CS122D: Beyond SQL Data Management
—Lecture #18 —

Mike Carey
UC Irvine
mjcarey@ics.uci.edu
Announcements

• We’re on final approach to landing...

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>W 5/19</td>
<td>Big Data Analytics: Google, MapReduce, HDFS</td>
<td>Big Data Platforms paper (skim)</td>
</tr>
<tr>
<td>M 5/24</td>
<td>Big Data Analytics: Spark &amp; SparkSQL</td>
<td>Spark Overview paper (skim)</td>
</tr>
<tr>
<td>W 5/26</td>
<td>Big Data Analytics: Spark &amp; DataFrames</td>
<td>Databricks and Spark materials (as needed)</td>
</tr>
<tr>
<td>W 6/02</td>
<td>Data Stream Systems: Spark Structured Streaming</td>
<td>Databricks and Spark materials (as needed)</td>
</tr>
<tr>
<td></td>
<td>Overflow</td>
<td>Column stores, search, message streams, timeseries/IoT, ...</td>
</tr>
<tr>
<td>M 6/07</td>
<td>Final Exam (Cumulative)</td>
<td>4:00-6:00 PM -- be there!!!</td>
</tr>
</tbody>
</table>

• HW #6 is in flight – and due next Monday

<table>
<thead>
<tr>
<th>HW6</th>
<th>Date</th>
<th>Time</th>
<th>Topic</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Th 5/27</td>
<td>Mo 6/07 (11:59 PM)</td>
<td>Spark</td>
<td>HW6 Setup HW6 Details Template HW6 Solution</td>
</tr>
</tbody>
</table>

• Two more quizzes still ahead!
  • Spark, and Course Feedback (after final)

• Today: Data Stream Systems...

• One more video: Whirlwind tour (of “everything” else) & wrap-up
A Typical Big Data Software Stack

Thanks to Tamer Özsu & Patrick Valduriez!

(*Only ~$60 on Amazon! 😊)
Traditional DBMS vs. Streaming

- Other differences
  - Push-based (data-driven)
  - Persistent queries
  - Unbounded stream
  - System conditions may not be stable

© 2020, M.T. Özsu & P. Valduriez
History

- **Data Stream Management System (DSMS)**
  - Typical DBMS functionality, primarily query language
  - Earlier systems: STREAM, Gigascope, TelegraphCQ, Aurora, Borealis
  - Mostly single machine (except Borealis)

- **Data Stream Processing System (DSPS)**
  - Do not embody DBMS functionality
  - Later systems: Apache Storm, Heron, Spark Streaming, Flink, MillWheel, TimeStream
  - Almost all are distributed/parallel systems

- Use the term **Data Stream System (DSS)** when the distinction is not important
DSMS Architecture

- Input Monitor
- Working Storage
- Summary Storage
- Static Storage
- Query Repository
- Query Processor
- Output Buffer

Updates to Static Data
User Queries

© 2020, M.T. Özsu & P. Valduriez
Stream Data Model

• Standard def’n: An append-only sequence of timestamped items that arrive in some order

• Relaxations
  • Revision tuples (e.g., updates/deletes)
  • Sequence of events that are reported continually (publish/subscribe systems)
  • Sequence of sets of elements (bursty arrivals)

• Typical arrival:
  \( \langle \text{timestamp}, \text{payload} \rangle \)
  • Payload changes based on system
    • Relational: tuple
    • Graph: edge
    • ...

© 2020, M.T. Özsu & P. Valduriez
Processing Models

• Continuous
  • Each new arrival is processed as soon as it arrives in the system
  • Examples: Apache Storm, Heron

• Windowed
  • Arrivals are batched in windows and executed as a batch
  • For user, recently arrived data may be more interesting and useful
  • Examples: Aurora, STREAM, Spark Streaming
Window Definition

• According to the direction of endpoint movement
  • Fixed window: both endpoints are fixed
  • Sliding window: both endpoints can slide (backward or forward)
  • Landmark window: one endpoint fixed, the other sliding

• According to definition of window size
  • Logical window (time-based) – window length measured in time units
  • Physical window (count-based) – window length measured in number of data items
  • Partitioned window: split a window into multiple count-based windows
  • Predicate window: arbitrary predicate defines the contents of the window
Stream Query Models

• Queries are typically persistent
• They may be monotonic or non-monotonic
• Monotonic: result set always grows
  • Results can be updated incrementally
  • Answer is a continuous, append-only stream of results
  • Results may be removed from the answer only by explicit deletions (if allowed)
• Non-monotonic: some answers in the result set become invalid with new arrivals
  • Recomputation may be necessary
Stream Query Languages

• Declarative
  • SQL-like syntax, stream-specific semantics
  • Examples: CQL, GSQL, StreaQuel

• Procedural
  • Construct queries by defining an acyclic graph of operators
  • Example: Aurora

• Windowed languages
  • size: window length
  • slide: how frequently the window moves
  • E.g.: size=10min, slide=5sec

• Monotonic vs. non-monotonic
Streaming Operators

- Stateless operators (a) are no problem: e.g., selection, projection
- Stateful operators (e.g., nested loop join) are blocking
  - You need to see the entire inner operand
  - State management can be a challenge
- Some blocking operators have non-blocking versions (symmetric hash join (b))
- Otherwise limit options to windowed execution
Query Processing over Streams

• Similar to relational, except
  • *persistent queries*: registered to the system and continuously running
  • data pushed through the query plan, not pulled

• Stream query plan
Query Processing Issues

• Continuous execution
  • Each new arrival is processed as soon as the system gets it
  • E.g. Apache Storm, Heron

• Windowed execution
  • Arrivals are batched and processed as a (small) batch
  • E.g. Aurora, STREAM, Spark Streaming

• More opportunities for multi-query optimization in the world of streaming data systems
  • E.g. easier to determine shared subplans
Windowed Query Execution

• Two main lifecycle events need to be managed
  • Arrivals
  • Expirations
• System actions depend on operators
  • E.g. Join generates new result, negation removes previous result
• Window movement also affects results
  • As window moves, some items in the window move out
  • What to do to the results...?
  • If monotonic, do nothing; if non-monotonic, two options:
    • Direct approach (expiration only)
    • Negative tuple approach
Load Management

• Stream arrival rate > processing capability
• Load shedding
  • Random
  • Semantic (priorities)
• Early drop
  • All of the downstream operators will benefit
  • Accuracy may be negatively affected
• Late drop
  • May not reduce the system load much
  • Allows the shared subplans to be evaluated
• Set aside (spill to disk)
  • Assuming things will calm down later
Out-of-Order Processing

• Assumption: arrivals are in timestamp order
• May not hold
  • Arrival order may not match generation order
  • Late arrivals $\rightarrow$ no more, or just late?
  • Multiple external (distributed) sources

• Approaches
  • Built-in slack
  • Punctuations
Multiquery Optimization

• More opportunity since the persistent queries are known beforehand
  • Aggregate queries over different window lengths or with different slide intervals
  • State and computation may be shared (usual)
Parallel Data Stream Processing

1) Partitioning the incoming stream
2) Execution of the operation on the partition
3) (Possibly) Aggregation of the results from multiple machines
Stream Partitioning

• Shuffle (round-robin) partitioning

■ Hash partitioning
Parallel Stream Query Plan

© 2020, M.T. Özsu & P. Valduriez
Structured Streaming


• Scalable, fault-tolerant stream processing engine
• Stream computations expressed in the same way as batch computations on static data (DataFrame)
• Internally uses a micro-batch processing approach
• Maybe we should start with a demo...?

https://community.cloud.databricks.com/?o=8435228212499315#notebook/1561419453258551
Input Sources

• File source
  • Reads files written in a directory as a stream of data (supports text, csv, json, orc, and parquet)

• Kafka source
  • Reads data from Apache Kafka message broker

• Socket source
  • Reads UTF8 data from a TCP socket (for testing)

• Rate source
  • Generates synthetic timestamp/value data (for testing)
Ex: File Source

M. Carey, Spring 2021: CS122D
Basic Concepts

Data stream as an unbounded table

Unbounded Table

new data in the data stream
= new rows appended to an unbounded table
Supported Output Modes:

- **Complete Mode**: Entire result table is written to external storage
- **Append Mode**: Only newly appended rows are written to external storage
- **Update Mode**: Only newly updated rows are written to external storage

Programming Model for Structured Streaming
Ex: Streaming Wordcount

```scala
# Create DataFrame representing the stream of input lines from connection to localhost:9999
lines = spark \
  .readStream \
  .format("socket") \
  .option("host", "localhost") \
  .option("port", 9999) \
  .load()

# Split the lines into words
words = lines.select(  
  explode(  
    split(lines.value, " ")  
  ).alias("word")
)

# Generate running word count
wordCounts = words.groupBy("word").count()
```
Ex: Streaming Wordcount (cont.)

Model of the Quick Example

M. Carey, Spring 2021: CS122D
Selection, Projection, Aggregation

def = ...  # streaming DataFrame with IOT device data with schema { device: string, deviceType: string, signal: double, time: DateType }

# Select the devices which have signal more than 10
df.select("device").where("signal > 10")

# Running count of the number of updates for each device type
df.groupBy("deviceType").count()

df.createOrReplaceTempView("updates")
spark.sql("select count(*) from updates")  # returns another streaming DF

df.isStreaming()
Windows on Event Time

- *Ex:* Assume word count input stream contains lines along with the **time** when the line was generated
- Suppose we want to count words within 10 minute windows, updating every 5 minutes
- *I.e.,* track the word counts received in 10 minute windows 12:00-12:10, 12:05-12:15, 12:10-12:20, ...
- *Note:* a word received at 12:07 will affect 2 windows

```python
words = ...  # streaming DataFrame of schema { timestamp: Timestamp, word: String }

# Group the data by window and word and compute the count of each group
windowedCounts = words.groupBy(
    window(words.timestamp, "10 minutes", "5 minutes"),
    words.word
).count()
```
Windows on Event Time (cont.)

Input Stream

Time
12:00 12:05 12:10 12:15

Result Tables after 5 minute triggers

Counts incremented for windows 12:00 - 12:10 and 12:05 - 12:15

Windowed Grouped Aggregation with 10 min windows, sliding every 5 mins

Counts incremented for windows 12:05 - 12:15 and 12:10 - 12:20
Late Data and Watermarking

Late data handling in Windowed Grouped Aggregation

```java
words = ... # streaming DataFrame of schema { timestamp: Timestamp, word: String }

# Group the data by window and word and compute the count of each group
windowedCounts = words \\
    .withWatermark("timestamp", "10 minutes") \\
    .groupBy(
        window(words.timestamp, "10 minutes", "5 minutes"),
        words.word)
    .count()
```
Join Operations

• Can join a streaming DataFrame with a static DataFrame or another streaming DataFrame

• Stream-static Joins are straightforward

```python
staticDf = spark.read. ...
streamingDf = spark.readStream. ...
streamingDf.join(staticDf, "type")  # inner equi-join with a static DF
streamingDf.join(staticDf, "type", "right_join")  # right outer join with a static DF
```

(Basically the same as static-static DataFrame Joins)
Join Operations (cont.)

• Stream-stream Joins present a state challenge (!)
• Inner Joins with optional Watermarking

```python
from pyspark.sql.functions import expr

impressions = spark.readStream. ... 
clicks = spark.readStream. ...

# Apply watermarks on event-time columns
impressionsWithWatermark = impressions.withWatermark("impressionTime", "2 hours")
clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")

# Join with event-time constraints
impressionsWithWatermark.join(
    clicksWithWatermark,
    expr("......
    clickAdId = impressionAdId AND
    clickTime >= impressionTime AND
    clickTime <= impressionTime + interval 1 hour
    ....")
)
```

How late can the data be?
How long after an ad can a click be?
Structured Streaming Recap


- Scalable, fault-tolerant stream processing engine
- Stream computations expressed in the same way as batch computations on static data (*DataFrame*)
- Internally uses a *micro-batch* processing approach
- Or maybe we should end with a demo...?
  
  [https://community.cloud.databricks.com/?o=8435228212499315#notebook/1561419453258551](https://community.cloud.databricks.com/?o=8435228212499315#notebook/1561419453258551)
- See the docs for (lots more) info and details!
Questions?