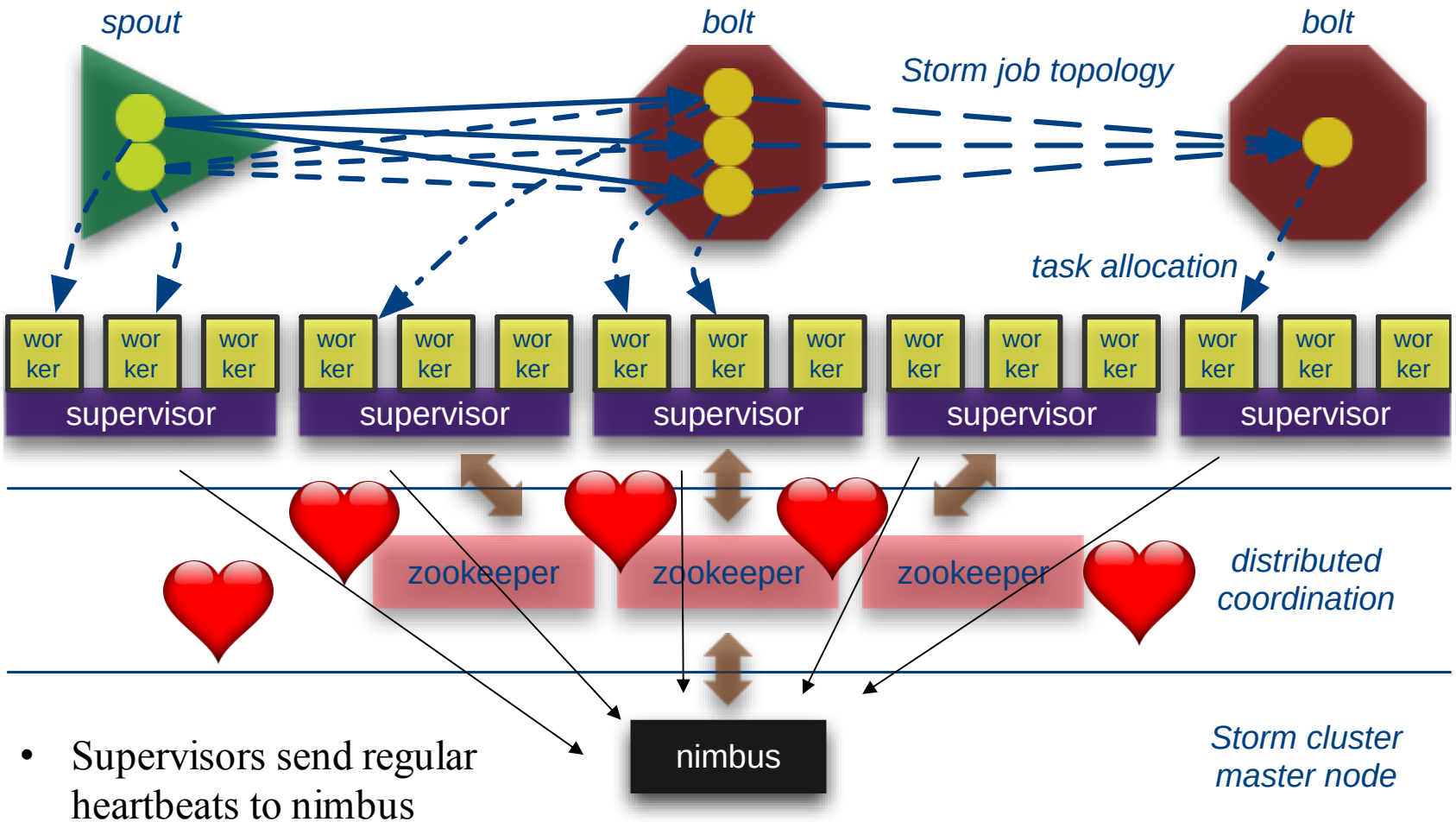
# Storm Workflow



# Storm Workflow

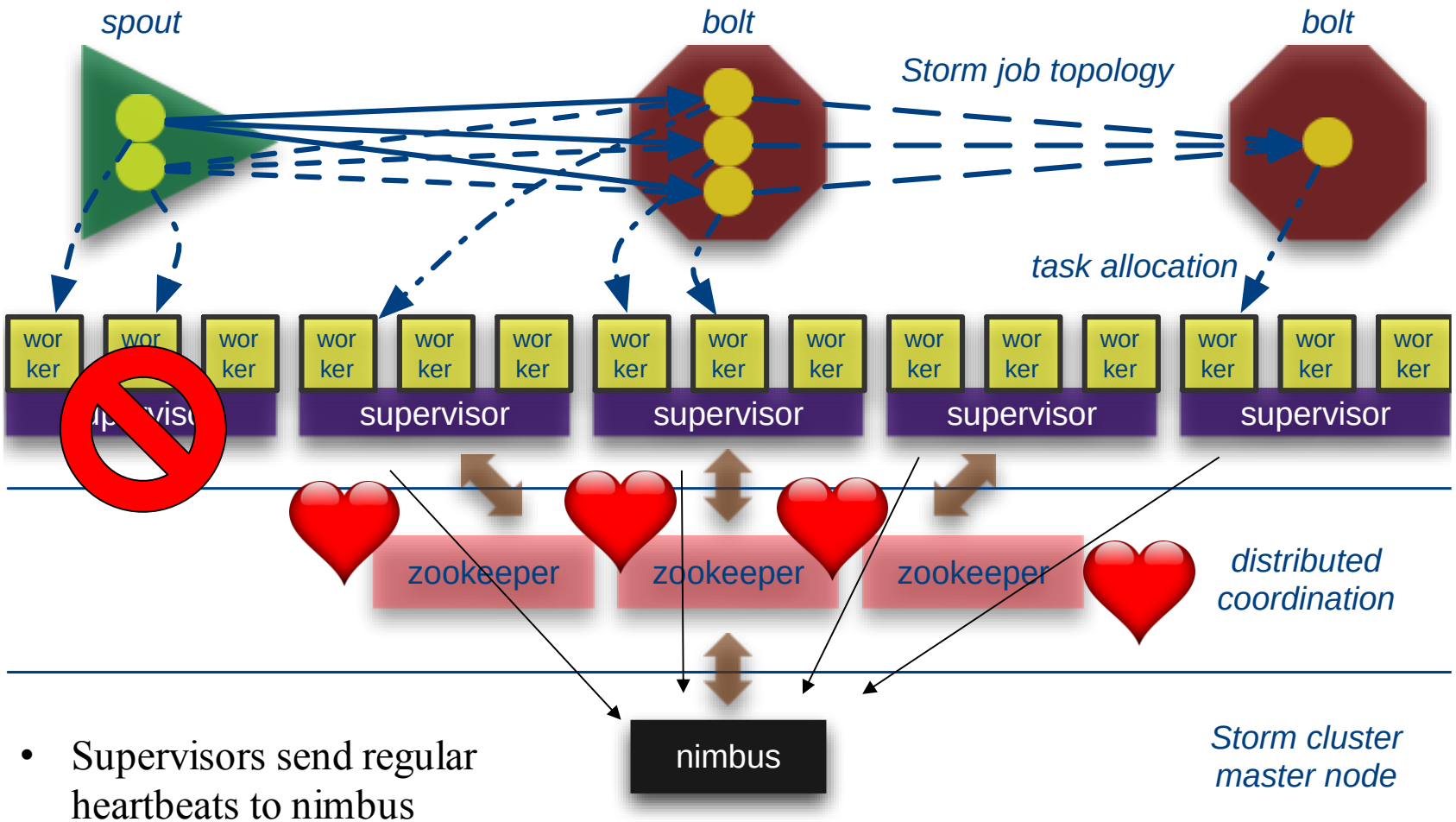


# Failure Recovery



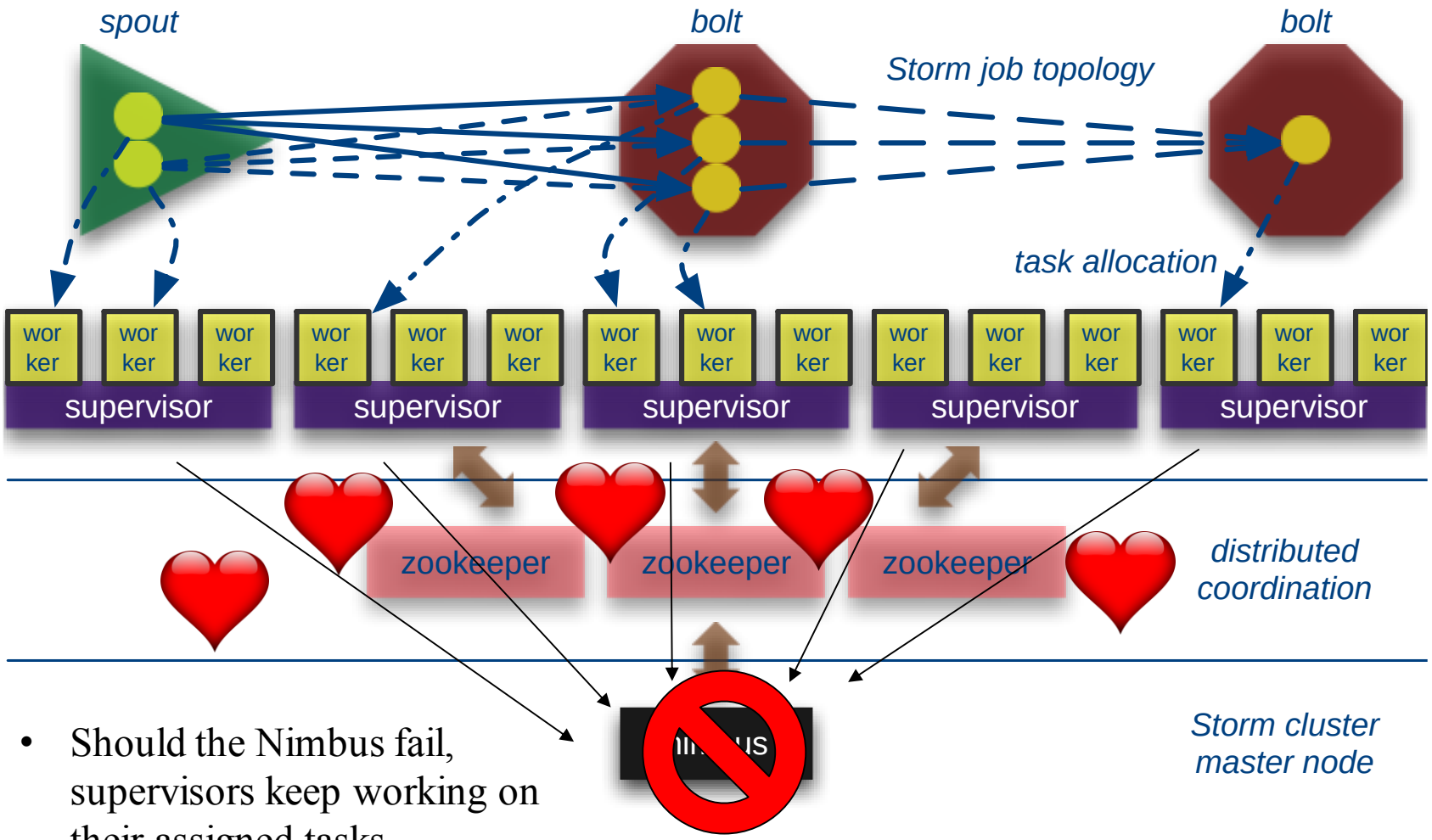

- Supervisors send regular heartbeats to nimbus
- When heartbeat stops, nimbus assigns tasks to other supervisors

# Failure Recovery



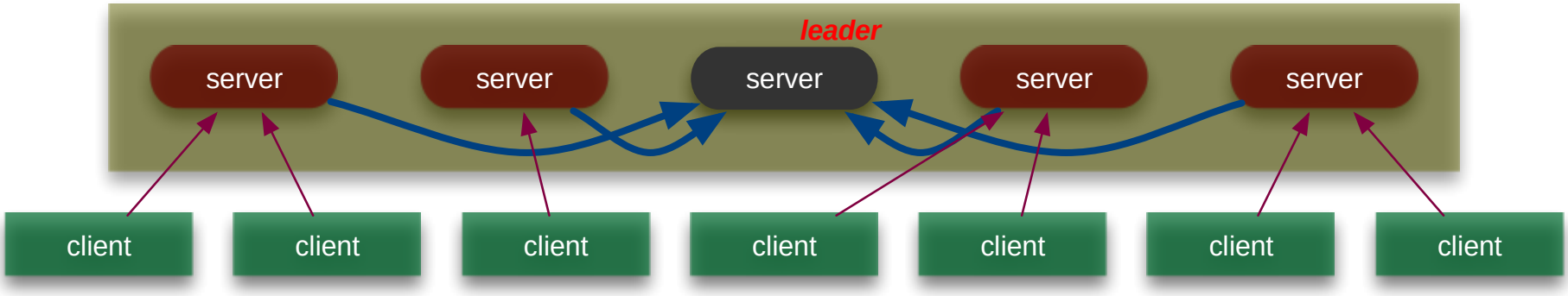

- Supervisors send regular heartbeats to nimbus
- When heartbeat stops, nimbus assigns tasks to other supervisor

# Failure Recovery




- Should the Nimbus fail, supervisors keep working on their assigned tasks

# Zookeeper: distributed reliable storage and coordination




- Design goals
  - Distributed coordination service
  - Hierarchical name space
  - All state kept in main memory, replicated across servers
  - Read requests are served by local replicas
  - Client writes are propagated to the leader
  - Changes are logged on disk before applied to in-memory state
  - Leader applies the write and forwards to replicas

- Guarantees
  - Sequential consistency: updates from a client will be applied in the order that they were sent
  - Atomicity: updates either succeed or fail; no partial results
  - Single system image: clients see the same view of the service regardless of the server
  - Reliability: once an update has been applied, it will persist from that time forward
  - Timeliness: the clients' view of the system is guaranteed to be up-to-date within a certain time bound

# Putting it all together: word count

```
// instantiate a new topology
TopologyBuilder builder = new TopologyBuilder();
// set up a new spout with five tasks
builder.setSpout("spout", new RandomSentenceSpout(), 5);
// the sentence splitter bolt with eight tasks
builder.setBolt("split", new SplitSentence(), 8)
    .shuffleGrouping("spout"); // shuffle grouping for the ouput
// word counter with twelve tasks
builder.setBolt("count", new WordCount(), 12)
    .fieldsGrouping("split", new Fields("word")); // field grouping
// new configuration
Config conf = new Config();
// set the number of workers for the topology; the 5x8x12=480 tasks
// will be allocated round-robin to the three workers, each task
// running as a separate thread
conf.setNumWorkers(3);
// submit the topology to the cluster
StormSubmitter.submitTopology("word-count", conf, builder.createTopology());
```

# Summary

- Introduction to Apache Storm low latency stream processing
- Storm topology consisting of Spouts and Bolts
- Storm Architecture