Drinking From The Fire Hose:
The Rise of Scalable Stream Processing Systems

Peter Pietzuch
prp@doc.ic.ac.uk

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

Cambridge MPhil – February 2013
The Data Deluge

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

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 also querying
Real Time Traffic Monitoring

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?”
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
Analysing transient cosmic events: γ-ray bursts

Large Synoptic Survey Telescope (LSST)
- Generates 1.28 Petabytes per year
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
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-dependent data stream

- Working storage
  - Transient streams
    - Sequential access
    - Potentially high rate
  - Continuous queries
    - Produce time-dependent result stream
    - Indexing?

Result stream
Overview

Why Stream Processing?

Stream Processing Models
- Streams, windows, operators
- Data mining of streams

Stream Processing Systems
- Distributed Stream Processing
- Scalable Stream Processing in the Cloud
Stream Processing

Need to define

1. Data model for streams

2. Processing (query) model for streams
“**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

```
Sensors(id, temp, rain)
```

sensor output

```
<table>
<thead>
<tr>
<th>id</th>
<th>temp</th>
<th>rain</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>temp1</td>
<td>rain1</td>
</tr>
<tr>
<td>id2</td>
<td>temp2</td>
<td>rain2</td>
</tr>
<tr>
<td>id3</td>
<td>temp3</td>
<td>rain3</td>
</tr>
</tbody>
</table>
```

Sensors data stream over time
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
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 $n$]

```
| temp | rain | temp | rain | temp | rain | temp | rain | temp | rain | temp | rain |
```

window
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]

<table>
<thead>
<tr>
<th>Sliding window:</th>
<th>S &lt; T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tumbling window:</td>
<td>S = T</td>
</tr>
</tbody>
</table>

![Diagram of sliding window]
Continuous Query Language (CQL)

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

Apart from that regular SQL syntax

```
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;
```
Join Processing

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;
```
Converting Relations \(\rightarrow\) Streams

Define mapping from relation back to stream
- Assumes discrete, monotonically increasing timestamps \(\tau, \tau+1, \tau+2, \tau+3, \ldots\)

\text{Istream}(R)
- Stream of all tuples \((r, \tau)\) where \(r \in R\) at time \(\tau\) but \(r \notin R\) at time \(\tau-1\)

\text{Dstream}(R)
- Stream of all tuples \((r, \tau)\) where \(r \in R\) at time \(\tau-1\) but \(r \notin R\) at time \(\tau\)

\text{Rstream}(R)
- Stream of all tuples \((r, \tau)\) where \(r \in R\) at time \(\tau\)
Data Mining in Streams
Often continuous queries relate to long-term characteristics of streams
- Frequency of stock trades, number of invalid sensor readings, ...

May have insufficient memory to evaluate query
- Consider stream with window of $10^9$ integers
  - Can store this in 4GB of memory
- What about $10^6$ such streams?
  - Cannot keep all windows in memory

Need to compress data in windows
Limitations of Window Compression

Consider window compression for following query:

```
SELECT SUM(num)
FROM Numbers [Rows 10^9];
```

Assume that \( W \) can be compressed as \( C(W) = W_C \)
- Then \( W_1 \neq W_2 \) must exist, with \( C(W_1) = C(W_2) \)
- Let \( t \) be oldest time in window for which \( W_1 \) and \( W_2 \) differ:

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>5</th>
<th>8</th>
<th>9</th>
<th>2</th>
<th>3</th>
<th>9</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W_1 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( W_2 )</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- For \( W_1 \): subtract \( W_1(t) = 3 \); for \( W_2 \): subtract \( W_2(t) = 4 \)
  - Cannot distinguish between cases from \( C(W_1) = C(W_2) \)
- No correct compression scheme \( C(W) \) possible
Approximate Sum Calculation

Keep sums $\Sigma_i$ for each $n$ tuples in window
- Compression ratio is $1/n$

Now $v_1$ leaves window and $v_{2n+3}$ arrives:
- Accuracy of approximation depends on variance

$\Sigma_W = (n-1/n) * \Sigma_1 + \Sigma_2 + \cdots + \Sigma_{\text{incomplete}}$

$\Sigma_W = \Sigma_1 + \Sigma_2 + \cdots + \Sigma_{\text{incomplete}}$

Estimate of window sum $\Sigma_W$ is total of group sums $\Sigma_i$
Counting Bits

Assume sliding window $W$ of size $N$ contains bits 1 and 0

- How many 1s are there in the most recent $k$ bits? ($1 \leq k \leq N$)

Could answer question trivially with $O(N)$ storage

- But can we approximate answer with, say, logarithmic storage?
Approximate Counting with Buckets

Divide window into multiple buckets $B(m, t)$
- $B(m, t)$ contains $2^m$ 1s and starts at $t$
- Size of buckets does not decrease as $t$ increases
- Either one or two buckets for each size $m$
- Largest bucket only partially filled

\[ \Sigma_k = \{\text{sizes of buckets within } k\} + \frac{1}{2} \{\text{last partial bucket}\} \]

\[ \Sigma_N = 2^0 + 2^0 + 2^1 + 2^2 + \frac{1}{2} * 2^3 = 12 \text{ (exact answer: 13)} \]
Maintaining Buckets

Discard/merge buckets as window slides

1. Discard largest bucket once outside of window
2. Create new bucket $B(0,1)$ for new tuple if 1
3. Merge buckets to restore invariant of at most 2 buckets of each size $m$
Space Complexity

Need $O(\log N)$ buckets for window of size $N$

Need $O(\log N)$ bits to represent bucket $B(m, t)$:
- $m$ is power of 2, so representable as $\log_2 m$
  $m$ can be represented with $O(\log \log N)$ bits
- $t$ is representable as $t \mod N$
  $t$ can be represented with $O(\log N)$ bits

Overall window compressed to $O(\log^2 N)$ bits
Stream Processing Systems
General DSPS Architecture

Source: Golab & Ozsu 2003
Stream Query Execution

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**/“Synopsis” (windows, ...)
- **Base streams**

```sql
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
Distributed DSPS
Distributed DSPS

Interconnect multiple DSPSs with network
  – Better scalability, handles geographically distributed stream sources

Interconnect on LAN or Internet?
  – Different assumptions about time and failure models
Stream Processing to the Rescue!

- Process data streams on-the-fly:
  Apache S4, Twitter Storm, Nokia Dempsy, ...

- Most interesting operators are **stateful**

- Exploit intra-query parallelism for scale out
Query Planning in DAPS

Query Plan
- Operator placement
- Stream connections
- Resource allocation: CPU, network bandwidth, ...

State-of-the-art planners
- Based on heuristics (e.g., IBM’s SODA)
- Assume over-provisioned system
  - Simplifies query planning
  - Not true when you pay for resources...
Planning Challenges

- Premature exhaustion of resources
- Multi-resource constraints

Waste of resources due to query overlap ➔ reuse streams

Premature exhaustion of resources ➔ multi-resource constraints
Unified optimisation problem for

- query admission
- operator allocation
- stream reuse

maximise:

\[ \lambda_1 \times (\text{no of satisfied queries}) - \lambda_2 \times (\text{CPU usage}) - \lambda_3 \times (\text{net usage}) - \lambda_4 \times (\text{balance load}) \]

subject to constraints:
1. availability: streams for operators exist on nodes
2. resource: allocations within resource limits
3. demand: final query streams are generated eventually
4. acyclicity: all streams come from real sources

This is hard!

- Solve approximate problem to obtain tractable solution
Tractable Optimisation Model

Idea: Only optimise over streams related to new query

- Add relay operators to work around constraints under reuse
Scalable Stream Processing
Stream Processing in the Cloud

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

In a utility-based pricing model:

How do you use the optimal number of resources?
– Needlessly overprovisioning system is expense
– Using too few resources leads to poor performance
Challenges in Cloud-Based Stream Processing

**Intra-query parallelism**
- Provisioning for workload peaks unnecessarily conservative

**Failure resilience**
- Active fault-tolerance requires 2x resources
- Passive fault-tolerance leads to long recovery times

*Dynamic scale out:* increase resources when peaks appear

*Hybrid fault-tolerance:* low resource overhead with fast recovery

*Stateful operators:* both mechanisms must support stateful operators
Operator State Management in stream processing

Two state-aware mechanisms:
1. Dynamic Operator Scale Out
2. Upstream Backup with Checkpointing (UBC)

Evaluation results

Operator State Management

State cannot be lost, or stream results are affected

On **scale out:**
- Partition operator state correctly, maintaining consistency

On **failure recovery:**
- Restore state of failed operator

👉 Make operator state an external entity that can be managed by the stream processing system
- Define primitives for state management and build other mechanisms on top of them
State Management

What is state in stream processing system?

- Need to externalise processing state of operators
State Management Primitives

- **Checkpoint**: Takes snapshot of state and makes it externally available.
- **Backup**: Moves copy of state from one operator to another.
- **Restore**: Splits state in a semantically correct fashion for parallel processing.
State Partitioning

Processing state modeled as (key, value) dictionary

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

Tuples will be routed to correct operator
- \( x \) is splitting key that partitions state

\[
\begin{align*}
\text{stream } 0-n & \quad \text{stream } 0-x \\
\text{stream } x-n & \quad \text{stream } 0-x
\end{align*}
\]

\[
\begin{align*}
K1-V1 & \quad K2-V2 \\
K3-V3 & \quad \ldots \\
Kn-Vn & \quad A \\
& \quad B
\end{align*}
\]
1. **Dynamic Scale Out**: Detect bottleneck, remove by adding new parallelised operator

2. **Failure Recovery**: Detect failure, replace with new operator
Dynamic Scale Out: Detecting bottlenecks

Bottleneck detector

CPU utilisation report

Local infrastructure view

35% 85% 30%

Bottleneck
The VM Pool: Adding operators

**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**

**Cloud provider**

**(dynamic pool size)**
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.
Recreate operator state by replaying tuples after failure
  – Send acknowledgements upstream for tuples processed downstream

May result in long recovery times due to large buffers
  – System is reprocessing streams after failure ➔ inefficient
Upstream Backup + Checkpointing

Benefit from state management primitives
- Use periodically backed up state on upstream node to recover faster

State is restored and unprocessed tuples are replayed from buffer
Experimental Evaluation

Goals
- Investigate effectiveness of scale out mechanism
- Recovery time after failure using UBC
- Overhead of state management

Prototype system: Scalable and Elastic Event Processing (SEEP)
- Implemented in Java; Storm-like data flow model

Sample queries + workload
- Linear Road Benchmark (LRB) to evaluate scale out [VLDB’04]
  - Provides an increasing stream workload over time for given load factor
  - Query with 8 operators; SLA: results < 5 secs
- Windowed word count query to evaluate fault tolerance
  - Induce failure to observe performance impact

Deployment on Amazon AWS EC2
- Sources and sinks on high-memory double extra large instances
- Operators on small instances
Scale Out: LRB Workload

Scales to load factor $L=350$ with 60 VMs on Amazon EC2
- Automated query parallelisation

$L=512$ highest report result [VLDB’12]
- Hand-crafted query on dedicated cluster

Scale out leads to latency peaks, but remains within LRB SLA
UB+C: Recovery Time

Source Replay:
Upstream Backup with tuples replayed by source only

State backed up every 5 seconds in UB+C

UB+C achieves faster recovery, especially for fast stream rates
Tradeoff of Checkpointing Interval

Shorter checkpointing interval leads to faster recovery times. But also incurs more overhead, impacting tuple processing latency.
Related Work

Scalable stream processing systems
- **Twitter Storm, Yahoo S4, Nokia Dempsey**
  Exploit operator parallelism mainly for stateless queries
- **ParaSplit operator** [VLDB’12]
  Partition stream for intra-query parallelism

Support for elasticity
- **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
Conclusions

Stream processing will grow in importance
- Handling the data deluge
- Just provide a view/window on subset of data
- Enables real-time response and decision making

Principled models to express stream processing semantics
- Enables automatic optimisation of queries, e.g. finding parallelism
- What is the right model?

Resource allocation matters due to long running queries
- High stream rates and many queries require scalable systems
- Handling overload becomes crucial requirement
- Volatile workloads benefit from elastic DSPS in cloud environments
Thank You! Any Questions?

Peter Pietzuch
<prp@doc.ic.ac.uk>
http://lsds.doc.ic.ac.uk
Backup
Global Sensor Applications: EarthScope

Using sensors to understand geological evolution
- Many sources: 400 seismometers, 1000 GPS stations, ...

How do you process all this data?
Stream Processing in the Cloud

Stream

... n servers in cloud DC

Results

Scalability: Scale horizontally across 1000 VMs to support
  - larger number of queries
  - high stream rates

Elasticity: Dynamically tune number of processing servers
  - Tune n to affect stream processing throughput
Load Balancing with the Cloud

Idea: Using cloud resources for handling peak processing demand

- Network latency to cloud major issue
- Partitioning granularity important

How do you perform stream processing in the cloud?
Existing workloads have peaks and troughs
  - Scope for improvement in terms of **elasticity and adaptability**

Current solutions in distributed stream processing
  - **Over-provisioning** to handle peak demand
  - **Load-shedding** to discard data during peaks
The Map/Reduce Hammer?

Strawman idea:
- Adapt batch processing model
- Pipelined implementation of map/reduce

Partitioning granularity?
- Window = job?
- Apache Hadoop has large per job overhead

Stream processing semantics?

Data exchange based on distributed file system
Application Domains for Stream Processing

Processing sensor data
- Readings of physical quantity from sensors
- Readings of RFID tags

Scientific experiments
- Result streams from particle accelerators
- Photon sightings from radio telescopes

Financial transactions
- Detection of credit card fraud
- Debit card transactions from shops
- Trades from stock markets

Network monitoring
- Packet monitoring in intrusion detection systems
Detecting Transient Sky Objects

Detection requires non-trivial processing
- Needs to happen within minutes
- Can’t express it in SQL

Where do we do the computation?
- What data do we store?

Often looking for needle in haystack
Database Triggers

Database triggers are stored queries
- Triggered by stream of updates

```
CREATE TRIGGER PrizeStudent
AFTER UPDATE OF mark ON Exam
FOR EACH ROW
WHEN (mark > 80)
BEGIN
    INSERT Prizes(name, mark)
    VALUES (...)
END
```

Often written as event-condition-action rules
- Action can be any stored procedure

Hard to support efficiently
- Difficult to take advantage of overlap between triggers
- Low performance with high update rates
Sliding Windows

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 $n$]
Memory Overhead

Queues & State kept in memory
- Keep in memory for fast access
- Large state swapped out to disk?

Goal: Minimise memory usage
1. Detect and exploit constraints on streams to reduce state
2. Share state within and between queries
3. Schedule operators intelligently to keep queues short
Exploiting Stream Constraints

Exploit query semantics to bound windows

- Provide additional information about streams:
  - Stream semantics
  - Ordering
  - Referential integrity

Assume all sensors checked once a day:

```
SELECT S.id, S.rain
FROM Sensors [Rows 10] as S, Faulty as F
WHERE S.rain > 10 AND F.id != S.id;
```

[Range 1 day]
Sharing State + Processing

**Base streams:** Shared by all queries
- Maintain single maximum window

**Intermediate streams:** Shared by some queries
- Share state and processing
- Reduce memory consumption of sliding window aggregates
Open Questions

Where will be the bottleneck in the system?
- Can we partition/filter the stream fast enough?

Are EPMs expressive enough to be useful?
- Other computational models possible

How can we adapt to workload changes?
- Migration of EPMs?

Currently building a prototype system to play around with...
Space Complexity

Need $O(\log N)$ buckets for window of size $N$

Need $O(\log N)$ bits to represent bucket $B(m, t)$:
- $m$ is power of 2, so representable as $\log_2 m$
  $m$ can be represented with $O(\log \log N)$ bits
- $t$ is representable as $t \mod N$
  $t$ can be represented with $O(\log N)$ bits

Overall window compressed to $O(\log^2 N)$ bits

Estimation error at most 50%:
- Assume partial bucket has size $m$
  Average contribution of partial bucket: $\frac{1}{2} m$
- Sum of smaller buckets: $m/2 + m/4 + \ldots = m$
  Worst case: estimate too low by half
- Reduce error: keep between $p$ and $p+1$ buckets of each size
This Talk

**Efficiency**

How can a stream processing system allocate resources efficiently?

**SQPR: Stream Query Planning with Reuse**
- Initial allocation of processing operators to machines in a cluster
- Treat query planning as an optimisation problem

**Scalability**

How can a stream processing system scale to arbitrary workloads?

**SEEP: Scalable and Elastic Stream Processing**
- Elastic architecture for stream processing in the cloud
- Two phase architecture: filtering and transformation
1: wait until new query $q$ arrives
2: if $q$ is already satisfied then
3: reuse stream
4: else
5: add demand constraint for $q$
6: fix optimisation variables relating to unrelated streams
7: solve optimisation model (MILP problem) using standard branch & bound techniques
8: update solution
9: notify hosts of changed streams and operators
Evaluation Results

Custom simulator
- Workload based on multi-way join queries
- CPU and network constrained environments

Prototype deployment with DISSP platform
- 15 nodes with 10Mbps network bandwidth
- Comparison with IBM’s SODA scheduler
Planning Efficiency

SQPR manages to place more queries than heuristics/SODA
Publish/Subscribe Layer

Incoming streams broadcast to P/S layer VMs
- Match predicates \((P_1, P_2, ..., P_n)\) on incoming streams
- Matched tuples dispatched to VMs in partitioning layer

Inverted index created over predicates to speed up matching
- Predicates composed from language for efficient indexing
- Indexed according to matched attributes, operators and values
- Rich literature on efficient matching

Stream augmentation with stored data
Partitioning Layer

Event Processing Machines (EPMs) transform streams
- Implemented as non-deterministic FSAs
- Composed of detection/aggregation states
  - Each EPM instance contains state S derived by tuples processed so far
  - States linked by edge predicates (computed in P/S layer)

When matched tuples dispatched to EPM:
1. Makes transition to new state
   - Transition might generate new EPM instances (non-determinism)
2. Aggregation function incorporates new tuple in S
3. On accepting state, state S becomes part of result stream
EPM Decomposition

Decompose EPM into fragments hosted on different VMs
  - Pipelines EPM execution

Support EPMs with large state requirements

Execute state transitions in parallel

query

compile

decompose

synchronisation

SELECT temp
FROM Sensors
WHERE temp > 42;

84
Resource Allocation

Allocate EPM fragments to VMs in partitioning layer
- Must balance CPU load across all VMs
- Observe network bandwidth constraints
SEEP Architecture

Partitioning layer

Pub/sub layer

Event broadcast

Event Streams

Elasticity

Applications 1, 2, 3
Scratch
Two Layers: Dispatching and Processing

Structured architecture for stream processing
- Separates stream partitioning from computation
- Partitioning reduces amount of data for computation

Simple function in each operators:

1. Stream partitioning performed by dispatching layer
   - Identify relevant data for queries
   - Partitioning of data streams and multicast to multiple operators

2. Computation done by processing layer
   - Execution of query operators
SEEP: Scalable & Elastic Event Processing

Decompose queries into multiple stream processing operators
- System exploits intra-query parallelism

Adapt to variations in workload by scaling out
SEEP: Scalable & Elastic Event Processing

Partition and merge streams to utilise more hosts
Twitter Storm & Yahoo S4

**Yahoo! S4** (http://incubator.apache.org/s4/)
- Java framework for implementing stream processing applications
- Hides stream “plumbing” from developers
- Uses *Zookeeper* for coordination

**Twitter Storm** (https://github.com/nathanmarz/storm)
- Focus on **fault-tolerance**: acknowledgement of processed tuples
- *Spouts* produce data; *bolts* process data
- Different mechanisms for stream partitioning and bolt parallelisation

This is just the beginning... lots of open challenges...