The Eight Requirements of Real-Time Stream Processing: STREAM vs Storm

Presentation by:
Alex Galakatos
John Meehan
Tianyu Qian
Introduction to Streams

- Why streaming processing?

- Two ideas
  - High-volume streams of real-time data
  - Low-latency
Applications

- **Stream filters**
- **Stream-relation joins**
  - Select `Rstream(Item.id, PriceTable.price)`
    From `Item [Now], PriceTable`
    Where `Item.id = PriceTable.itemId`
  - Stream items with current price appended
- **Sliding-window joins**
  - Select `Istream(*)`
    From `s1[rows 5], s2[rows 10]`
    Where `s1.A = s2.A`
  - Natural join of `s1` and `s2` with 5-tuple window on `s1` and 10-tuple window on `s2`
- **Streaming aggregations**
  - Produce relation, not streams
Introduction to Streams (cont)

- Streaming Softwares
- Two Types
  - DB-based
  - Application-based
Introduction to STREAM / CQL

- DSMS (data stream management system) designed by Stanford in the early/mid 2000's

- Three main goals
  - Exploit well-understood relational semantics
  - Queries performing simple tasks are easy to write
  - Simple yet expressive

- SQL-like language
Streams and Relations

- **Streams**
  - Continuous, possibly infinite multiset of elements \{tuple, timestamp\}

- **Relations**
  - Static, finite multiset of tuples belonging to a given timestamp

Example: Moving vehicles through tolls
Streams vs Relations

- CQL is designed to perform all transformative operations on relations
- Streams are converted into relations before operations are performed, and then back into streams
- Tuples with the same timestamp are treated as a relation, similar to a "batch"
Transform Relations to Streams

Three methods of generating a new stream

- **Istream** (insert stream)
  - new tuple at present
- **Dstream** (delete stream)
  - tuple removed at present
- **Rstream** (relation stream)
  - tuple exists at present
Introduction to Storm

- "Workflow engine" or "Computation Graph"
- Distributed, fault tolerant stream processing
- Hadoop : MapReduce Job :: Storm : Topology
- Scales horizontally
- No single point of failure
Topology

- Topology
  - network of spouts & bolts
  - runs indefinitely
- Spout -- source of a stream (Twitter API, queue)
- Bolt -- processes input stream(s) and can produce output stream(s)
Example

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("words", new TestWordSpout());

builder.setBolt("exclaim1", new ExclamationBolt()).shuffleGrouping("words");

builder.setBolt("exclaim2", new ExclamationBolt()).shuffleGrouping("exclaim1");
```
Features

● Guarantees
  ○ EVERY tuple will be processed
  ○ At-least-once & exactly once processing

● Fault Tolerant
  ○ Worker failures (Supervisor)
  ○ Coordinator failures (Nimbus)

● Scalable on commodity hardware
● Open Source
● Bolts defined in any language
Rule 1: Keep the Data Moving

- Latency of Storage operations and polling
- Process messages "in-stream"
- No requirement to store to perform any operations
- Active processing model (non-polling)

Figure 1: “Straight-through” processing of messages with optional storage.
Rule 1: STREAM / CQL

- **Push-based system**
  - Actively processes data as it arrives

- Able to output results as streams

- Stores data as a relation once operations are performed (joins, aggregates, etc.)

- Designed to facilitate incremental processing
Rule 1: Storm

- Data processed in real-time
- ZeroMQ used for messaging
  - Asynchronous messaging library
  - Push based communication
  - Automatic batching of messages
- No data is written during processing
Rule 2: Query using SQL on Streams

- Low-level language VS high-level "StreamSQL" language
- Built-in extensible stream-oriented primitives and operators
  - Window, Aggregate, joins

Figure 2: Windows define the scope of operations. The window has a size of 5 messages and slides by 1 each time the associated operator is executed. Consecutive windows overlap.
Rule 2: STREAM / CQL

- All comparisons are done between relations
- CQL is very SQL-like in its design
- Uses sliding window system
Rule 2: STREAM / CQL (cont)

Types of sliding windows:

- **Time-based**
  - Uses only tuples from recent timestamps

- **Tuple-based**
  - Uses the last n tuples provided by the stream

- **Partitioned windows**
  - "Group-by" window that returns the latest n aggregated tuples

- **Windows with a "slide" parameter**
  - Time-based, but with a specified range
Rule 2: Storm

- All functionality defined in a general purpose language
  - Bolts
  - Spouts

- More control but more complex

- Basic functionality must be defined by user
  - Windowing
  - Joins
  - Aggregates
Rule 2: Storm (cont.)

- Central window manager
- Using stream grouping to achieve windowing
  - Shuffle Grouping
  - Field Grouping
  - All Grouping
Rule 3: Handle Stream Imperfections

- Delayed data & time out
- Out of order data & stay open
- Time out vs. data moving
Rule 3: STREAM / CQL

- Processes each timestamp as a "batch"
- Must be able to recognize that all tuples for one "batch" have arrived
- Uses meta-input called "heartbeats"
  - Indicates that no new tuples will arrive with that timestamp
Rule 3: STREAM / CQL (cont)

Methods by which heartbeats are generated:

- Assigned using the DSMS clock when stream tuples arrive

- Stream source can generate its own heartbeats (only if tuples arrive in order)

- Properties of stream sources and the system environment can be used
Rule 3: Storm

- Manually handle imperfections in spout definition
  - Missing data
  - Out of order data

- Timeouts for blocking calculations specified in bolt definition
Rule 4: Generate Predictable Outcomes

- **Time-ordered, deterministic processing**
  - Example:
    - `TICKS(stock_symbol, volume, price, time)`
    - `SPLITS(symbol, time, split_factor)`
  - Process in ascending order
  - Out-of-order process results in wrong ticks
  - Sort-order messages are insufficient

- **Fault tolerance and recovery**
  - Replay & reprocess
Rule 4: STREAM / CQL

- **Time-based windowing is deterministic**
  - All tuples within a window of timestamps are processed

- **Tuple-based windowing is NOT deterministic**
  - No guarantee which tuples are processed
Rule 4: Storm

- Non-deterministic processing

- Use stream grouping to ensure deterministic processing
  - Field Grouping -- same tuple goes to same node
Rule 5: Integrated Stored and Streaming Data

- Compare "Present" with "Past"
  - Store, access, and modify state information

- Two motives
  - Switch to a live feed seamlessly (Trading app)
  - Compute from past and catch up to real time

- Low Latency
  - State stored in the same OS address space as application using an embedded database system
Rule 5: STREAM / CQL

● All streams are processed as relations, allowing easy comparison to other relations
  ○ Streams CANNOT be directly operated upon
  ○ Highly convenient for comparing stored data to streaming data

● Uses sliding window system in order to convert streams to relations
Rule 5: Storm

- Interact with database using a Bolt
  - Perform joins with stored data
  - Insert value into database
  - Modify existing stored data

- No common language
- JDBC / ODBC
Rule 6: Guarantee Data Safety and Availability

- "Tandem-style" hot backup and failover
- Secondary system synchronization

Figure 3: “Tandem-style” hot backup and failover can ensure high availability for real-time stream processing.
Rule 6: STREAM / CQL

- Provides similar data security to DBMS
- No obvious form of data backup, but could be accomplished with two separate systems taking in the same stream
Rule 6: Storm

- Guaranteed tuple processing
  - At-least-once
  - Exactly-once (Trident)

- Highly available / Automatic recovery
  - Worker node failure
  - Supervisor failure
  - Nimbus failure
Rule 7: Partition and Scale Applications Automatically

- Distribute processing across multiple processors and machines
- Incremental scalability
- Facilitating low latency
Rule 7: STREAM / CQL

- No distributed system

- Load shedding
  - Dynamically degrades performance based on the velocity of incoming data
  - Reduces load in order to minimize latency
  - Load manager chooses locations that will distribute error evenly across all queries
Rule 7: STREAM / CQL (cont)

Load Shedding
Rule 7: Storm

- Distributed
  - set number of workers
  - set level of parallelism for each component
- Automatic rebalancing for adding nodes

![Diagram of distributed system]
Rule 8: Process and Respond Instantaneously

- Low latency & real-time response

- Highly-optimized, minimal-overhead execution engine
  - minimize the ratio of overhead to useful work
  - All system components to be designed with high performance
Rule 8: STREAM / CQL

- Query plans are merged with existing plans when possible

- Heuristics to improve efficiency
  - Push selections below joins
  - Maintain and use indexes
  - Share synopses and operators
Rule 8: Storm

- Disk write not in critical path
- ZeroMQ used for efficient network communication
- Performance varies by topology
- One benchmark: 1m tuples per node per sec
Conclusions

- Greatly depends on the application
  - Not one-size-fits-all

- Rules were made to be broken
  - SQL not necessarily required
  - Non-deterministic processing can be ok

- Some rules more important than others
  - Maintain velocity of data
  - Integrate stored and streaming data
  - Data availability/scalability
Works cited

- STREAM / CQL

- Storm
  - https://github.com/nathanmarz/storm/wiki/Tutorial