Northeastern University - Seattle

CS6650 Building Scalable Distributed Systems

Professor Ian Gorton

Building Scalable Distributed Systems

Week 11 – Scaling Request Processing

Outline

- Scaling request processing
- Apache Storm
- Apache Kafka
- Kafka Streaming

Scaling Request Processing

Background

Synchronous request processing

- Well established since 1990s
- Multithreaded containers
- Scaling through stateless servers

Asynchronous request processing

- Original approaches based on brokers
- Single logical queue/topics
- Scaling bottlenecks
- Non-persistent messaging to attain high throughput

Internet Scale Message Streams

Internet scale apps need:

- To capture/reliably store vast quantities of messages/sec
- Beyond single topic message broker approaches
- E.g: like’s on Facebook, GPS readings from vehicles, latencies from application monitoring

Need to derive ‘real time’ value from messages

- Trending topics on Twitter
- Average latencies in last minute
- Identify traffic holdups from GPS data
- Perform sales analysis from supermarket checkouts

Message Streams

Unbounded series of events

- From a producer
- Continuous
- Immutable
- Reliable
- Never ends …..

Analysis required

- Low latency
- Potentially multiple event streams
- Potentially multiple consumers

Example

- Supermarket transactions from multiple stores
  - Producer per store
  - Stream of transactions from checkouts
- Analysis
  - Sales volume per store in last minute
    - Number of items
    - $$ value (total, median, mean)
  - Most popular item per store in last 5 minutes
  - Most popular items across all stores in last 5 minutes
  - Most popular items across stores in region (city/state/etc) in last 5 minutes
  - Compare all store's revenues in last hour

Stream Processing

Define event sources

producers write to

Consumers read from

Enable events to be routed to multiple consumers

Enable events to be reliably processed

Low latencies

Consumers perform analyses concurrently

Scalable across multiple CPUs/nodes

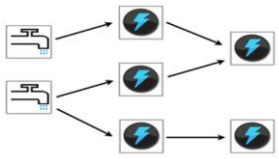
	Flume	NiFi	Gearpump	Apex	Kafka Streams	Spark Streaming	Storm	Storm + Trident	Samza	Flink	Ignite Streaming	Beam [*GC DataFlow]
Current version	1.6.0	0.6.1	incubating	3.3.0	0.10.0.0	1.6.1	1.0.0	1.0.0	0.10.0	1.0.2	1.5.0	incubating
Category	DC/SEP	DC/SEP	SEP	DC/ESP	ESP	ESP	ESP/CEP	ESP/CEP	ESP	ESP/CEP	ESP/CEP	SDK
Event size	single	single	single	single	single	micro-batch	single	mini-batch	single	single	single	single
Available since (incubator since)	June 2012 (June 2011)	July 2015 (Nov 2014)	(Mar 2016)	Apr 2016 (Aug 2015)	May 2016 (July 2011)	Feb 2014 (2013)	Sep 2014 (Sep 2013)	Sep 2014 (Sep 2013)	Jan 2014 (July 2013)	Dec 2014 (Mar 2014)	Sep 2015 (Oct 2014)	(Feb 2016)
Contributors	26	78	19	53	183	891	215	215	54	184	65	82
Main backers	Apple Cloudera	Hortonworks	Intel Lightbend	Data Torrent	Confluent	AMPLab Databricks	Backtype Twitter	Backtype Twitter	LinkedIn	dataArtisans	GridGain	Google
Delivery guarantees	at least once	at least once	exactly once at least once (with non-fault-tolerant sources)	exactly once	at least once	exactly once at least once (with non-fault-tolerant sources)	at least once	exactly once	at least once	exactly once	at least once	exactly once*
State management	transactional updates	local and distributed snapshots	checkpoints	checkpoints	local and distributed snapshots	checkpoints	record acknowledgements	record acknowledgements	local snapshots distributed snapshots (fault-tolerant)	distributed snapshots	checkpoints	transactional updates*
Fault tolerance	yes (with file channel only)	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes*
Out-of-order processing	no	no	yes	no	yes	no	yes	yes	yes (but not within a single partition)	yes	yes	yes*
Event prioritization	no	yes	programmable	programmable	programmable	programmable	programmable	programmable	yes	programmable	programmable	programmable
Windowing	no	no	time-based	time-based	time-based	time-based	time-based count-based	time-based count-based	time-based	time-based count-based	time-based count-based	time-based
Back-pressure	no	yes	yes	yes	N/A	yes	yes	yes	yes	yes	yes	yes*
Primary abstraction	Event	FlowFile	Message	Tuple	KafkaStream	DStream	Tuple	TridentTuple	Message	DataStream	IgniteDataStreamer	PCollection
Data flow	agent	flow (process group)	streaming application	streaming application	process topology	application	topology	topology	job	streaming dataflow	job	pipeline
Latency	low	configurable	very low	very low	very low	medium	very low	medium	low	low (configurable)	very low	low*
Resource management	native	native	YARN	YARN	Any process manager (e.g. YARN, Mesos, Chef, Puppet, Salt, Kubernetes, ...)	YARN Mesos	YARN Mesos	YARN Mesos	YARN	YARN	YARN Mesos	integrated*
Auto-scaling	no	no	no	yes	yes	yes	no	no	no	no	no	yes*
In-flight modifications	no	yes	yes	yes	yes	no	yes (for resources)	yes (for resources)	no	no	no	no
API	declarative	compositional	declarative	declarative	declarative	declarative	compositional	compositional	compositional	declarative	declarative	declarative
Primarily written in	Java	Java	Scala	Java	Java	Scala	Clojure	Java	Scala	Java	Java	Java
API languages	text files Java	REST (GUI)	Scala Java	Java	Java	Scala Java Python	Scala Java Clojure Python Ruby	Java Python Scala	Java	Java Scala Python	Java .NET C++	Java*
Notable users	Meebo Sharethrough SimpleGeo	N/A	Intel Levi's Honeywell	Capital One GE Predix PubMatic	N/A	Kelkoo Localytics AsiaInfo Opentable Faimdata Guavus	Yahoo! Spotify Groupon Flipboard The Weather Channel Alibaba Baidu Yelp WebMD	Klout GumGum CrowdFlower	LinkedIn Netflix Intuit Uber	King Otto Group	GridGain	N/A

https://databaseline.files.wordpress.com/2016/03/apache-streaming-technologies3.png

Apache Storm

Apache Storm

- Distributed computation framework for unbounded streams of data
- Defines "spouts" and "bolts" for information sources and manipulations that can be organized into arbitrary topologies
- Example uses:
    - Real-time analytics
    - ETL
    - Machine learning
- Open sourced by Twitter
- Apache incubator in 2013



Apache Storm

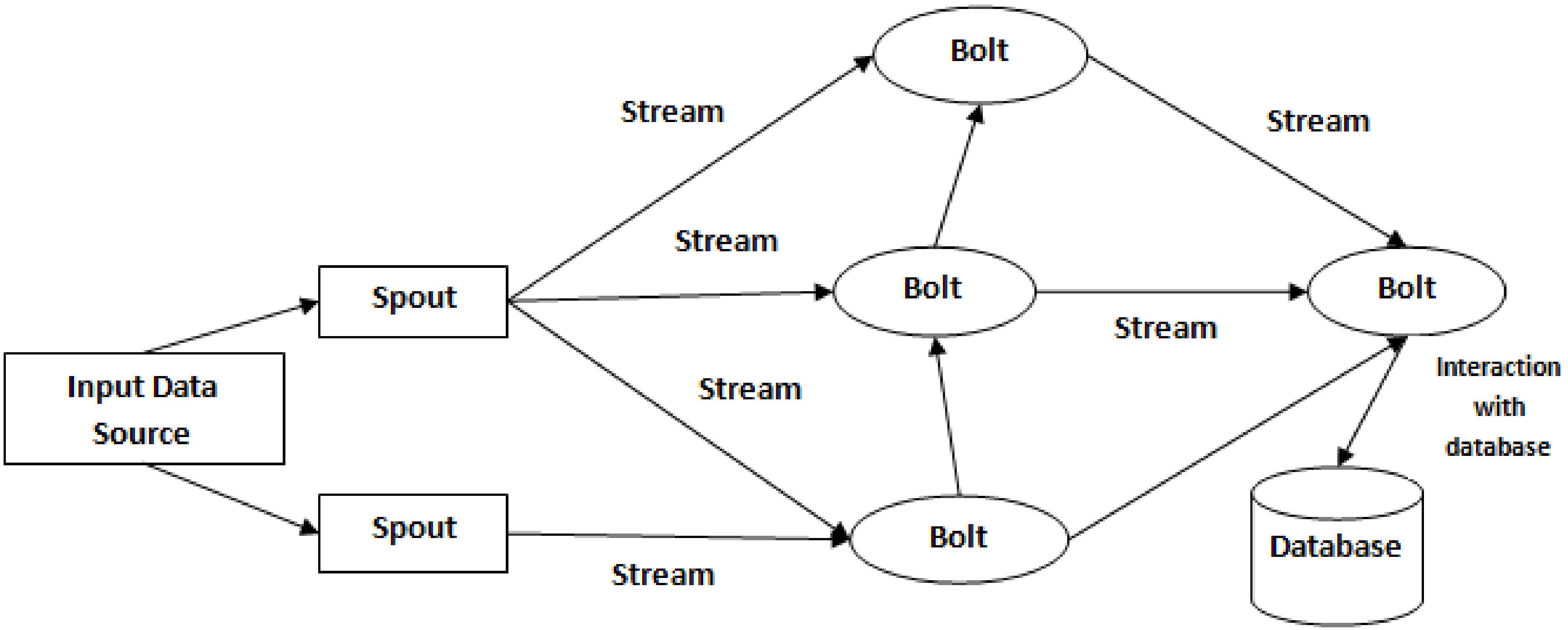
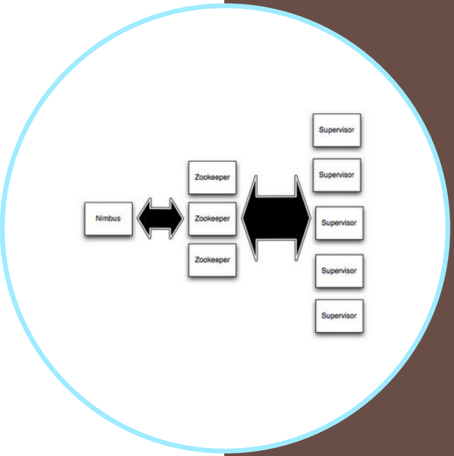


Figure- Storm Topology

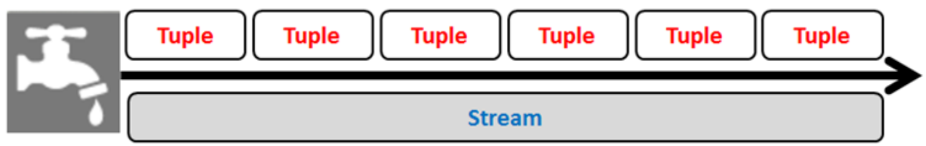
Storm Topolgies

- Storm *topology* runs many workers across many processors
  - Workers implement a subset of a topology
  - Topology is essentially a directed graph
- Topologies run forever
  - Accepting and processing message from the stream
- Topologies comprise *spouts* and *bolts*
  - Applications need to implement Storm-defined interfaces to run application-specific logic.



Storm Streams

- Core Storm abstraction
    - Uniquely identified
- Unbounded sequence of tuples
    - Spout is a source of streams
    - Bolts may transform one stream into another stream
- Defined by a schema
    - Identifies field in tuples
    - Built in types
    - Custom types require serializers
- Storms allows processing multiple streams in parallel.



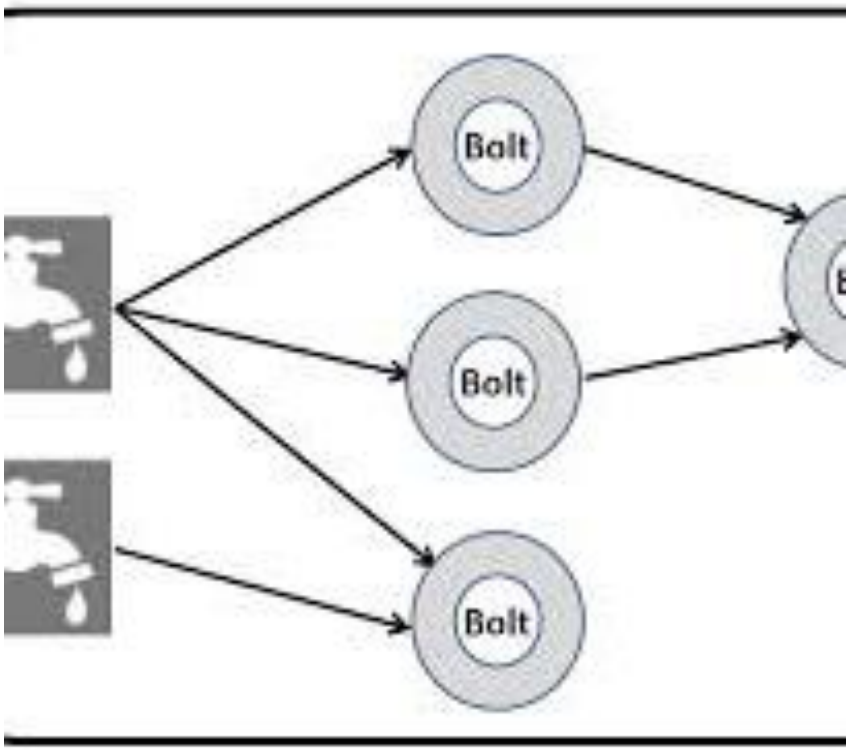
Stream Data Model - Tuples

- A tuple is a named list of values
- A field in a tuple can be an object of any type.
- Storm supports all primitive types, strings, and byte arrays as tuple field values.
- To use an object of another type, you implement a serializer
- Every node in a topology must declare the output fields for the tuples it emits.

```
public void declareOutputFields(OutputFieldsDeclarer
declarer) {
        declarer.declare(new Fields(“field1", “field2"));
 }
```

Spouts

- A spout is a source of tuples for a stream, eg:
  - Sensor messages
  - Log files
  - Databases
  - Kafka
- Reliable or unreliable
  - Reliable replays tuple if processing fails
  - Unreliable emits and forgets
- Methods
  - **nextTuple**: emits a tuple or returns Null.
  - **Ack**: acknowledge successful tuple processing
  - **Fail**: tuple failed
  - Ack and fail only valid for reliable spouts



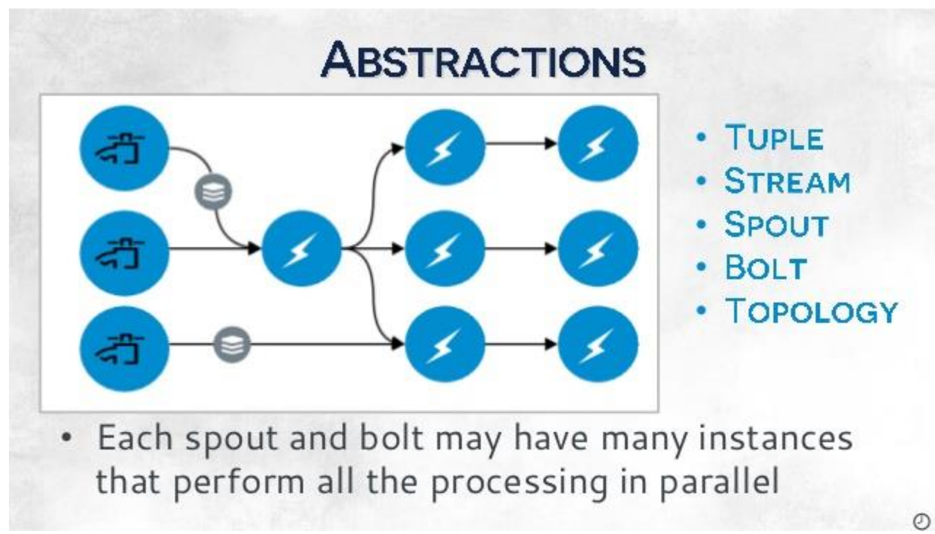
Spout Example - Unreliable

```
public class RandomNumberSpout extends BaseRichSpout {
    private Random random;
    private SpoutOutputCollector collector;
    @Override
    public void open(Map map, TopologyContext topologyContext,
      SpoutOutputCollector spoutOutputCollector) {
        random = new Random();
        collector = spoutOutputCollector;
    }
    @Override
    public void nextTuple() {
        Utils.sleep(1000);
        int operation = random.nextInt(101);
        long timestamp = System.currentTimeMillis();

        Values values = new Values(operation, timestamp);
        collector.emit(values);
    }
     @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
        outputFieldsDeclarer.declare(new Fields("operation", "timestamp"));
    }
}
```

Example from https://www.baeldung.com/apache-storm

Bolts

- A bolt consumes tuples, does computation, and may emit new tuples into a stream
  - Filtering, Transformations, Aggregations
  - Database query and store
- Bolts can be pipelined to perform complex computations in stages
- Can emit to multiple streams
- **Execute():** accepts a new input tuple
- Multiple concurrent instances



Simple Bolt Example

```
public class PrintingBolt extends BaseBasicBolt {
    @Override
   public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {
        System.out.println(tuple);
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {

    }
}
```

FilteringBolt

```
public class FilteringBolt extends BaseBasicBolt {
    @Override
    public void execute(Tuple tuple, BasicOutputCollector basicOutputCollector) {
        int operation = tuple.getIntegerByField("operation");
        if (operation > 0) {
            basicOutputCollector.emit(tuple.getValues());
        }
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {
        outputFieldsDeclarer.declare(new Fields("operation", "timestamp"));
    }
}
```

Windowing Bolts

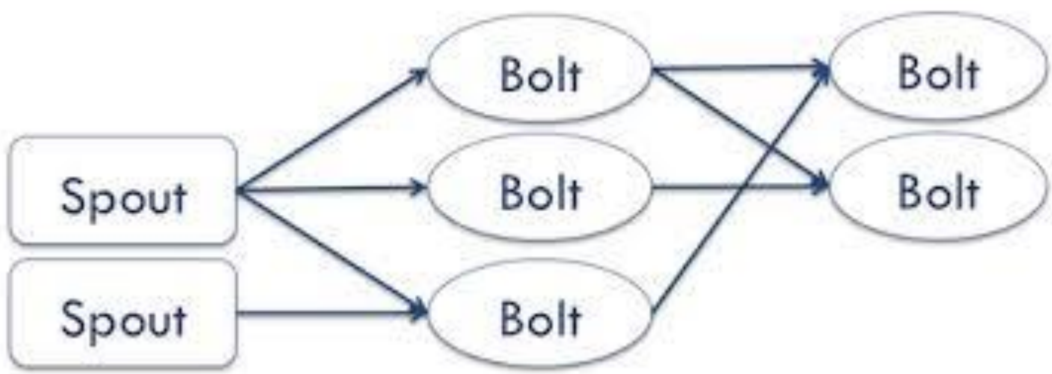
- **Time windows** are used to group elements from a given time period using timestamps.
    - Time windows may have a different number of elements.
- **Count windows** are used to create windows with a defined size.
    - all windows will have the same size
    - window will not be emitted if there are fewer elements than the defined size.

Windowing Bolt

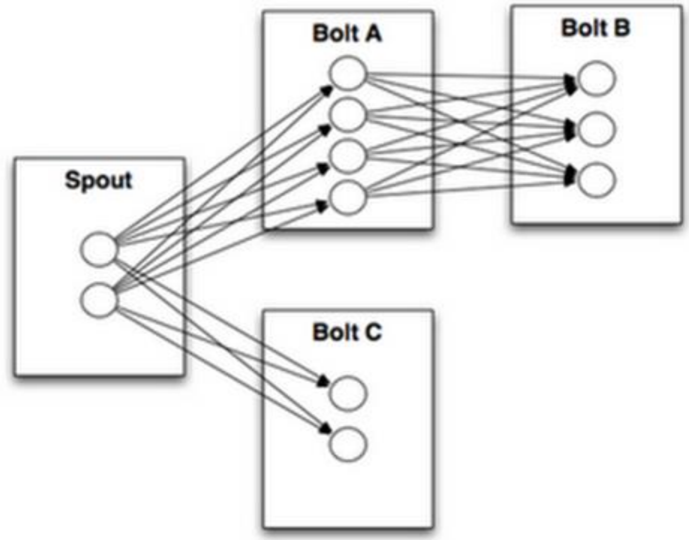
```
public class AggregatingBolt extends BaseWindowedBolt {
    private OutputCollector outputCollector;
    // prepare method omitted – initializes outputCollector
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("sumOfOperations", "beginningTimestamp", "endTimestamp"));
    }
    @Override
    public void execute(TupleWindow tupleWindow) {
        List<Tuple> tuples = tupleWindow.get();  // time window period is set in topology configuration
        tuples.sort(Comparator.comparing(this::getTimestamp));

        int sumOfOperations = tuples.stream()
          .mapToInt(tuple -> tuple.getIntegerByField("operation"))
          .sum();
        Long beginningTimestamp = getTimestamp(tuples.get(0));
        Long endTimestamp = getTimestamp(tuples.get(tuples.size() - 1));

        Values values = new Values(sumOfOperations, beginningTimestamp, endTimestamp);
        outputCollector.emit(values);
    }
    private Long getTimestamp(Tuple tuple) {
        return tuple.getLongByField("timestamp");
    }
}
```

Topologies

- A topology is a graph where every node is a spout of bolt
  - Data model based on tuples
  - Edges represent a subscription to a stream
- In a topology:
  - All nodes run in parallel
  - Parallelism configurable for every node
  - A topology never ends

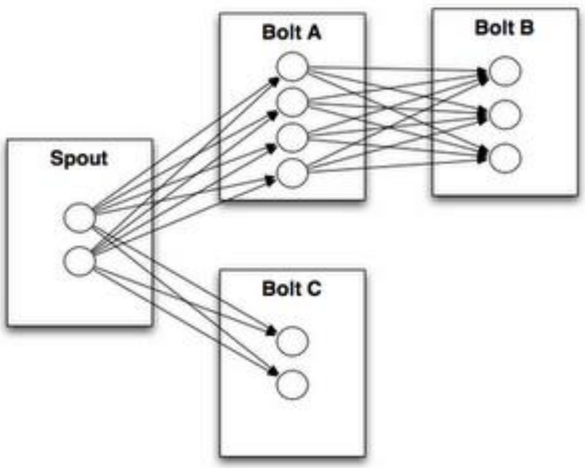


Scalability



- Each node in a Storm topology executes in parallel.
- In your topology
  - specify how much parallelism you want for each node
  - Storm spawns that number of threads across the cluster for execution.

Topology Definition



- Bolts define how they wish to distribute tuples across threads using *stream groupings, e.g.:*
  - Shuffle: random assignment of tuples to tasks
  - Field: Send data to tasks based on a value in the tuple
  - All: send all data to all tasks

Topology Example – Word Count

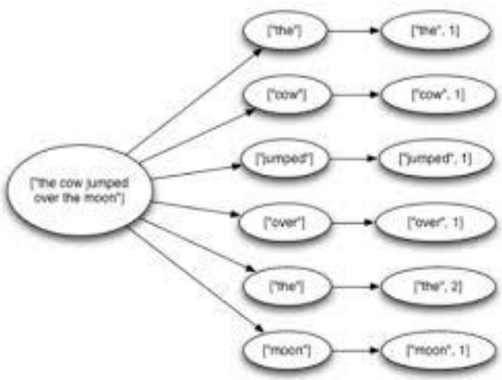
```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("sentences", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(), 8)
        .shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 12)
        .fieldsGrouping("split", new Fields("word"));
```

https://github.com/apache/storm/blob/v2.1.0/examples/storm-starter/src/jvm/org/apache/storm/starter/WordCountTopology.java

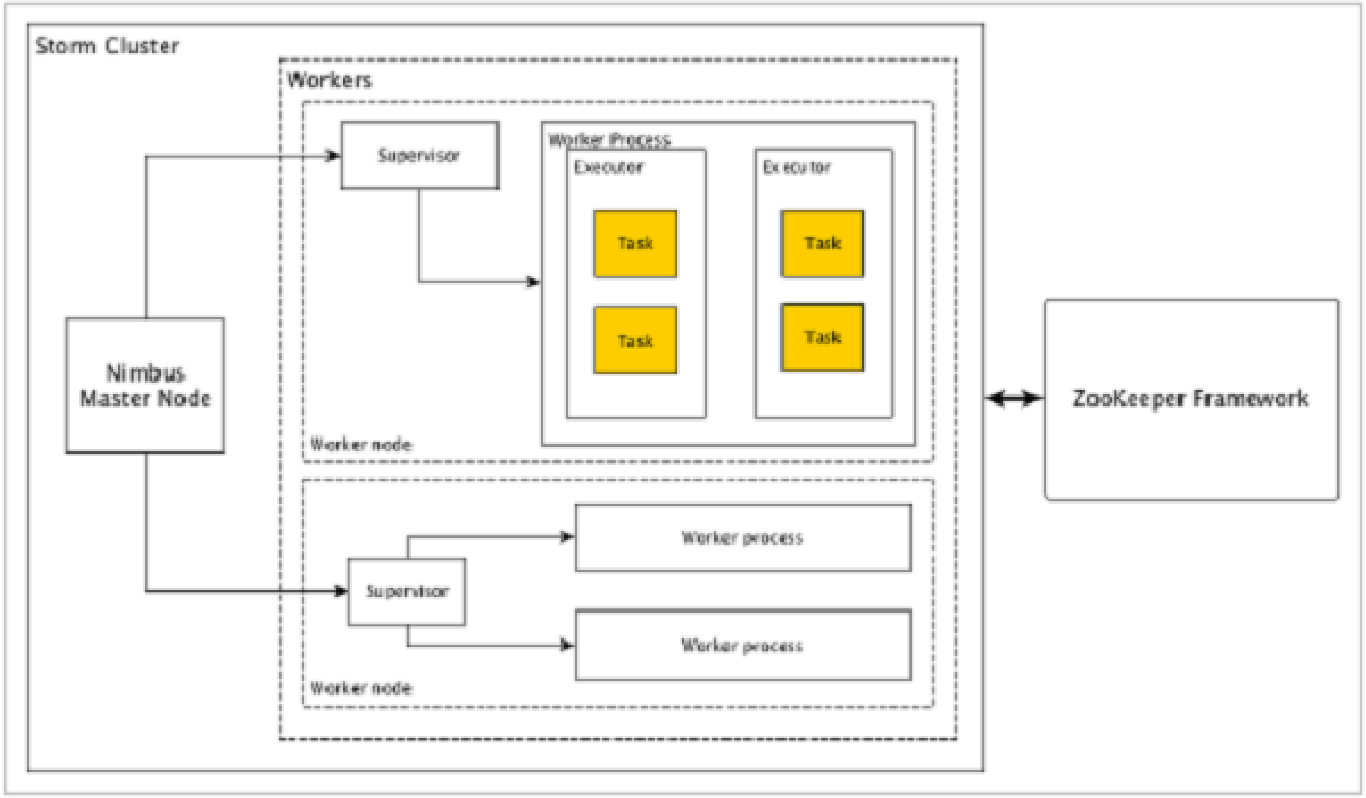
Tasks and Workers

- Spouts and bolts execute as multiple single threaded tasks in a cluster
- Number of tasks specified in TopologyBuilder
  - setSpout
  - setBolt
- Topologies execute across 1..N worker processes
  - Worker = JVM
  - Storm attempts to distribute total tasks evenly across workers
  - Eg if 100 tasks for a bolt and 10 workers, then Storm allocates 10 tasks per worker
- Configured for each topology
  - public static final String TOPOLOGY_WORKERS
  - public static final String TOPOLOGY_TASKS

Reliability



- Storm tracks the processing of every tuple from a bolt
  - Uses a tuple tree
  - Replays a message if not completed with a timeout period
- Spouts assign each tuple an id
- Id used to track processing through the topology
- Bolts must:
  - Anchor the tuples they emit to the originating tuple
    - Unanchored tuples not replayed
  - Acknowledge completion of processing for every tuple
- *How it works:*
  - *http://storm.apache.org/releases/1.0.0/Guaranteeing-message-processing.html*



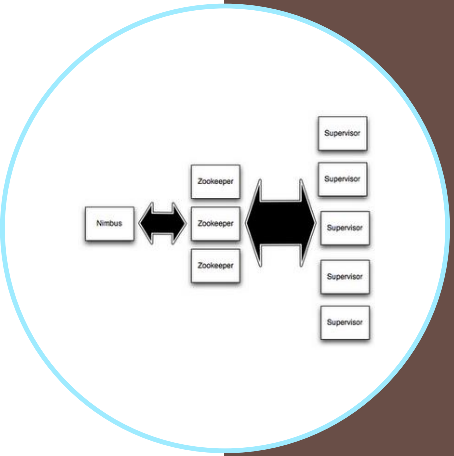
Storm Cluster Architecture

Storm Cluster Architecture

- **Nimbus** daemon
  - central component of Apache Storm.
  - analyzes the topology
  - distributing code around the cluster, assigning tasks to machines
  - monitoring for failures.
- **Supervisor** daemon.
  - listens for work assigned to its machine
  - starts and stops worker processes as necessary based on what Nimbus assignments
  - Each worker process executes a subset of a topology;
  - Spawns executor threads to run one or more tasks for a specific spout or bolt.
- a running topology consists of many worker processes spread across many machines.

Cluster Configuration

- Nimbus and supervisors are stateless
  - Fail fast and restartable
- Cluster configuration is kept in Apache ZooKeeper
  - Fault tolerant service used by a cluster for maintaining shared data/configuration
  - Nimbus cluster/job configuration
  - Supervisor local worker configuration



Storm Exercise

- Read and work through the Storm tutorial at:
  - http://storm.apache.org/releases/current/Tutorial.html

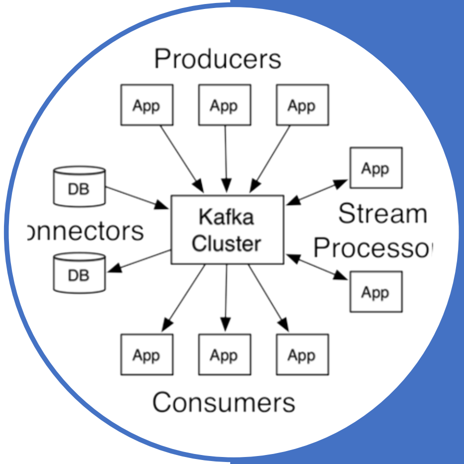
Apache Kafka

Apache Kafka

- Originally developed by LinkedIn
- Subsequently open sourced in early 2011.
- Graduation from the Apache Incubator on 23 October 2012.
- 2014, several engineers who worked on Kafka at LinkedIn created a company named Confluent with a focus on Kafka.

Kafka Overview

- Kafka stores streams of records as topics on disk
    - Key, value, timestamp
    - Publishers and subscribers
- Topics organized as a structured commit log
    - ordered, immutable sequence of records
    - continually appended to
    - records are assigned a sequential id number called the *offset* that uniquely identifies each record in the log
- 4 APIs
    - Producer
    - Consumer
    - Connector
    - Streams

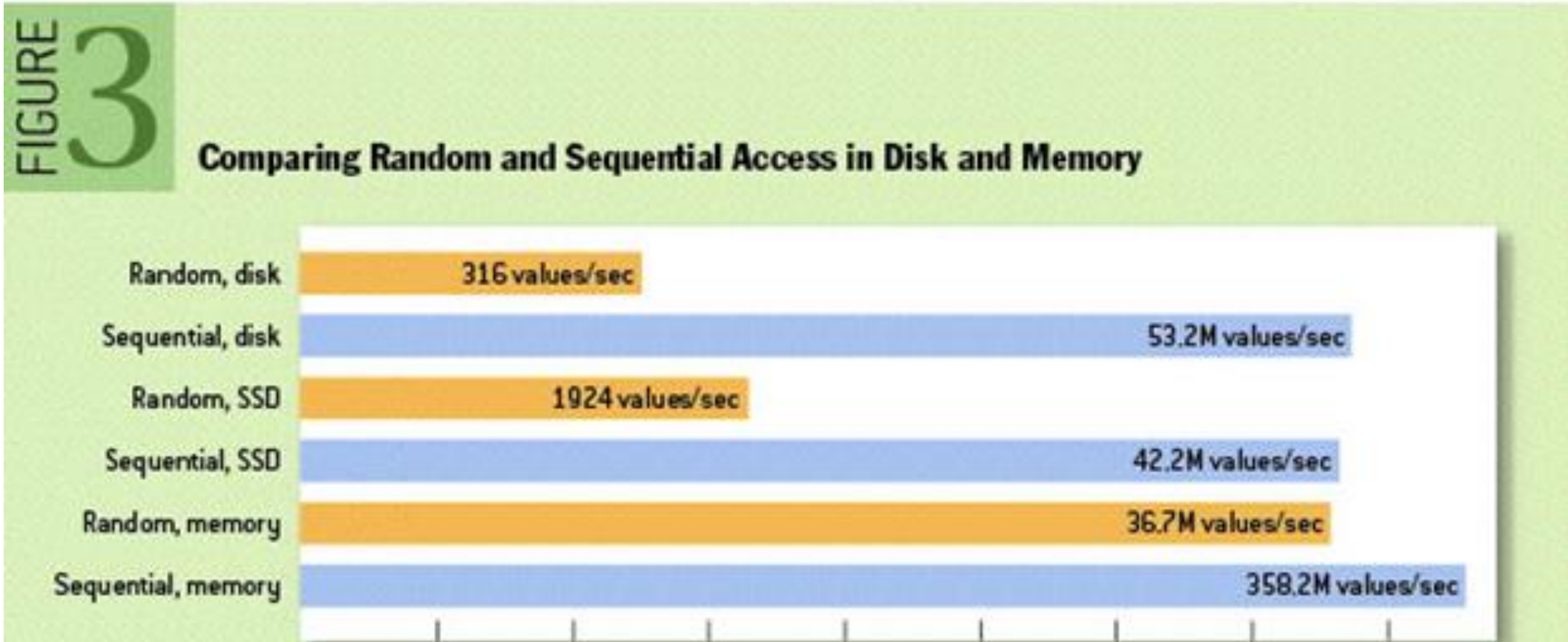


But aren't disks slow?

the performance of linear writes on a six 7200rpm SATA RAID-5 array is about 600MB/sec but the performance of random writes is only about 100k/sec—**a difference of over 6000X.**

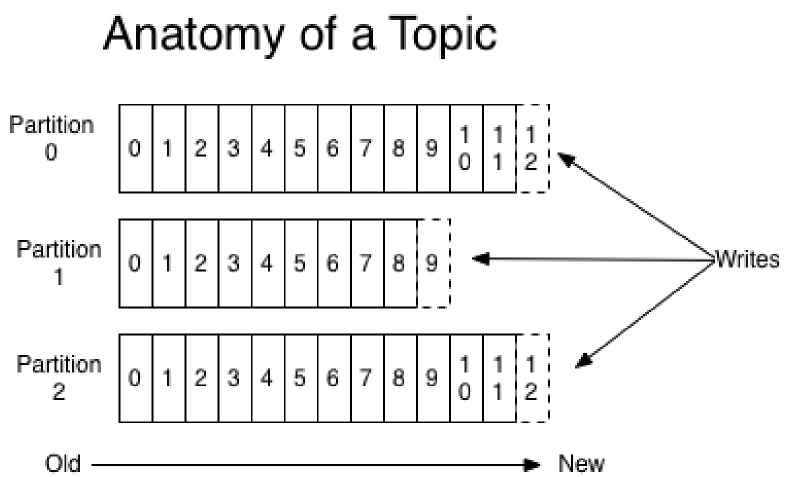
linear reads and writes are the most predictable of all usage patterns, and are heavily optimized by the operating system.

A modern operating system provides read-ahead and write-behind techniques that prefetch data in large block multiples and group smaller logical writes into large physical writes.



Note: Disk tests were carried out on a freshly booted machine (a Windows 2003 server with 64-GB RAM and eight 15,000-RPM SAS disks in RAID5 configuration) to eliminate the effect of operating-system disk caching. SSD test used a latest-generation Intel high-performance SATA SSD.

Topics and Logs



- Log organized
  - append, read, message identified by an index
  - Message exist until they are expired
- Topics can be partitioned
  - Messages distributed across partitions by a hash function/round-robin
- Partitions can be replicated
  - One leader replica
  - N follower replicas
- Publishers always send messages to leader

Producer API

```
Properties props = new Properties();

// set Producer properties
props.put("bootstrap.servers", "localhost:4242");
props.put("retries", 0);                    // no duplicates possible
props.put("batch.size", 16384);             // send buffer size – default value
props.put("linger.ms", 1);                  // wait between sending batches
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);
for(int i = 0; i < 100; i++)
    producer.send(new ProducerRecord<String, String>("my-topic",
              Integer.toString(i), Integer.toString(i)));

producer.close();
```

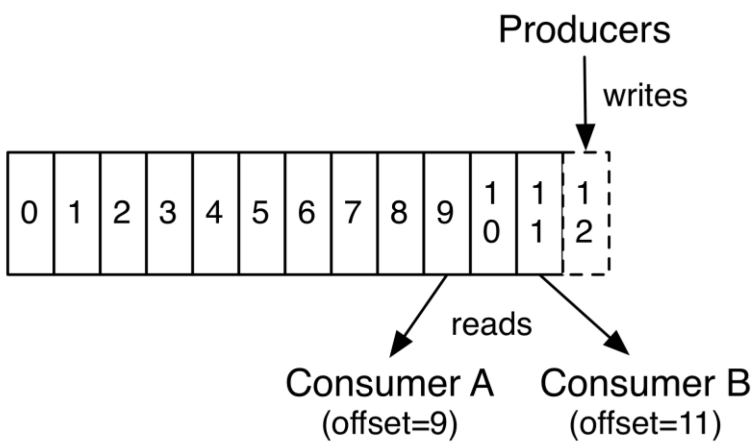
Message Send Properties

- Producer message property
  - “acks”
  - 1 (Default) – message persisted to leader
  - all– message persisted to all partitions
  - 0 – message not acknowledged (may be lost)
  - E.g.: props.put("acks", “all");
- Send() returns a RecordMetadata object
  - Future<RecordMetadata> future = producer.send(record);
  - Contains id of partition and offset of item
- Messages sent in batches
  - Batches can be compressed

Asynk Producer

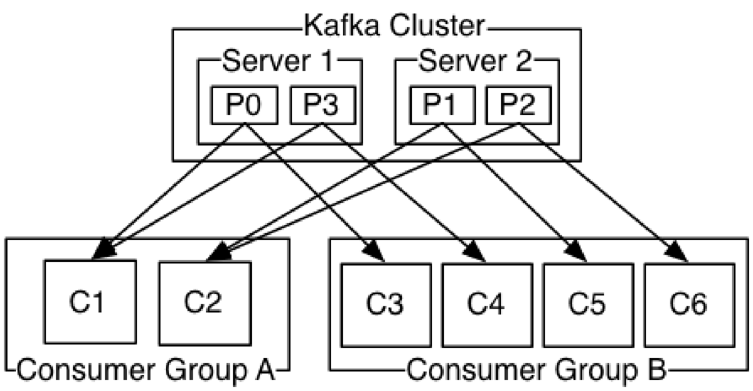
```
final ProducerRecord<K, V> record = new ProducerRecord<>(topic, key, value);
producer.send(record, new Callback() {
  public void onCompletion(RecordMetadata metadata, Exception e) {
    if (e != null)
      log.debug("Send failed for record {}", record, e);
  }
});
```

Consumers



- Consumers specify the index of message the want
- Don’t have to read messages sequentially
  - Eg go back to earlier messages to reprocess

Consumer Groups



- Consumers on separate processes/machines
- Each consumer is part of a 'consumer group'
- Each topic partition delivers messages to a single consumer group member
    - Consumer group size <= number of topic partitions
- Essentially pub-sub but with each subscriber being a consumer group
    - Scalability
    - Fault tolerance

Simple Consumer Example

```
Properties config = new Properties();
config.put("client.id", InetAddress.getLocalHost().getHostName());
config.put("group.id", "foo");
config.put("bootstrap.servers", "host1:9092,host2:9092");
props.put("enable.auto.commit", "true");
KafkaConsumer<K, V> consumer = new KafkaConsumer<>(config);

// subscribe to a topic
consumer.subscribe(topic);

while (running) {
  ConsumerRecords<K, V> records = consumer.poll(1000);
  records.forEach(record -> process(record));
  consumer.commitSync();    // commit advances the offset in the topic
}
```

Asynk Commit

```
try {
    consumer.subscribe(topic);

    while (true) {
      ConsumerRecords<K, V> records = consumer.poll(1000);
      records.forEach(record -> process(record));
      consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception exception) {
          if (e != null)
            log.debug("Commit failed for offsets {}", offsets, e);
          }
      });
    }
  } catch (WakeupException e) {
    // ignore, we're closing
  } catch (Exception e) {
    log.error("Unexpected error", e);
  } finally {
    consumer.close();
}
```

Kafka Summary

Scalable architecture

- Replication
- Fault tolerant

Compared to conventional message queue and pubsub platforms:

- Abstractions (partitions, consumer groups) make scaling easier
- Designed to exploit clusters

Additional features:

- Configurable persistence based on time-to-live
- Streaming API

Kafka Exercise

- Work through the Quick Start at:
  - https://kafka.apache.org/quickstart
- Read about Kafka Streams:
  - https://softwaremill.com/kafka-streams-how-does-it-fit-stream-landscape/
- Note Kafka uses ZooKeeper. Read about it at:
  - https://zookeeper.apache.org/doc/trunk/zookeeperOver.html
- Work through the Getting Started guide at:
  - https://zookeeper.apache.org/doc/trunk/zookeeperStarted.html

Scalable Streaming Middleware

Streams are everywhere:

- Sensors (eg wave, earthquake)
- Cloud node monitoring
- Financial transactions
- Etc

Streams need to be processed 'in flight'

- Gain 'situational awareness'
- Quickly detect anomalies
- Often merged, cleaned, summarized before storage

Many specialized streams platforms

Kafka Streaming

Overview

- simple and lightweight client library for streaming applications.
- Has no external dependencies on systems other than Apache Kafka
- Supports exactly-once processing semantics
- Offers stream processing primitives, a high-level Streams DSL and a low-level Processor API.

Example DSL API: Word Count

```
final Properties streamsConfiguration = new Properties();

// Give the Streams application a unique name.  The name must be unique in the Kafka cluster
streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-lambda-example");
streamsConfiguration.put(StreamsConfig.CLIENT_ID_CONFIG, "wordcount-lambda-example-client");

// Where to find Kafka broker(s).
streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);

// Specify default (de)serializers for record keys and for record values.
streamsConfiguration.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
streamsConfiguration.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());

// Records should be flushed every 10 seconds.
streamsConfiguration.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG, 10 * 1000);

// Use a temporary directory for storing state.
streamsConfiguration.put(StreamsConfig.STATE_DIR_CONFIG, TestUtils.tempDirectory().getAbsolutePath());

return streamsConfiguration;
```

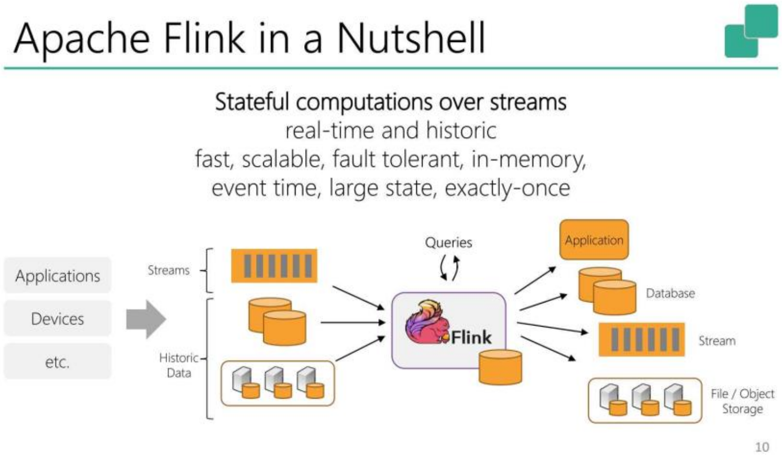
Example Word Count

```
// Define the processing topology of the Streams application.
final StreamsBuilder builder = new StreamsBuilder();
createWordCountStream(builder);
final KafkaStreams streams = new KafkaStreams(builder.build(), streamsConfiguration);

// run the processing topology via `start()` to begin processing its input data.
streams.start();

// Add shutdown hook to respond to SIGTERM and gracefully close the Streams application.
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
```

Example: Word Count

```
static void createWordCountStream(final StreamsBuilder builder) {

    // Construct a `KStream` from the input topic "streams-plaintext-input", where message values
    // represent lines of text
    final KStream<String, String> textLines = builder.stream(inputTopic);

    final Pattern pattern = Pattern.compile("\\W+", Pattern.UNICODE_CHARACTER_CLASS);

    final KTable<String, Long> wordCounts = textLines
      // Split each text line, by whitespace, into words.
     .flatMapValues(value -> Arrays.asList(pattern.split(value.toLowerCase())))
      // Group the split data by word so that we can subsequently count the occurrences per word.
     .groupBy((keyIgnored, word) -> word)
      // Count the occurrences of each word (record key).
      .count();

    // Write the `KTable<String, Long>` to the output topic.
    wordCounts.toStream().to(outputTopic, Produced.with(Serdes.String(), Serdes.Long()));
  }
```

Highly Competitive Area



Summary

- Serverless platforms available for all major cloud providers
- Aim to make deployment and management 'admin free'
- Examples GAE and AWS Lambda
- Different features and scaling approaches