Big Data Stream Processing

Tilmann Rabl
Berlin Big Data Center

www.dima.tu-berlin.de | bbdc.berlin | rabl@tu-berlin.de
Agenda

Introduction to Streams
• Use cases
• Stream Processing 101

Stream Processing Systems
• Ingredients of a stream processing system
• Some examples
• More details on Storm, Spark, Flink
• Maybe a demo (!)

Stream Processing Optimizations (if we have time)
• How to optimize

With slides from Data Artisans, Volker Markl, Asterios Katsifodimos, Jonas Traub
Big Fast Data

• Data is growing and can be evaluated
  – Tweets, social networks (statuses, check-ins, shared content), blogs, click streams, various logs, …
  – Facebook: > 845M active users, > 8B messages/day
  – Twitter: > 140M active users, > 340M tweets/day

• Everyone is interested!
But there is so much more…

- **Autonomous Driving**
  - Requires rich navigation info
  - Rich data sensor readings
  - 1GB data per minute per car (all sensors)\(^1\)

- **Traffic Monitoring**
  - High event rates: millions events / sec
  - High query rates: thousands queries / sec
  - Queries: filtering, notifications, analytical

- **Pre-processing of sensor data**
  - CERN experiments generate ~1PB of measurements per second.
  - Unfeasible to store or process directly, fast preprocessing is a must.

\(^1\)Cobb: [http://www.hybridcars.com/tech-experts-put-the-brakes-on-autonomous-cars/]
Why is this hard?

Tension between performance and algorithmic expressiveness
Stream Processing 101

With some Flink Examples
Based on the Data Flow Model
What is a Stream?

• Unbounded data
  – Conceptually infinite, ever growing set of *data items / events*
  – Practically continuous stream of data, which needs to be processed / analyzed

• Push model
  – Data production and procession is controlled by the source
  – Publish / subscribe model

• Concept of time
  – Often need to reason about *when* data is produced and when processed data should be output
  – Time agnostic, processing time, ingestion time, event time

This part is largely based on Tyler Akidau’s great blog on streaming - https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101
Stream Models

\[ S = s_i, s_{i+1}, \ldots \quad s_i = \langle \text{data item}, \text{timestamp} \rangle \]

- **Turnstile**
  - Elements can come and go
  - Underlying model is a vector of elements (domain)
  - \( s_i \) is an update (increment or decrement) to a vector element
  - Traditional database model
  - Flexible model for algorithms

- **Cash register**
  - Similar to turnstile, but elements cannot leave

- **Time series**
  - \( s_i \) is a new vector entry
  - Vector is increasing
  - This is what all big stream processing engines use
Event Time

- Event time
  - Data item production time
- Ingestion time
  - System time when data item is received
- Processing time
  - System time when data item is processed

- Typically, these do not match!
- In practice, streams are unordered!
Time Agnostic Processing

- Filtering
  - Stateless
  - Can be done per data item
  - Implementations: hash table or bloom filter

Image: Tyler Akidau
Time Agnostic Processing II

- Inner join
  - Only current elements
  - Stateful
  - E.g., hash join
- What about other joins (e.g., outer join)?

Image: Tyler Akidau
Approximate Processing

- Streaming k-means, sketches
  - Low overhead
  - Notion of time
- Not covered in this talk
Windows

- Fixed
  - Also tumbling
- Sliding
  - Also hopping
- Session
  - Based on activity
- Triggered by
  - Event time, processing time, count, watermark
- Eviction policy
  - Window width / size

Image: Tyler Akidau
Processing Time Windows

- System waits for x time units
  - System decides on stream partitioning
  - Simple, easy to implement
  - Ignores any time information in the stream -> any aggregation can be arbitrary

- Similar: Counting Windows

Image: Tyler Akidau
Event Time Windows

- Windows based on the time information in stream
  - Adheres to stream semantic
  - Correct calculations
  - Buffering required, potentially unordered (more on this later)

Images: Tyler Akidau
Basic Stream Operators

- **Windowed Aggregation**
  - E.g., average speed
  - Sum of URL accesses
  - Daily highscore

- **Windowed Join**
  - Correlated observations in timeframe
  - E.g., temperature in time
Flink’s Windowing

• Windows can be any combination of (multiple) triggers & evictions
  – Arbitrary tumbling, sliding, session, etc. windows can be constructed.

• Common triggers/evictions part of the API
  – Time (processing vs. event time), Count

• Even more flexibility: *define your own UDF* trigger/eviction

• Examples:

```java
dataStream.windowAll(TumblingEventTimeWindows.of(Time.seconds(5)));
dataStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(5)));
```
Example Analysis: Windowed Aggregation

```
val windowedStream = stockStream.window(Time.of(10, SECONDS)).every(Time.of(5, SECONDS))
val lowest = windowedStream.minBy("price")
val maxByStock = windowedStream.groupBy("symbol").maxBy("price")
val rollingMean = windowedStream.groupBy("symbol").mapWindow(mean _)
```

```
StockPrice(SPX, 2113.9)
StockPrice(FTSE, 6931.7)
StockPrice(HDP, 23.8)
StockPrice(HDP, 26.6)
```

```
(1) val windowedStream = stockStream.window(Time.of(10, SECONDS)).every(Time.of(5, SECONDS))
(2) val lowest = windowedStream.minBy("price")
(3) val maxByStock = windowedStream.groupBy("symbol").maxBy("price")
(4) val rollingMean = windowedStream.groupBy("symbol").mapWindow(mean _)
```
Complex Event Processing

- Detecting patterns in a stream
- Complex event = sequence of events
- Defined using logical and temporal conditions
  - Logical: data values and combinations
  - Temporal: within a given period of time

\[
\text{SEQ}(A, B, C) \text{ WITH} \\
A. \text{Temp} > 23^\circ \text{C} \&\& \\
B. \text{Station} = A.\text{Station} \&\& B.\text{Temp} < A.\text{Temp} \&\& \\
C. \text{Station} = A.\text{Station} \&\& A.\text{Temp} \& B.\text{Temp} > 3
\]
Complex Event Processing Contd.

- Composite events constructed e.g. by
  - SEQ, AND, OR, NEG, ...
  - SEQ(e1, e2) → (e1, t1) ∧ (e2, t2) ∧ t1 ≤ t2 ∧ e1, e2 ∈ \( \mathbb{W} \)
  - Implemented by constructing a NFA
    - Example: SEQ(A, B, C)

Slide by Kai-Uwe Sattler
Stream Processing Systems

What makes a system a stream processing system?
8 Requirements of Big Streaming

- Keep the data moving
  - Streaming architecture

- Declarative access
  - E.g. StreamSQL, CQL

- Handle imperfections
  - Late, missing, unordered items

- Predictable outcomes
  - Consistency, event time

- Integrate stored and streaming data
  - Hybrid stream and batch

- Data safety and availability
  - Fault tolerance, durable state

- Automatic partitioning and scaling
  - Distributed processing

- Instantaneous processing and response

The 8 Requirements of Real-Time Stream Processing – Stonebraker et al. 2005
8 Requirements of Big Streaming

- **Keep the data moving**
  - Streaming architecture

- **Declarative access**
  - E.g. StreamSQL, CQL

- **Handle imperfections**
  - Late, missing, unordered items

- **Predictable outcomes**
  - Consistency, event time

- **Integrate stored and streaming data**
  - Hybrid stream and batch

- **Data safety and availability**
  - Fault tolerance, durable state

- **Automatic partitioning and scaling**
  - Distributed processing

- **Instantaneous processing and response**

The 8 Requirements of Real-Time Stream Processing – Stonebraker et al. 2005
Big Data Processing

• Databases can process very large data since forever (see VLDB)
  – Why not use those?

• Big data is not (fully) structured
  – No good for database 😞

• We want to learn more from data than just
  – Select, project, join

• First solution: MapReduce
Map Reduce

- Framework / programming model by Google
  - Presented 2004 at OSDI'04
- Inspired by map and reduce functions in functional languages / MPI
  - Second order functions
- Simple parallelization model for shared nothing architectures (“commodity hardware”)
- Apache Hadoop
  - Open-source implementation
  - Initiated at Yahoo

Map: Computation
For each input create list of output values
Example:
For each word in a sentence emit a k/v pair indicating one occurrence of the word
(key, “hello world”) -> (“hello”, “1”), (“world”, “1”)

Signature
map (key, value) -> list(key’, value’)

Reduce: Aggregation
Combine all intermediate values for one key
Example:
Sum up all values for the same key
(“Hello”, (“1”, “1”, “1”, “1”)) -> (“Hello”, (“4”))

Signature
reduce (key, list(value)) -> list(value’)

Signature
reduce (key, list(value)) -> list(value’)

Signature
reduce (key, list(value)) -> list(value’)
MR Data Flow
MR / Batch Processing
MR / Batch Processing
MR / Batch Window Processing
MR Discussion

- Great for large amounts of static data
- For streams: only for large windows
- *Data is not moving!*
- *High latency, low efficiency*

Images: Tyler Akidau
How to keep data moving?

Discretized Streams (mini-batch)

while (true) {
    // get next few records
    // issue batch computation
}

Native streaming

while (true) {
    // process next record
}
Discussion of Mini-Batch

- Easy to implement
- Easy consistency and fault-tolerance
- Hard to do event time and sessions

Image: Tyler Akidau
True Streaming Architecture

- Program = DAG* of operators and intermediate streams
- Operator = computation + state
- Intermediate streams = logical stream of records

- Stream transformations
  - Basic transformations: Map, Reduce, Filter, Aggregations…
  - Binary stream transformations: CoMap, CoReduce…
  - Windowing semantics: Policy based flexible windowing (Time, Count, Delta…)
  - Temporal binary stream operators: Joins, Crosses…
  - Native support for iterations
Handle Imperfections – Watermarks

- Data items arrive early, on-time, or late
- Solution: Watermarks
  - Perfect or heuristic measure on when window is complete

Image: Tyler Akidau
Handle Imperfections – Watermarks

- Data items arrive early, on-time, or late
- Solution: Watermarks
  - Perfect or heuristic measure on when window is complete
Data Safety and Availability

• Ensure that operators see all events
  – “At least once”
  – Solved by replaying a stream from a checkpoint
  – No good for correct results

• Ensure that operators do not perform duplicate updates to their state
  – “Exactly once”
  – Several solutions

• Ensure the job can survive failure
Lessons Learned from Batch

- If a batch computation fails, simply repeat computation as a transaction
- Transaction rate is **constant**
- Can we apply these principles to a true streaming execution?
Taking Snapshots – the naïve way

Initial approach (e.g., Naiad)
- Pause execution on t1, t2, ...
- Collect state
- Restore execution
Asynchronous Snapshots in Flink

Propagating markers/barriers

Automatic partitioning and scaling

• 3 Types of Parallelization

(a) Pipeline-parallel A || B.  
(b) Task-parallel D || E.  
(c) Data-parallel G || G.

• Big streaming systems should support all three
Big Data Streaming Systems
Streaming Systems Overview

Closed Source
- Cloud DataFlow (BigTable)
- Naiad
- StreamInsights

Open Source
- Flink
- Apache
- kafka
- STORM
- Samza

Academia
- Esper
- Aurora
- NiagaraCQ
- CQL

InfoSphere Stream Processing Language (SPL)
- AWS Kinesis
Closed Source/Commercial Systems

**Cloud DataFlow:**
- Unified primitives for batch and stream processing
- Runs in Google's cloud only
- Open Source SDK (programs can run on other systems)
- Check out the Apache Beam Project! (http://beam.apache.org/)

**Google**

**BigTable:**
- Not a real streaming solution
- Allows to feed streams as source into a google DB
- Data can be immediately queried

**Naiad:**
- Goals of Naiad:
  - High throughput (typical for batch processors)
  - Low latency (known from single system stream processors)
- Is able to process iterative data flows
- Can discretize windows only based on time

**Microsoft**

**StreamInsights:**
- Available through Microsoft's cloud
- Windows based on count-, time- and punctuation/snapshot
- Optimized for .NET framework applications

**IBM**

**InfoSphere:**
- Well specified in several publications
- Can be deployed in customer clusters
- Own SQL-like query language enables many optimization means
- Window discretization based on trigger- and eviction policies
Open Source Systems by Apache (1/2)

- Reliable handling of huge numbers of concurrent reads and writes
- Can be used as data-source / data-sink for Storm, Samza, Flink, Spark and many more systems
- Fault tolerant: Messages are persisted on disk and replicated within the cluster. Messages (reads and writes) can be repeated

- True streaming over distributed dataflow
- Low level API: Programmers have to specify the logic of each vertex in the flow graph
- Full understanding and hard coding of all used operators is required
- Enables very high throughput (single purpose programs with small overhead)

- True streaming built on top of Apache Kafka and Hadoop YARN
- State is first class citizen
- Low level API
Spark implements a batch execution engine
• The execution of a job graph is done in stages
• Operator outputs are materialized in memory (or disk) until the consuming operator is ready to consume the materialized data

Spark uses Discretized Streams (D-Streams)
• Streams are interpreted as a series of deterministic batch-processing jobs
• Micro batches have a fixed granularity
• All windows defined in queries must be multiples of this granularity

Flink’s runtime is a native streaming engine
• Based on Nephele/PACTs
• Queries are compiled to a program in the form of an operator DAG
• Operator DAGs are compiled to job graphs
• Job graphs are generic streaming programs

Flink implements “true streaming”
• The whole job graph is deployed concurrently in the cluster
• Operators are long-running: Continuously consume input and produce output
• Output tuples are immediately forwarded to succeeding operators and are available for further processing (enables pipeline parallelism)
Further open source systems

Esper
- Open source Complex Event Processing (CEP) engine
- Tightly coupled to Java: **Executable on J2EE application servers**
- Describing events in Plain Old Java Objects (POJOs)
- Time-based or count-based windows

Aurora
- First design and implementation that parallelizes stream computation including rich operation and windowing semantics
- Windows are always specified as ranges on some measure
- Was continued in Borealis Project

NiagaraCQ
- Focuses more on scalability than on the flexibility
- Provides various optimizations techniques to share common computation within and across queries
- Only time-based windows are possible

CQL
- Continuous query language
- Implemented by the STREAM DSMS at Stanford
- Captures a wide range of streaming application in an SQL-like query language
Cloud-Based Streaming Systems (example)
Storm, Spark Streaming, and Flink
Big Data Analytics Ecosystem

**Applications & Languages**
- Hive
- Cascading
- Giraph
- Mahout
- Pig
- Crunch

**Data processing engines**
- MapReduce
- Flink
- Spark
- Storm
- Tez

**App and resource management**
- Yarn
- Mesos

**Storage, streams**
- HDFS
- HBase
- Kafka
- ...
Apache Storm

Scalable Stream Processing Platform by Twitter

- Tuple wise computation
- Programs are represented in a **topology** graph
  - **vertices** are computations / data transformations
  - **edges** represent data **streams** between the computation nodes
  - streams consist of an unbounded sequence of data-items/tuples
- Low-level stream processing engine

**Topology:**

The source of a stream is called **spout**.
(listening to data feed)

Computation vertices are called **bolt**.
(doing data manipulation)
Storm’s Fault Tolerance

- At least once guarantee via acknowledgments

- Acker logs progress of each tuple emitted by a spout
public class DoubleAndTripleBolt extends BaseRichBolt {
    private OutputCollectorBase _collector;

    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollectorBase collector) {
        _collector = collector;
    }

    @Override
    public void execute(Tuple input) {
        int val = input.getInteger(0);
        _collector.emit(input, new Values(val*2, val*3));
        _collector.ack(input);
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("double", "triple"));
    }
}
Building a Storm Topology

1) Use the TopologyBuilder class to connect spouts and bolts:
   builder.setSpout("name", new MySpout());
   builder.setBolt("name", new MyBolt());

2) Additionally, specify groupings to allow parallelization (shuffle, all, global, field)
   builder.shuffleGrouping("BoltName");

3) Create topology using the factory method
   StormTopology st = builder.createTopology();

4) Use LocalCluster class to test the topology
   LocalCluster cluster = new LocalCluster();
   cluster.submitTopology("name", new Config(), st);
Storm – Trident

• High-level abstraction built on top of Storm core:
  – operators like filter, join, groupBy, ...
• Stream-oriented API + UDFs
• Stateful, incremental processing
• Micro-Batch oriented (ordered & partitionable)
• Exactly-once semantics
• Trident topology compiled into spouts and bolt
Storm – Heron

- New real-time streaming system based on Storm
- Introduced June 2015 by Twitter (SIGMOD)
- Fully compatible with Storm API
- Container-based implementation
- Back pressure mechanism
- Easy debugging of heron topologies through UI
- Better performance than Storm (latency + throughput)

- No exactly once guarantee
Apache Spark

- In memory abstraction for big data processing
  - Resilient Distributed Data Sets
  - Fault-tolerance through lineage
  - Right APIs for all kinds of processing

Loop outside the system, in driver program
Iterative program looks like many independent jobs
Spark Job

- Similar to MR, but much faster

Image: Tyler Akidau
Spark Streaming

- Key abstraction: discretized streams (DStream)
  - micro-batch = series of RDDs
  - Stream computation = series of deterministic batch computation at a given time interval
- API very similar to Spark core (Java, Scala, Python)
  - (stateless) transformations on DStreams: map, filter, reduce, repartition, cogrop, ...
  - Stateful operators: time-based window operations, incremental aggregation, time-skewed joins
- Exactly-once semantics using checkpoints (asynchronous replication of state RDDs)
- No event time windows
Apache Flink

Apache Flink is an open source platform for scalable batch and stream data processing.

- The core of Flink is a distributed streaming dataflow engine.
  - Executing dataflows in parallel on clusters
  - Providing a reliable foundation for various workloads
- **DataSet** and **DataStream** programming abstractions are the foundation for user programs and higher layers

http://flink.apache.org
Architecture

• Hybrid MapReduce and MPP database runtime

• Pipelined/Streaming engine
  – Complete DAG deployed
Built-in vs. driver-based looping

Loop outside the system, in driver program
Iterative program looks like many independent jobs
Dataflows with feedback edges
System is iteration-aware, can optimize the job
Sneak peak: Two of Flink’s APIs

```scala
case class Word (word: String, frequency: Int)

**DataSet API** (batch):

```scala
val lines: DataSet[String] = env.readTextFile(...)
lines.flatMap {line => line.split(" ").map(word => Word(word,1))
  .groupBy("word").sum("frequency")
.print()
```

**DataStream API** (streaming):

```scala
val lines: DataStream[String] = env.fromSocketStream(...)
lines.flatMap {line => line.split(" ").map(word => Word(word,1))
  .keyBy("word")
  .window(Time.of(5,SECONDS)).every(Time.of(1,SECONDS))
  .sum("frequency")
.print()
```
Some Benchmark Results

Initially performed by Yahoo! Engineering, Dec 16, 2015,

[...]Storm 0.10.0, 0.11.0-SNAPSHOT and Flink 0.10.1 show sub-second latencies at relatively high throughputs[...]. Spark streaming 1.5.1 supports high throughputs, but at a relatively higher latency.

Processing Time vs Event Time

DEMO - STREAMING

„Inspired“ by
https://github.com/dataArtisans/oscon
Stream Optimizations

Overview

- **11 Optimizations** (numbered from 2 to 12)

<table>
<thead>
<tr>
<th>Section</th>
<th>Optimization</th>
<th>Graph</th>
<th>Semantics</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.</td>
<td>Operator reordering</td>
<td>changed</td>
<td>unchanged</td>
<td>(depends)</td>
</tr>
<tr>
<td>3.</td>
<td>Redundancy elimination</td>
<td>changed</td>
<td>unchanged</td>
<td>(depends)</td>
</tr>
<tr>
<td>4.</td>
<td>Operator separation</td>
<td>changed</td>
<td>unchanged</td>
<td>static</td>
</tr>
<tr>
<td>5.</td>
<td>Fusion</td>
<td>changed</td>
<td>unchanged</td>
<td>(depends)</td>
</tr>
<tr>
<td>6.</td>
<td>Fission</td>
<td>changed</td>
<td>(depends)</td>
<td>(depends)</td>
</tr>
<tr>
<td>7.</td>
<td>Placement</td>
<td>unchanged</td>
<td>unchanged</td>
<td>(depends)</td>
</tr>
<tr>
<td>8.</td>
<td>Load balancing</td>
<td>unchanged</td>
<td>unchanged</td>
<td>(depends)</td>
</tr>
<tr>
<td>9.</td>
<td>State sharing</td>
<td>unchanged</td>
<td>unchanged</td>
<td>static</td>
</tr>
<tr>
<td>10.</td>
<td>Batching</td>
<td>unchanged</td>
<td>(depends)</td>
<td>(depends)</td>
</tr>
<tr>
<td>11.</td>
<td>Algorithm selection</td>
<td>unchanged</td>
<td>(depends)</td>
<td>(depends)</td>
</tr>
<tr>
<td>12.</td>
<td>Load shedding</td>
<td>unchanged</td>
<td>changed</td>
<td>dynamic</td>
</tr>
</tbody>
</table>
Reordering and Elimination

2. OPERATOR REORDERING (A.K.A. HOISTING, SINKING, ROTATION, PUSH-DOWN)
Move more selective operators upstream to filter data early.

3. REDUNDANCY ELIMINATION (A.K.A. SUBGRAPH SHARING, MULTIQUERY OPTIMIZATION)
Eliminate redundant computations.
Operator Separation

4. OPERATOR SEPARATION (A.K.A. DECOUPLED SOFTWARE PIPELINING)

Separate operators into smaller computational steps.

Operator separation is profitable if it enables other optimizations such as operator reordering or fission, or if the resulting pipeline parallelism pays off when running on multiple cores.
Fusion

Goal: Reduce communication costs
Method: Shared memory among operators instead of network communication

Flink

In Apache Flink (and many other applications) we call this **chaining**
Fission

Directly maps to data parallelism:
Placement

Assigning Operators to slots

Co-locating Data and Computations
Load Balancing

8. LOAD BALANCING

Distribute workload evenly across resources.
State Sharing

9. STATE SHARING (A.K.A. SYNOPSIS SHARING, DOUBLE-BUFFERING)

Optimize for space by avoiding unnecessary copies of data.

Distributed File Systems
A single storage layer for the whole cluster

Chaining again...
Share memory among several operators instead of copying the data
Batching

“Under the hood” batch wise network traffic (buffering)

D-Streams*: All the stream processing is done in micro-batches

Algorithm Selection & Load Shedding

11. ALGORITHM SELECTION (A.K.A. TRANSLATION TO PHYSICAL QUERY PLAN)
Use a faster algorithm for implementing an operator.

\[ A_\alpha \rightarrow A_\beta \]

The optimizer selects the (hopefully) optimal join implementation

12. LOAD SHEDDING (A.K.A. ADMISSION CONTROL, GRACEFUL DEGRADATION)
Degrade gracefully when overloaded.

\[ \text{A} \rightarrow \text{Shredder} \rightarrow \text{A} \]
Cost Model

- Traditional cost-based query optimization is based on cardinality estimation → inadequate for unbounded streams
- Possible solution: rate-based cost estimation
  - (Viglas et al.: Rate-based query optimization for streaming information sources, SIGMOD 2002)

\[
\text{output rate} = \frac{\text{#outputs transmitted}}{\text{time for transmission}}
\]

- Challenges:
  - Fluctuating streams
  - Data-parallel processing

Slide by Kai-Uwe Sattler
Conclusion

**Introduction to Streams**
- Stream Processing 101
- How to do real streaming

**Stream Processing Systems**
- Ingredients of a stream processing system
- Storm, Spark, Flink
- Continuously evolving

**Stream Processing Optimizations**
- How to optimize
Thank You

Contact:
Tilmann Rabl
rabl@tu-berlin.de

We are hiring!
Further Reading

Historical papers on STREAM, Aurora, TelegraphCQ, Borealis, CQL, ...
- Papers and blogs on Storm, Heron, Flink, Spark Streaming, ...

Windows & Semantics
- Ghanem et al.: Incremental Evaluation of Sliding-Window Queries over Data Streams, TKDE 19(1), 2007
- Tucker et al.: Exploiting Punctuation Semantics in Continuous Data Streams, TKDE 15(3), 2003
- Krämer et al.: Semantics and Implementation of Continuous Sliding Window Queries over Data Streams, TODS 34(1), 2009

CEP:
- Wu et al.: High-Performance Complex Event Processing over Streams, SIGMOD 2006
- Schultz-Moeller et al.: Distributed Complex Event Processing with Query Rewriting, DEBS 2009

Fault Tolerance:
- Hwang et al.: High-availability algorithms for distributed stream processing, ICDE 2005

Partitioning & Optimization:
- Viglas et al.: Rate-Based Query Optimization for Streaming Information Sources, SIGMOD 2002