Drinking from the Garden Hose: Stateful Stream Processing in the Cloud

Peter Pietzuch
prp@doc.ic.ac.uk

(joint work with Raul Castro Fernandez, Matteo Migliavacca, Eva Kalyvianaki)

Large-Scale Distributed Systems Group
Department of Computing
Imperial College London
http://lsds.doc.ic.ac.uk

Lisbon - May 2014
The Big Data Deluge is here...

1200 Exabytes (billion GBs) created in 2010 alone
- Increased from 150 Exabytes in 2005

Many new sources of data become available
- Sensors, mobile devices
- Web feeds, social networking
- Cameras
- Databases
- Scientific instruments

How can we make sense of all data?
- Most data is not interesting
- New data supersedes old data
Real Time Traffic Monitoring

Instrumenting a country’s transportation infrastructure

Many parties interested in data
- Road authorities, traffic planners, emergency services, commuters

High-level queries
- “What is the best time/route for my commute through central London between 7-8am?”
Web/Social Feed Mining

Social Cascade Detection

Detection and reaction to social cascades
Fraud Detection

How to detect identity fraud as it happens?

Illegal use of mobile phone, credit card, etc.
  – Offline: avoid aggravating customer
  – Online: detect and intervene

Huge volume of call records

More sophisticated forms of fraud
  – e.g. insider trading

Supervision of laws and regulations
  – e.g. Sabanes-Oxley, real-time risk analysis
Stream Processing to the Rescue!

Process data streams on the fly without storage

Stream data rates can be high
- High resource requirements for processing (clusters, data centres)

Processing stream data has real-time aspect
- Latency of data processing matters
- Must be able to react to events as they occur
State in Stream Processing

Consider a streaming recommender application (collaborative filtering)

Stream Processing System (eg Twitter Storm, Yahoo S4,...)

User Activities
(eg item purchases, page views, clicks, ...)

Processing State

Recommendations

Most online machine learning algorithms require state
Outline of the Talk

Stream Processing Motivation

Challenges: Stateful Stream Processing in Cloud Environments

SEEP: Explicit State in Stateful Stream Processing

Evaluation Results

Conclusions
Stream Processing in the Cloud

Clouds provide virtually infinite pools of resources
- Fast and cheap access to new machines for operators

How do you decide on the optimal number of VMs?
- Needlessly overprovisioning system is expense
- Using too few nodes leads to poor performance
Challenge 1: Elastic Data-Parallel Processing

Typical stream processing workloads are bursty

High + bursty input rates $\Rightarrow$ Detect bottleneck + parallelise
Challenge 2: Fault-Tolerant Processing

Large scale deployment \rightarrow Handle node failures

Failure is a common occurrence
- Active fault-tolerance requires 2x resources
- Passive fault-tolerance leads to long recovery times
State Complicates Things...

1. Dynamic scale out impacts state

2. Recovery from failures

Partitioning of state

Loss of state after node failure
Current Approaches for Stateful Processing

**Stateless** stream processing systems (eg Yahoo S4, Twitter Storm, ...)
- **Developers manage state**
- Typically combine with external system to store state (eg Cassandra)
- Design complexity

**Relational** stream processing systems (eg Borealis, Stream)
- State is **window** over stream
- No support for arbitrary state
- Hard to realise complex ML algorithms
Outline of the Talk

Stream Processing Motivation

Challenges:
Stateful Stream Processing in Cloud Environments

**SEEP: Explicit State in Stateful Stream Processing**
- State management primitives
- Integrated mechanism for scale out and failure recovery
- SEEP Stream Processing Platform

Evaluation Results

Conclusions
Idea: State as First Class Citizen

- Expose operator state as external entity so that it can be managed by stream processing system.

Operators have direct access to state

System manages state
Operators can maintain arbitrary state

State management primitives to:
- Backup and recover state
- Partition state

Integrated mechanism for scale out and failure recovery
- Operator recovery and scale out equivalent from state perspective

Example: Streaming Recommender Application

User: “BB”
Item: “iPad”
Rating: 5

User: "Peter"
Item: "iPad"
Rating: 3

User: "BB"
Item: "iPad"
Rating: 5

Rec: "iPhone"
What is State?

**Processing state**

<table>
<thead>
<tr>
<th>Item 1</th>
<th>Item 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>User A</td>
<td>2</td>
</tr>
<tr>
<td>User B</td>
<td>4</td>
</tr>
</tbody>
</table>

**Routing state**

*Dynamic data flow graph:*
Based on data, A ➔ B or A ➔ C

**Buffer state**

- Data ts1
- Data ts2
- Data ts3
- Data ts4

User

- Item 1
- Item 2

2

4
State Management Primitives

- **Checkpoint**
  - Makes state available to system
  - Attaches *last processed tuple timestamp*

- **Partition**
  - Moves copy of state from one operator to another

- **Backup**
  - Splits state to scale out an operator

- **Restore**
  - Splits state to scale out an operator
State Primitives: Backup and Restore

Backup

Checkpoint

A

B

Data t1

Data t2

Data t3

Data t4

Restore

Data t1

Data t2

Data t3

Data t4
State Primitives: Partition

Processing state modeled as (key, value) dictionary

**State partitioned** according to **key** $k$ of tuples
- Same key used to partition streams

![Diagram showing state partitioning based on userId]
Outline of the Talk

Stream Processing Motivation

Challenges:
Stateful Stream Processing in Cloud Environments

SEEP: Explicit State in Stateful Stream Processing
  – State management primitives
  – **Integrated mechanism for scale out and failure recovery**
  – SEEP Stream Processing Platform

Evaluation Results

Conclusions
Two cases:
- Operator B becomes bottleneck $\Rightarrow$ Scale out
- Operator B fails $\Rightarrow$ Recover
Scaling Out Stateful Operators

Periodically, stateful operators checkpoint and back up state to designated upstream backup node. For scale out, backup node already has state of operator to be parallelised.

Finally, upstream operators replay unprocessed tuples to update checkpointed state.
Recovering Failed Operators

Use backed up state to recover quickly

State restored and unprocessed tuples replayed from buffer
Experimental stateful stream processing platform

Implements dynamic scale out and recovery
- Detect failed or overloaded operators
- Have fast access to new VMs
Detecting Bottlenecks

Bottleneck detector

CPU utilisation report

35%

85%

30%

Local infrastructure view

35% 85% 30%

Bottleneck
**Problem:** Allocating new VMs takes minutes...

- **Monitoring information**
- **Bottleneck detected**
- **Decision to scale-out**
- **Select pre-provisioned VM (order of secs)**
- **Provision VM from cloud (order of mins)**

**Virtual Machine Pool**

- **VM1**
- **VM2**
- **VM3**

- **Add new VM to pool**
- **(dynamic pool size)**

- **Cloud provider**
Outline of the Talk

Stream Processing Motivation

Challenges:
Stateful Stream Processing in Cloud Environments

SEEP: Explicit State in Stateful Stream Processing
  – State management primitives
  – Integrated mechanism for scale out and failure recovery
  – SEEP Stream Processing Platform

Evaluation Results

Conclusions
Evaluation: Goals and Methodology

1. Effectiveness of dynamic scale out
2. Measurement of failure recovery time
3. Overhead of state management

Workload: **Linear Road Benchmark** [VLDB’04]
- Operator state depends on whole stream history
- Input stream rate increases over time according to Load Factor L
- SLA: results < 5 secs
- Data flow graph with 7 operators

Deployed SEEP on **Amazon AWS EC2**
Scale Out with Elastic Workload

Scales to load factor $L=350$ with 60 VMs on Amazon EC2
- $L=512$ highest report result [VLDB’12]

→ SEEP scales out dynamically with low impact on latency
Upstream Backup saves all tuples in buffers

Source Replay saves tuples only in the source
Failure Recovery Time

Workload: Windowed word counting query
- 30 sec window with 5 sec checkpointing interval

Checkpointing leads to smaller buffers
Overhead of Checkpointing

Tradeoff between latency and recovery time
Support for an imperative programming model
  – Fine-grained mutable state in dataflows

Handling state larger than single node memory
  – State that cannot be partitioned?

Modeling different data-parallel processing paradigms (eg Hadoop, Spark, Naiad) as stateful stream processing
  – Eg graph processing, incremental processing, iterative computation, …

Conclusions

Stream processing grows in importance for Big Data applications
– Data-parallel processing needed for many domains
– Almost all interesting processing algorithms are stateful

New challenges for cloud-based stream processing systems
– Bursty workloads benefit from on-demand cloud resources
– Fault tolerance with low resource overhead

Stream processing systems need operator state management
– Synergy between scale out and fault-tolerance mechanisms
– Relies on state partitioning and state checkpointing

Thank You! Any Questions?
Peter Pietzuch
<prp@doc.ic.ac.uk>
http://lsds.doc.ic.ac.uk