Drinking From The Fire Hose: Scalable Stream Processing Systems

Peter Pietzuch
prp@doc.ic.ac.uk

Large-Scale Distributed Systems Group
http://lsds.doc.ic.ac.uk

Cambridge MPhil – November 2015
The Data Deluge

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
- Challenge is not only storage but processing
Sensing and IoT

Instrumenting country’s transportation infrastructure

Many parties interested in data
- Road authorities, traffic planners, emergency services, commuters
- But access not everything: Privacy

High-level queries
- “What is the best time/route for my commute through central London between 7-8am?”
Problem:
Want to provide up-to-date predictions regarding which ads to serve

Solution:
Bayesian online learning algorithm ranks adverts according to probability of “click”
Social Data 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
Astronomic Data Processing

- Analysing transient cosmic events: $\gamma$-ray bursts
- Large Synoptic Survey Telescope (LSST)
  - Generates 1.28 Petabytes per year

Analysing transient cosmic events: $\gamma$-ray bursts
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
Traditional Databases (Boring)

Database Management System (DBMS):
- Data relatively static but queries dynamic

- Persistent relations
  - Random access
  - Low update rate
  - Unbounded disk storage

- One-time queries
  - Finite query result
  - Queries exploit (static) indices
Data Stream Processing System

DSPS: Queries static but data dynamic
- Data represented as time-depandan data stream

- Transient streams
  - Sequential access
  - Potentially high rate
  - Bounded main memory

- Continuous queries
  - Produce time-depandan result stream
  - Indexing?
Overview

Why Stream Processing?

Stream Processing Models
- Streams, windows, operators

Scalable Stream Processing Systems
- Distributed stream processing
- Stream processing with distributed dataflows

Scalable Stateful Stream Processing
- Managing state in stream processing
- Elasticity and fault tolerance mechanisms
Stream Processing

Need to define

1. Data model for streams

2. Processing (query) model for streams
“A **data stream** is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) **sequence of items**. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety.”

[Golab & Oszu (SIGMOD 2003)]

Relational model for stream structure?
- Can’t represent audio/video data
- Can’t represent analogue measurements
Relational Data Stream Model

**Streams** consist of infinite sequence of tuples
- Tuples often have associated time stamp
  - e.g. arrival time, time of reading, ...

**Tuples** have fixed relational schema
- Set of attributes

```
id = 27182  
temp = 24 C  
rain = 20mm
```

Sensors(id, temp, rain)

t
t
t
t

Sensors data stream

<table>
<thead>
<tr>
<th>id</th>
<th>temp</th>
<th>rain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**sensor output**

time
Stream Relational Model

Window converts stream to dynamic relation
- Similar to maintaining view
- Use regular relational algebra operators on tuples
- Can combine streams and relations in single query

Stream

- Window converts stream to dynamic relation
  - Similar to maintaining view
  - Use regular relational algebra operators on tuples
  - Can combine streams and relations in single query

Relation

Special operators: Istream, Dstream, Rstream

Any relational query
Sliding Window I

How many tuples should we process each time?

Process tuples in window-sized batches

**Time-based window** with size $\tau$ at current time $t$

$[t - \tau : t]$ Sensors [Range $\tau$ seconds]

$[t : t]$ Sensors [Now]

**Count-based window** with size $n$:

*last n tuples* Sensors [Rows $n$]

```
<table>
<thead>
<tr>
<th>temp</th>
<th>temp</th>
<th>temp</th>
<th>temp</th>
<th>temp</th>
<th>temp</th>
<th>temp</th>
<th>temp</th>
<th>temp</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>rain</td>
<td>rain</td>
<td>rain</td>
<td>rain</td>
<td>rain</td>
<td>rain</td>
<td>rain</td>
<td>rain</td>
<td>rain</td>
<td>rain</td>
</tr>
</tbody>
</table>
```
Sliding Window II

How often should we evaluate the window?

1. Output new result tuples as soon as available
   - Difficult to implement efficiently

2. Slide window by \( s \) seconds (or \( m \) tuples)

Sensors [Slide \( s \) seconds]

**Sliding window:** \( S < T \)

**Tumbling window:** \( S = T \)
Continuous Query Language (CQL)

Based on SQL with streaming constructs
- Tuple- and time-based windows
- Sampling primitives

```
SELECT temp
FROM Sensors [Range 1 hour]
WHERE temp > 42;
```

```
SELECT *
FROM S1 [Rows 1000],
    S2 [Range 2 mins]
WHERE S1.A = S2.A
    AND S1.A > 42;
```

Apart from that regular SQL syntax
Naturally supports joins over windows

```
SELECT *
FROM S1, S2
WHERE S1.a = S2.b;
```

Only meaningful with window specification for streams
- Otherwise requires unbounded state!

```
SELECT S.id, S.rain
FROM Sensors [Rows 10] as S, Faulty [Range 1 day] as F
WHERE S.rain > 10 AND F.id != S.id;
```
Define mapping from relation back to stream
  – Assumes discrete, monotonically increasing timestamps $\tau, \tau+1, \tau+2, \tau+3, \ldots$

$I_{stream}(R)$
  – Stream of all tuples $(r, \tau)$ where $r \in R$ at time $\tau$ but $r \notin R$ at time $\tau-1$

$D_{stream}(R)$
  – Stream of all tuples $(r, \tau)$ where $r \in R$ at time $\tau-1$ but $r \notin R$ at time $\tau$

$R_{stream}(R)$
  – Stream of all tuples $(r, \tau)$ where $r \in R$ at time $\tau$
Stream Processing Systems
Continuous queries are long-running

➤ properties of base streams may change

  – Tuple distribution, arrival characteristics, query load, available CPU, memory and disk resources, system conditions, ...

Solution: Use **adaptive query plans**

  – Monitor system conditions

  – Re-optimise query plans at run-time

DBMS didn’t quite have this problem...
Query Plan Execution

Executed query plans include:
- **Operators**
- **Queues** between operators
- **State**/“Synposis” (windows, ...)
- **Base streams**

```
SELECT *
FROM S1 [Rows 1000],
  S2 [Range 2 mins]
WHERE S1.A = S2.A
AND S1.A > 42;
```

**Challenges**
- State may get large (e.g. large windows)
Operator Scheduling

Need scheduler to invoke operators (for time slice)
  – Scheduling must be adaptive

Different scheduling disciplines possible:
  1. Round-robin
  2. Minimise queue length
  3. Minimise tuple delay
  4. Combination of the above
Load Shedding

DSMS must handle overload:
Tuples arrive faster than processing rate

Two options when overloaded:

1. **Load shedding**: Drop tuples
   - Much research on deciding which tuples to drop: c.f. result correctness and resource relief
   - e.g. sample tuples from stream

2. **Approximate processing**: Replace operators with approximate processing
   - Saves resources
Scalable Stream Processing
Big Data Centres + Big Data

Google: 20 data centre locations
- over 1 million servers
- 260 Megawatts (0.01% of global energy)
- 4.2 billion searches per day (2011)
- Exabytes (10^{18}) of storage

Assumptions:
- **Scale out** and not scale up
  - Commodity servers with local disks
  - Data-parallelism is king
- Software designed for **failure**

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

How do you parallelise stream processing across VMs?
MapReduce: Distributed Dataflow

Data model: (key, value) pairs

Two processing functions:
- \( \text{map}(k_1, v_1) \rightarrow \text{list}(k_2, v_2) \)
- \( \text{reduce}(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_3) \)

Benefits:
- Simple programming model
- Transparent parallelisation
- Fault-tolerant processing

$2$ billion market revenue (2013)
MapReduce Execution Model

Map/reduce tasks scheduled across cluster nodes

Intermediate results persisted to local disks
- Restart failed tasks on another node
- Distributed file systems contains replicated data

But this is a batch processing model...
Design Space for Big Data Systems

Volume and Velocity

Algorithmic complexity
- Arbitrary data transformation
- Iterative algorithms
- Large state as part of computation
**Spark: Micro-Batching**

**Idea:**
Reduce size of data partitions to produce up-to-date, incremental results

**Micro-batching for data**
- Window-based task semantics
- Parallel recomputation of RDDs

**Challenge:**
Need to control scheduling overhead
SEEP: Pipelined Dataflows

Idea:
Materialise dataflow graph to avoid scheduling overhead

Challenges:
1. Support for iteration
2. Resource allocation of tasks to nodes
3. Failure recovery

Cycles in graph for iteration
Dynamic scale out of tasks
Checkpoint-based recovery
SEEP: Low Latency Processing

Dataflow graph for window-based word count
- Deployed on 4 nodes (4-core 3.4 Ghz Intel Xeon with 8GB RAM)
Scalable Stateful Stream Processing
What about Processing State?

Online collaborative filtering:

Customer activity on website

User A
Item: “iPad”
Rating: 5

User-item matrix

<table>
<thead>
<tr>
<th></th>
<th>Item 1</th>
<th>Item 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>User A</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>User B</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

GBs to TBs in size
Matrix userItem = new Matrix();
Matrix coOcc = new Matrix();

void addRating(int user, int item, int rating) {
    userItem.setElement(user, item, rating);
    updateCoOccurrence(coOcc, userItem);
}

Vector getRec(int user) {
    Vector userRow = userItem.getRow(user);
    Vector userRec = coOcc.multiply(userRow);
    return userRec;
}
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
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: “Peter”
Item: “iPad”
Rating: 3

User: “BB”
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</td>
<td>4</td>
</tr>
</tbody>
</table>

**Routing state**

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

**Buffer state**

<table>
<thead>
<tr>
<th>Data</th>
<th>Data</th>
<th>Data</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ts1</td>
<td>ts2</td>
<td>ts3</td>
<td>ts4</td>
</tr>
</tbody>
</table>

User A: Item 2
User B: Item 1
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**

- **Restore**

- **A**
- A1
- A2

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

A

Data

t4

Data

t3

Restore

Data

t2

Data

t1

B

Backup

Checkpoint

t2

Data

t2

Data

t1

Data

t2

Data

t1
State Primitives: Partition

Processing state modeled as (key, value) dictionary

**State partitioned** according to **key** $k$ of tuples

- Same key used to partition streams
Two cases:
- Operator B **fails** ➔ **Recover**
- Operator B becomes **bottleneck** ➔ **Scale out**
Recovering Failed Operators

Periodically, stateful operators checkpoint and back up state to designated upstream backup node.

Use backed up state to recover quickly.

State restored and unprocessed tuples replayed from buffer.
For scale out, backup node already has state of operator to be parallelised

Finally, upstream operators replay unprocessed tuples to update checkpointed state
Experimental stateful stream processing platform

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

CPU utilisation report

Bottleneck detector

85%

35%

30%

Local infrastructure view

Bottleneck

35% 85% 30%
**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
- Provision VM from cloud (order of mins)
Evaluation
Linear Road Benchmark [VLDB’04]
- Network of toll roads of size $L$
- Input rate increases over time
- Dataflow graph with 5 operators; SLA: results < 5 secs

SEEP deployed on Amazon EC2
- Scales to 60 VMs (small instances with 2GB RAM)

Achieves $L=350$
- $L=512$ highest reported result in literature [VLDB’12]
Performance of SEEP

Logistic regression

- Deployed on Amazon EC2 ("m1.xlarge" VMs with 4 vCPUs and 16 GB RAM)
- 100 GB dataset
Overhead of Checkpointing

![Graph showing the tradeoff between latency and recovery time.](image)

- **Tradeoff between latency and recovery time**
Related Work

Scalable stream processing systems

- **Twitter Storm, Yahoo S4, Nokia Dempsey, Apache Samza**
  Exploit operator parallelism mainly for stateless queries

Distributed dataflow systems

- **MapReduce, Dryad, Spark, Apache Flink, Naiad, SEEP**
  Shared nothing data-parallel processing on clusters

Elasticity in stream processing

- **StreamCloud** [TPDS’12]
  Dynamic scale out/in for subset of relational stream operators
- **Esc** [ICCC’11]
  Dynamic support for stateless scale out

Resource-efficient fault tolerance models

- **Active Replication at (almost) no cost** [SRDS’11]
  Use under-utilized machines to run operator replicas
- **Discretized Streams** [HotCloud’12]
  Data is checkpointed and recovered in parallel in event of failure
Summary

**Stream processing** grows in importance
- Handling the data deluge
- Enables real-time response and decision making

**Principled models** to express stream processing semantics
- Window-based declarative query languages
- What is the right programming model for machine learning?

**Stateful distributed dataflows** for stream processing
- High stream rates require data-parallel processing
- Fault-tolerant support for state important for many algorithms
- Convergence of batch and stream processing
Thank You! Any Questions?

Peter Pietzuch
<prp@doc.ic.ac.uk>
http://lsds.doc.ic.ac.uk