CLOUD-BASED DATA STREAM PROCESSING

Thomas Heinze, Leonardo Aniello, Leonardo Querzoni, Zbigniew Jerzak

DEBS 2014
Mumbai 26/5/2014
TUTORIAL GOALS

Data stream processing (DSP) was in the past considered a solution for very specific problems.

- Financial trading
- Logistics tracking
- Factory monitoring

Today the potentialities of DSPs start to be used in more general settings.

- DSP : online processing = MR : batch processing

DSPs will possibly be offered as services from cloud-based providers?
TUTORIAL GOALS

Here we present an overview about current research trends within data streaming systems

- How they consider the requirements imposed by recent use-cases
- How are they moving toward cloud platforms

Two focus areas:
- Scalability
- Fault Tolerance
THE AUTHORS

Thomas Heinze
Phd Student @ SAP
Phd Topic: Elastic Data Stream Processing

Leonardo Aniello
Researcher fellow @ Sapienza Univ. of Rome

Leonardo Querzoni
Assistant professor @ Sapienza Univ. of Rome

Zbigniew Jerzak
Senior Researcher @ SAP
Co-Organizer of DEBS Grand Challenge

25/05/14
OUTLINE

- Historical overview of data stream processing
- Use cases for cloud-based data stream processing
- How DSPs work: Storm as example
- Scalability methods
- Fault tolerance integrated in modern DSPs
- Future research directions and conclusions
CLOUD-BASED DATA STREAM PROCESSING

Historical Overview
Data Stream Processing Engine

- It is a piece of software that
  - continuously calculates results for long-standing queries
  - over potentially infinite data streams
  - using operators
    - algebraic (filters, join, aggregation)
    - user defined
  - That can be stateless or stateful
REQUIREMENTS

**Rule 1** - Keep the data moving
**Rule 2** - Query using SQL on stream (StreamSQL)
**Rule 3** - Handle stream imperfections (delayed, missing and out-of-order data)
**Rule 4** - Generate predictable outcomes
**Rule 5** - Integrate stored and streaming data
**Rule 6** - Guarantee data safety and availability
**Rule 7** - Partition and scale applications automatically
**Rule 8** - Process and respond instantaneously

THREE GENERATIONS

- First Generation
  - Extensions to existing database engines or simplistic engines
  - Dedicated to specific use cases

- Second Generation
  - Enhanced methods regarding language expressiveness, load balancing, fault tolerance

- Third Generation
  - Dedicated towards trend of cloud computing; designed towards massive parallelization
**First Generation - Telegraph CQ** [2]

- Data stream processing engine built on top of Postgres DB

SECOND GENERATION - BOREALIS\[^{[3]}\]

- Joint research project of Brandeis University, Brown University and MIT
- Duration: ~3 years
- Allowed the experimentation of several techniques

BOREALIS MAIN NOVELTIES

- Load-aware Distribution
- Fine-grained High-availability
- Load Shedding mechanisms
THIRD GENERATION

- Novel use cases are driving toward
  - Uprecedented levels of parallelism
  - Efficient fault tolerance
  - Dynamic scalability
  - Etc.
CLOUD-BASED DATA STREAM PROCESSING

Use Cases for Cloud-Based Data Stream Processing
SCENARIOS FOR THIRD GENERATION DSPs

- Tracking of query trend evolution in Google
- Analysis of popular queries submitted to Twitter
- User profiling at Yahoo! based on submitted queries
- Bus routing monitoring and management in Dublin
- Sentiment analysis on multiple tweet streams
- Fraud monitoring in cellular telephony
Query Monitoring at Google

- Analyze queries submitted to Google search engine to create a query historical model
- Run on Zeitgeist on top of MillWheel
- Incoming searches are organized in 1-second buckets
- Buckets are compared to historical data
- Useful to promptly detect anomalies (spikes/dips)

When a relevant event happens, an increase occurs in the number of queries submitted to Twitter.

These queries have to be correlated in real-time with tweets (2.1 billion tweets per day).

Runs on Storm.

These spikes are likely to fade away in a limited amount of time.

Useful to improve the accuracy of popular queries.
Popular Query Analysis at Twitter

Problem
- Sudden peak of queries about a new (never seen before) event
- How to properly assign the right semantic (categorization) to the queries?
- Example

Solution
- Employ Human Evaluation (Amazon’s Mechanical Turk Service) to produce categorizations to queries unseen so far
- incorporate such categorizations into backend models

How can we understand that #bindersfullofwomen refers to politics?

25/05/14

Cloud-Based Data Stream Processing
Popular query analysis at Twitter

Search Engine

Kafka Queue

Spout (query, ts)

Bolt

Human Evaluation

query Log

Twitter queries

engine tuning

popular queries

categorizations of queries

Cloud-based data stream processing

25/05/14
USER PROFILING AT YAHOO!

- Queries submitted by users are evaluated (thousands queries per second by millions of users)
- Run on S4[1]
- Useful to generate highly personalized advertising

BUS ROUTING MANAGEMENT IN DUBLIN

- Tracking of bus locations (1000 buses) to improve public transportation for 1.2 million citizens
- Run on System S
- Position tracking by GPS signals to provide real-time traffic information monitoring
- Useful to predict arrival times and suggest better routes

SENTIMENT ANALYSIS ON TWEET STREAMS

- Tweet streams flowing at high rates (10K tweet/s)
- Sentiment analysis in real-time
- Limited computation time (latency up to 2 seconds)
- Run on Timestream\[1\]
- Useful to continuously estimate the mood about specific topics

Fraud Monitoring for Mobile Calls

- Fraud detection by real-time processing of Call Detail Records (10k-50k CDR/s)
- Requires self-joins over large time windows (queries on millions of CDRs)
- Run on StreamCloud[1]
- Useful to reactively spot dishonest behaviors

Requirements

- Real-time and continuous complex analysis of heavy data streams
  - More than 10k event/s
- Limited computation latency
  - Up to few seconds
- Correlation of new and historical data
  - Computations have a state to be kept
- Input load varies considerably over time
  - Computations have to adapt dynamically (Scalability)
- Hardware and Software failures can occur
  - Computations have to transparently tolerate faults with limited performance degradation (Fault Tolerance)
CLOUD-BASED DATA STREAM PROCESSING

How DSPs work: Storm as example
**STORM**

- Storm is an open source distributed real-time computation system
  - Provides abstractions for implementing event-based computations over a cluster of physical nodes
  - Manages high throughput data streams
  - Performs parallel computations on them
- It can be effectively used to design complex event-driven applications on intense streams of data
STORM

- Originally developed by Nathan Marz and team at BackType, then acquired by Twitter, now an Apache Incubator project.
- Currently used by Twitter, Groupon, The Weather Channel, Taobao, etc.
- Design goals:
  - Guaranteed data processing
  - Horizontal scalability
  - Fault Tolerance
STORM

An application is represented by a topology:
OPERATOR EXPRESSIVENESS

- Storm is designed for custom operator definition
  - Bolts can be designed as POJOs adhering to a specific interface
  - Implementations in other languages are feasible
- Trident\footnote{1} offers a more high level programming interface

Operator Expressiveness

- Operators are connected through grouping:
  - Shuffle grouping
  - Fields grouping
  - All grouping
  - Global grouping
  - None grouping
  - Direct grouping
  - Local or shuffle grouping
PARALLELIZATION

- Storm asks the developer to provide “parallelism hints” in the topology
A storm cluster is constituted by a Nimbus node and $n$ Worker nodes.
**TOPOLOGY EXECUTION**

- A topology is run by submitting it to Nimbus
  - Nimbus allocates the execution of components (spouts and bolts) to the worker nodes using a scheduler
    - Each component has multiple instances (parallelism)
    - Each instance is mapped to an executor
  - A worker is instantiated whenever the hosting node must run executors for the submitted topology
  - Each worker node locally manages incoming/outgoing streams and local computation
    - The local supervisor takes care that everything runs as prescribed
  - Nimbus monitors worker nodes during the execution to manage potential failures and the current resource usage
Storm’s default scheduler (EvenScheduler) applies a simple round robin strategy.
A PRACTICAL EXAMPLE

- Word count: the HelloWorld for DSPs
- Input: stream of text (e.g. from documents)
- Output: number of appearance for each word

Source: http://wpcertification.blogspot.it/2014/02/helloworld-apache-storm-word-counter.html
A practical example

- LineReaderSpout: reads docs and creates tuples

```java
public class LineReaderSpout implements IRichSpout {
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        this.context = context;
        this.fileReader = new FileReader(conf.get("inputFile").toString());
        this.collector = collector;
    }
    public void nextTuple() {
        if (completed) {
            Thread.sleep(1000);
        }
        String str; BufferedReader reader = new BufferedReader(fileReader);
        while ((str = reader.readLine()) != null) {
            this.collector.emit(new Values(str), str);
            completed = true;
        }
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("line"));
    }
    public void close() {
        fileReader.close();
    }
}
```
A PRACTICAL EXAMPLE

- WordSplitterBolt: cuts lines in words

```java
public class WordSplitterBolt implements IRichBolt {

    public void prepare(Map stormConf, TopologyContext context, OutputCollector c) {
        this.collector = c;
    }

    public void execute(Tuple input) {
        String sentence = input.getString(0);
        String[] words = sentence.split(" ");
        for(String word: words){
            word = word.trim();
            if(!word.isEmpty()){
                word = word.toLowerCase();
                collector.emit(new Values(word));
            }
        }
        collector.ack(input);
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```
A PRACTICAL EXAMPLE

- WordCounterBolt: counts word occurrences

```java
public class WordCounterBolt implements IRichBolt {

    public void prepare(Map stormConf, TopologyContext context, OutputCollector c) {
        this.counters = new HashMap<String, Integer>();
        this.collector = c;
    }

    public void execute(Tuple input) {
        String str = input.getString(0);
        if (!counters.containsKey(str)) {
            counters.put(str, 1);
        } else {
            Integer c = counters.get(str) + 1;
            counters.put(str, c);
        }
        collector.ack(input);
    }

    public void cleanup() {
        for (Map.Entry<String, Integer> entry : counters.entrySet()) {
            System.out.println(entry.getKey() + " : " + entry.getValue());
        }
    }
}
```

25/05/14
A Practical Example

- HelloStorm: contains the topology definition

```java
public class HelloStorm {

    public static void main (String[] args) throws Exception{
        Config config = new Config();
        config.put("inputFile", args[0]);
        config.setDebug(true);
        config.put(Config.TOPOLOGY_MAX_SPOUT_PENDING, 1);

        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("line-reader-spout", new LineReaderSpout());
        builder.setBolt("word-spitter", new WordSpitterBolt()).shuffleGrouping(
            "line-reader-spout");
        builder.setBolt("word-counter", new WordCounterBolt()).shuffleGrouping(
            "word-spitter");

        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("HelloStorm", config, builder.createTopology());
        Thread.sleep(10000);

        cluster.shutdown();
    }
}
```
Cloud-based Data Stream Processing

Scalability
PARTITIONING SCHEMES

- Data Parallelism:
  - How to parallelize the execution of an operator?
  - How to detect the optimal level of parallelization?

- Operator Distribution:
  - How to distribute the load across available hosts?
  - How to achieve a load balance between these machines?
**Requirements for Data Parallelism**

- Transparent to the user
  - correct results in correct order (identical to sequential execution)
DATA PARALLELISM

- First presented by FLUX[1] and Borealis[2]
  - Explicitely done using partitioning operators
  - User needs to decide:
    - partitioning scheme
    - merging scheme
    - level of parallelism

PARALLELISM FOR CLOUD-BASED DSP

- Massive parallelization (>100 partitions)
- Support custom operators
- Adapt parallelization level to the workload without user interaction
DATA PARALLELISM IN STORM

- User defines number of parallel task
- Storm support different partitioning schemes (aka grouping):
  - Shuffle grouping, Fields grouping, All grouping, Global grouping, None grouping, Direct grouping, Local or shuffle grouping, Custom
MAPREDUCE FOR STREAMING [3,4]

- Extend MapReduce model for streaming data:
  - Break strict phases
  - Introduce Stateful Reducer

**STATEFUL REDUCER**\[4\]

```java
reduceInit(k1) {
    // custom user class
    S = new State(); S.sum = 0; S.count = 0;
    // object S is now associated with key k1
    return S;
}
reduce(Key k, <List new, List expired, List window, UserState S>) {
    For v in expired do:
        // Remove contribution of expired events
        S.sum -= v; S.count--;

    For v in new do:
        // Add contribution of new events
        S.sum += v; S.count++;
    send(k1, S.sum/S.count);
}
```

**Auto Parallelization**\([5,6]\)

- **Goal:**
  - detect parallelizable regions in operator graph with custom operators for user-defined operators
  - Runtime support for enforcing safety conditions

- **Safety Conditions:**
  - For an operator: no state or partitioned state, selectivity < 1, at most one pre/successor
  - For an parallel region: compatible keys, forwarded keys, region-local fusion dependencies

---

**Auto Parallelization**

- Compiler-based Approach:
  - Characterize each operator based on a set of criterias (state type, selectivity, forwarding)
  - Merge different operators together into parallel regions (based on left-first approach)

- Best parallelization strategy is applied automatically
- Level of parallelism is decided on job admission
ELASTICITY[7]

- Goal: React to unpredicted load peaks & reduce the amount of ideling resources.

DIFFERENT TYPES OF ELASTICITY

- Vertical Elasticity (Scale up)
  - Adapt the number of threads per operator based on the workload.

- Horizontal Elasticity (Scale out)
  - Adapt the number of hosts based on the workload.
Elastic Operator Execution[8]

- Dynamically adapt number of threads for a stateless operator based on the workload
- Limitations: works only on thread-level and for stateless operators

Elastic Auto-Parallelization[9]

- Combines ideas of Elasticity and Auto-Parallelization
- Adapt parallelization level using controller-based approach
- Dynamic adaption of parallelization level based on operator migration protocol

Elastic Auto-Parallelization

- Dynamic adaption of parallelization: lend & borrow approach
**Operator Distribution**

- Well-studied problem since first DSP prototypes
- Typically referred to as „Operator Placement“
- Examples: Borealis, SODA, Pietzuch et al.

- Design decisions[10]
  - Optimization goal: What is optimized?
  - Execution mode: Centralized or Decentralized?
  - Algorithm runtime: Offline, online or both?

---

Optimization Goal

- Different objectives:
  - Balance CPU load (handle load variations)
  - Minimize Network latency (minimize latency for sensor networks)
  - Latency optimization (predictable quality of service)
Execution Mode

- Centralized Execution:
  - All decision done by centralized management component
  - Major disadvantage: manager becomes scalability bottleneck

- Decentralized Execution:
  - Different hosts try to agree on an operator placement
  - Two examples: operator routing and commutative execution
Decentralizer Execution

- Based on pairwise agreement[11]

DECENTRALIZER EXECUTION

- Based on decentralized routing\cite{12}

Algorithm Runtime

- Offline
  - Based on estimation of input rates, selectivities, etc.

- Online
  - Mostly simplistic initial estimation (e.g. round-robin or random placement)
  - Continuously measurements and adaption during runtime
MOVEMENT PROTOCOLS

- How to ensure loss-free movement of operators?
  - No input event shall be lost
  - State need to be restored on new host

- Movement strategies:
  - Pause & Resume vs. Parallel track
  - large overlap with research on adaptive query processing\cite{Zhu2004}

PAUSE & RESUME

- Approach:
  1. Stop execution
  2. Move state to new host
  3. Restart execution on new hosts

- Properties:
  - Simple & generic
  - Latency Peak can be observed due to pausing
PARALLEL TRACK

Approach:
1. Start new instance
2. Move or create up to date state
3. Stop old instance as soon as instances are in sync

Properties:
- No latency peak
- Requires duplicate detection, detection of „sync“ status
- Events processed twice
OPERATOR PLACEMENT WITHIN CLOUD-BASED DSP

- Different setup (highly location-wise distributed vs. single cluster)
- Larger scale (100 …. 1000 hosts)
- New objectives: Elasticity, Monetary Cost, Energy Efficiency

 Mostly centralized, adaptive solutions optimizing CPU utilization or monetary cost with Pause & Resume Operator Movement.
RECENT OPERATOR PLACEMENT APPROACHES

- **SQPR**[^14]
  - Query planner for data centers with heterogeneous resources
- **MACE**[^15]
  - Present san approach for precise latency estimation

[^14]: V. Kalyvianaki et al. „SQPR: Stream query planning with reuse“ In ICDE, 2011.
[^15]: B. Chandramouli et al. „Accurate latency estimation in a distributed event processing system.“ In ICDE, 2011.
STORM LOAD MODEL

- **Worker**: physical JVM and executes subset of all the tasks of the topology
- **Task**: Parallel instance of an operator
- **Executor**: Thread of an worker, executes one or more tasks of the same operator

Example for Storm Load Model

```java
Config conf = new Config();
conf.setNumWorkers(3); // use two worker processes
topologyBuilder.setSpout("src", new Source(), 3);

topologyBuilder.setBolt("aggr", new Aggregation(), 6).shuffleGrouping("src");

topologyBuilder.setBolt("sink", new Sink(), 6).shuffleGrouping("aggr");

StormSubmitter.submitTopology( "mytopology", conf, topologyBuilder.createTopology() );
```

**Storm: Distribution Model**[16]

Parallelism hint = 3

Source (spout) → Aggr (bolt) → Sink (bolt)

Parallelism hint = 6

Parallelism hint = 3

Worker 1

Worker 2

Worker 3

ADAPTIVE PLACEMENT IN STORM

- Manual reconfiguration (pause & resume complete topology)

$ storm rebalance mytopo -n 4 -e src=8

- Enhanced Load scheduler for Twitter Storm[17]
  - Load balancing adapted to Storm architecture

### DIFFERENT LEVELS OF ELASTICITY

<table>
<thead>
<tr>
<th>Elastic</th>
<th>Action Taken</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Machine</td>
<td>Move virtual machines transparent to the user.</td>
<td>[18]</td>
</tr>
<tr>
<td>Engine</td>
<td>New engine instance is started and new queries are employed on this engine</td>
<td>[19]</td>
</tr>
<tr>
<td>Query</td>
<td>New query instance is started and the input data is split</td>
<td>[20]</td>
</tr>
<tr>
<td>Operator</td>
<td>New operator instance is created and the input data is split</td>
<td>[21]</td>
</tr>
</tbody>
</table>

ELASTICS DSP SYSTEM ARCHITECTURE

Virtual Machine

Virtual Machine

Virtual Machine

Resource Manager

Elasticity Manager

Data Stream Processing Engine

Data Stream Processing Engine

Data Stream Processing Engine
STREAMCLOUD\textsuperscript{[22]}

- Realized different level of elasticity on a highly scalable DSP
- Studied different major design decisions:
  - Optimal Level of Elasticity
  - Migration strategy
  - Load balancing based on user-defined thresholds

SEEP\textsuperscript{[21]}

- Present the state of the art mechanisms for state management
- combine fault tolerance & scale out using same mechanism
- Highly scalable and fast recovery

**MillWheel**\(^{[23]}\)

- Similar mechanisms like SEEP or StreamCloud
  - Support massive scale out
  - Centralized manager
  - Optimization of CPU Load

---

CLOUD-BASED DATA STREAM PROCESSING

Fault tolerance
Fault tolerance in DSPs

- Small scale stream processing
  - Faults are an exception
  - Optimize for the lucky case
  - Catch faults at execution time and start recovery procedures (possibly expensive)

- Large scale DSPs (e.g. cloud based)
  - Faults are likely to affect every execution
  - Consider them in the design process
Fault Tolerance in DSPs

- Two main fault causes
  - Message losses
  - Computational element failures
- Event tracking
  - Makes sure each injected event is correctly processed with well-defined semantics
- State management
  - Makes sure that the failure of a computational element will not affect the system’s correct execution
**EVENT TRACKING**

- When a fault occurs, events may need to be replayed
- **Typical approach:**
  - Request acks from downstream operators
  - Detect losses by setting timeouts on acks
  - Replay lost events from upstream operators
- Tracking and managing information on processed events may be needed to guarantee specific processing semantics
Storm – ET

- Event tracking in storm is guaranteed by two different techniques:
  - Acker processes ➔ at-least-once message processing
  - Transactional topologies ➔ exactly-once message processing
STORM – ET

- A tuple injected in the system can cause the production of multiple tuples in the topology
- This production partially maps the underlying application topology
- Storm keeps track of the processing DAG stemming from each input tuple
Example: word-count topology

```
"stay hungry, stay foolish"
```

```
["stay", "hungry", "stay", "foolish"]
```

```
["stay", 2]
["hungry", 1]
["foolish", 1]
```

```
"stay"
"hungry"
"stay"
"foolish"
```

```
["stay", 1]
["hungry", 1]
["stay", 2]
["foolish", 1]
```
STORM – ET

- Acker tasks are responsible for monitoring the flow of tuples in the DAG
  - Each bolt “acks” the correct processing of a tuple
  - The processing of a tuple can be committed when it has fully traversed the DAG
  - In this case the acker notifies the original spout
- The spout must implement an ack(Object msgId) method
  - It can be used to garbage collect tuple state
STORM – ET

- If a tuple does not reach the end of the DAG
  - The acker timeouts and invoke fail(Object msgId) on the spout
  - The spout method implementation is in charge of replaying failed tuples
- Notice: the original data source must be able to reliably reply events (e.g. a reliable MQ)
STORM — ET

- Developer perspective:
  - Correctly connect bolts with tuple sources (anchoring)
  - Anchoring is how you specify the tuple tree

```java
public void execute(Tuple tuple) {
    String sentence = tuple.getString(0);
    for (String word: sentence.split(" ")) {
        _collector.emit(tuple, new Values(word));
    }
    _collector.ack(tuple);
}
```

Source: https://github.com/nathanmarz/storm/wiki/Guaranteeing-message-processing
STORM – ET

- Developer perspective:
  - Explicitly _ack_ processed tuples

```java
public void execute(Tuple tuple) {
  String sentence = tuple.getString(0);
  for (String word: sentence.split(" ")) {
    _collector.emit(tuple, new Values(word));
  }
  _collector.ack(tuple);
}
```

- Or extend _BaseBasicBolt_

Source: https://github.com/nathanmarz/storm/wiki/Guaranteeing-message-processing

Explicit ack
**STORM – ET**

- Implement `ack()` and `fail()` on the spout(s)

```java
public void nextTuple() {
    if (!toSend.isEmpty()) {
        for (Map.Entry<Integer, String> transactionEntry : toSend.entrySet()) {
            Integer transactionId = transactionEntry.getKey();
            String transactionMessage = transactionEntry.getValue();
            collector.emit(new Values(transactionMessage), transactionId);
        }
        toSend.clear();
    }
}

public void ack(Object msgId) {
    messages.remove(msgId);
}

public void fail(Object msgId) {
    Integer transactionId = (Integer) msgId;
    Integer failures = transactionFailureCount.get(transactionId) + 1;
    if (failures >= MAX_FAILS) {
        throw new RuntimeException("Too many failures on Tx [" + transactionId + "]");
    }
    transactionFailureCount.put(transactionId, failures);
    toSend.put(transactionId, messages.get(transactionId));
}
```

STORM – ET

- How does the acker task work?
  - It is a standard bolt
  - The acker tracks for each tuple emitted by a spout
  - The corresponding DAG
  - It acks the spout whenever the DAG is complete
- You can instantiate parallel ackers to improve performance
  - Tuples are randomly assigned to ackers to improve load balancing (uses mod hashing)
STORM – ET

How can ackers keep track of DAGs?

- Each tuple is identified by a unique 64bit ID
- The acker stores in a map for each tuple IP
  - The ID of the emitter task
  - An ack val

An ack val is a bit-vector that encodes

- IDs of tuples stemmed from the initial one
- IDs of acked tuples
Ack val management

- When a tuple is emitted by a spout
  - Initialize the vector an encode in it the tuple ID
- When a tuple is acked
  - XOR its ID in the ack val
- When an anchored tuple is emitted
  - XOR its ID in the ack val
- When the ack val is empty, all tuples in the DAG have been acked (with high probability)
STORM — ET

- If exactly-once semantics is required use a *transactional topology*
  - Transaction = processing + committing
  - Processing is heavily parallelized
  - Committing is strictly sequential

- Storm takes care of
  - State management (through Zookeeper)
  - Transaction coordination
  - Fault detection
  - Provides a batch processing API

- Note: requires a source able to reply data batches

Source: https://github.com/nathanmarz/storm/wiki/Transactional-topologies
Event Tracking

- In other systems the event tracking functionality is strongly coupled with state management
  - events cannot be garbage collected when they are acknowledged
  - Need to wait for the checkpointing of a state updated with such events
- Timestream does not store all the events and recompute those to be replayed by tracking their dependencies with input events, similarly to Storm
**Event Tracking**

- SEEP stores non-checkpointed events on the upstream operators.
- Millwheel persists all intermediate results to an underlying Distributed File System.
  - It also provides exactly-once semantics.
- D-Streams stores data to be processed in immutable partitioned datasets.
  - These are implemented as resilient distributed datasets (RDD).
Stateful operators require their state to be persisted in case of failures.

Two classic approaches

- Active replication
- Passive replication
State Management

Active replication

State implicitly synchronized by ordered evaluation of same data

Passive replication

State periodically persisted on stable storage and recovered on demand
What happen when tasks fail?

- If a worker dies its supervisor restarts it
  - If it fails on startup Nimbus will reassign it on a different machine
- If a machine fails its assigned tasks will timeout and Nimbus will reassign them
- If Nimbus/Supervisors die they are simply restarted
  - Behave like *fail-fast* processes
  - They’re *stateless* in practice
  - Their state is safely maintained in in a Zookeeper cluster
STORM - SM

- There is no explicit state management for operators in Storm
- Trident builds automatic SM on top of it
  - Batch of tuples have unique Tx id
  - If a batch is retried it will have the exact same Tx id
  - State updates are ordered among batches
    - A new Tx is not committed if an old one is still pending
- Transactional state guarantees transparently exactly-once tuple processing
STORM - SM

- Transactional state is possible only if supported by the data source
- As an alternative
  - Opaque transactional state
    - Each tuple is guaranteed to be executed exactly in one transaction
    - But the set of transactions for a given Tx id can change in case of failures
  - Non transactional state
STORM - SM

- Few combinations guarantee exactly-once sem.

<table>
<thead>
<tr>
<th>Spout</th>
<th>State</th>
<th>Non transactional</th>
<th>Transactional</th>
<th>Opaque transactional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non transactional</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Transactional</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Opaque transactional</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
**APACHE S4 - SM**

- Lightweight approach
  - Assumes that lossy failovers are acceptable.
  - PEs hosted on failed PNs are automatically moved to a standby server.
  - Running PEs periodically perform uncoordinated and asynchronous checkpointing of their internal state
    - Distinct PE instances can checkpoint their state autonomously without synchronization
      - No global consistency
    - Executed asynchronously by first serializing the operator state and then saving it to stable storage through a pluggable adapter
  
- Can be overridden by a user implementation
Millwheel - SM

- Guarantees strongly consistent processing
  - Checkpoints on persistent storage every single state change incurred after a computation
  - Can be executed
    - before emitting results downstream
      - operator implementations are automatically rendered idempotent with respect to the execution
    - after emitting results downstream
      - it’s up to the developer to implement idempotent operators if needed
  - Produced results are checkpointed with state (strong productions)
T i m e S t r e a m - S M

- Takes a similar approach
  - State information checkpointed to stable storage
  - For each operator the state includes
    - state dependency: the list of input events that made the operator reach a specific state
    - output dependency: the list of input events that made an operator produce a specific output starting from a specific state.
  - Allow to correctly recover from a fault without having to store all the intermediate events produced by the operators and their states.
  - Is it possibile to periodically checkpoint a full operator state in order to avoid re-emitting the whole history of input events in order to recompute it.
SEEP - SM

- Takes a different route allowing state to be stored on upstream operators
  - allows SEEP to treat operator recovery as a special case of a standard operator scale-out procedure
  - state in SEEP is characterized by three elements
    - internal state
    - output buffers
    - routing state
  - treated differently to reduce the state management impact on system performance.
SM depends strictly on the computation model

- A computation is structured as a sequence of deterministic batch computations on small time intervals.
- The input and output of each batch, as well as the state of each computation, are stored as reliable distributed datasets (RDDs).
- For each RDD, the graph of operations used to compute (its lineage) it is tracked and reliably stored.
- The recovery of an RDD can be performed in parallel on separate nodes in order to speed up recovery operation.
- Operator state can be optionally checkpointed on stable storage to limit the number of operations required to restore it.
Cloud-based Data Stream Processing

Open research directions
CONCLUSIONS

- A third generation of DPSs is coming out that promise
  - Unprecedented computational power through horizontal scalability
  - On-demand dynamic load adaptation
  - Simplified programming models through powerful event management semantics
  - Graceful performance degradation in case of faults
- A few issues remain to be solved to make DSP ready for the cloud-era
INFRASTRUCTURE AWARENESS

- Most existing DSP systems are infrastructure oblivious
  - deployment strategies do not take into account the peculiar characteristics of the available hardware
- The physical connection and relationship among infrastructural element is known to be a key factor to both improve system performance and fault tolerance.
  - E.g. Hadoop's “rack awareness”
- We think infrastructure awareness is an open research field for data stream processing systems that could possibly bring important improvements.
Cost-Efficiency

- Users in these days are not only interested in the performance of the system, but also the monetary cost.
- Cloud-based DSP systems should consider running cost as a variable for performance optimization:
  - Efficient scaling behavior maximizing the system utilization
  - Efficient fault tolerance mechanisms
- Bellavista et al. [1] proposed a first prototype, which allows the user to trade-off monetary cost and fault tolerance.
  - Their prototype only selects a subset of the operators for replication based on a user-defined value for the expected information completeness.

**Energy-Efficiency**

- Most large-scale datacenters are striving to "go green" by making energy consumption more effective.
- DSP systems should consider energy consumption a yet-another-variable in their performance optimization process.
- Note: this aspect is possibly strictly linked to the infrastructure awareness theme.
**Advanced Elasticity**

- Most of the existing elastic scaling solutions for DSPs apply simplistic schemes for load balancing.
  - Operator placement algorithms only optimize system utilization
  - Other metrics (end to end latency, network bandwidth, etc.) are only partially considered
- However these metric are often used to sign contract-binding SLAs
- Would it be possible to design DSP systems able to probabilistically guarantee performance?
**MULTI-DSP INTEGRATION**

- Most DSPs are particularly well suited for specific use cases
- No one-size-fits-all solution
- Components automatically selecting the best engine for each give use case would significantly improve the applicability of these systems.
  - No need to know in advance which is the best solution for given use case
  - No need to deploy and maintain different solutions
- First results are promising [2,3]

MULTI-DSP INTEGRATION

1. New Query
   With Window Size $w$

2. Choose DSP

Thank You!

Q & A