CMSC 724: Database Management Systems
Data Streams and Dataflow Engines

Instructor: Amol Deshpande
amol@cs.umd.edu
Overview and Early Work

Maintenance of Materialized Views

Models and issues in data stream systems

Discretized streams: fault-tolerant streaming computation at scale

Apache Flink: Stream and Batch Processing in a Single Engine

Incremental, Iterative Data Processing with Timely Dataflow

MacroBase: Prioritizing Attention in Fast Data
Data Streams

Why?
- Much data generated continuously (growing every day)
- Financial data
- Sensors, RFID
- Network/systems monitoring
- Video/Audio data
- etc...

Need to support:
- High data rates
- Real-time processing with low latencies
- Support for temporal reasoning (time-series operations)
- Data dissemination
- Distributed? (at least data generation)
- etc...
Examples of Tasks

- Continuous (SQL) queries
  - E.g. moving average over last hour every 10 mins
  - SQL extended to support “windows” over streams
  - Proposed extensions: SEQUENCE, CQL, StreamSQL

- Pattern recognition
  - Alert me when: A, then B within 10 mins
  - How to specify? StreamSQL has some support

- Probabilistic modeling; Applying financial models
  - Infer hidden variables
  - Remove noise (from measured readings)
  - Do complex analysis to decide whether to buy
  - We don’t even know how to specify these

- Multimedia data?
  - Online object detection, activity detection
  - Correlating events from different streams
How to Execute?

- Use traditional DBMS?
- Consider simplest case:
  - Report moving average over last hour every 10 minutes
  - 1. Insert all new items into database
  - 2. Execute the query every 10 minutes
- Not easily generalizable to other tasks
  - E.g. “alert me the moment moving average > 100”?
- Typically 1000’s of such continuous queries
- Even for one query, too slow and inefficient
  - Doesn’t reuse work from previous execution
- Application-level modules typically used for complex tasks
Materialized Views
- Derived tables that must be kept up-to-date when source tables change

Triggers?
- Similar, but current trigger systems not designed for the required scale

Publish-Subscribe Systems
- Similar concepts: Push-based, reactive execution
- Typically no complex queries
- Much focus on “dissemination”

Major research systems (late 90’s-early 00’s):
- NiagaraCQ (Wisc), Telegraph, TelegraphCQ (Berkeley) STREAM (Stanford), Autora, Borealis, Medusa (Brown/Brandeis/MIT)

Commercial?
- Different design points supported by different systems today
Outline

- Overview and Early Work
- Maintenance of Materialized Views
- Models and issues in data stream systems
- Discretized streams: fault-tolerant streaming computation at scale
- Apache Flink: Stream and Batch Processing in a Single Engine
- Incremental, Iterative Data Processing with Timely Dataflow
- MacroBase: Prioritizing Attention in Fast Data
Materialized Views

- View: A derived relation defined as an SQL expression over base relations
  - More generally, any derived product (e.g., an ML model) generated from a set of source datasets (e.g., a collection of images) using an automated query/program (e.g., a training program)

- Materialized views?
  - Views, by definition, are just expressions
  - Need to computed when required by running the query over base tables
  - Materialization == pre-computation
    - Benefits: much lower latencies when querying a view
    - Drawbacks: need to ”maintain”, i.e., modify the materialized result when the base tables change
"Incremental" much faster in most cases
  - i.e., figure out the changes to the materialized view given the changes to the base tables

Many dimensions to consider
  - What information is available/required for view maintenance?
    - What if the materialized view itself is not available any more? (This situation is closer to the “data streams” scenario)
  - What types of modifications need to be supported?
  - How are the views expressed?
  - ...

For each point in this space, we may have a different algorithm
Example 1

- Relation: part(part_no, part_cost, contract)
- View:
  \[
  \text{expensive-parts}(\text{part_no}) = \Pi_{\text{part_no}} \sigma_{\text{part}_\text{cost}>1000}(\text{part})
  \]
- Some possibilities when inserting a new tuple “part(p1, 5000, c15)” into “part”
  - Only the materialized view is available: We can check if p1 is already present in it, and insert if not
  - Only the base table is available:
    - Check if there exists another tuple with part_no = p1 and cost >1000
    - If yes, no need to insert into the view
    - If no, insert into the view
  - If part_no is a key for part \(\rightarrow\) insert p1 into the view
Example 1

- Relation: part(part_no, part_cost, contract)
- View:
  \[
  \text{expensive_parts(part.no) = II}_{\text{part no } \sigma \text{part cost} > 1000(\text{part})}
  \]
- Harder to handle deletions into “part” though
  - e.g., if part(p1, 2000, c12) is deleted
  - Access to the view alone is not sufficient
  - Need to check if another tuple with p1 and cost > 1000 exists in parts table
Example 2

- Relation: part(part_no, part_cost, contract), and supplier(supp_no, part_no, price)

- View:

  \[ \text{supp} \text{_parts}(\text{part}_\text{no}) = \Pi_{\text{part}_\text{no}}(\text{supp} \bowtie_{\text{part}_\text{no}} \text{part}) \]

- Insert: part(p1, 5000, c15)
  - If supp_parts already contains p1, then no effect
  - If supp_parts doesn’t contain p1, then need access to the supplier relation
Example 3

- Incremental maintenance focuses on defining changes to the output in terms of changes to the inputs.
- Base relation: $\text{link}(S, D)$
- View to define one-hop neighbors

$$ D : \text{hop}(X, Y) = \Pi_{X,Y}(\text{link}(X, V) \bowtie_{V=W} \text{link}(W, Y)) $$

- If a set of tuples inserted into link: $\Delta(\text{link})$
- Changes to the view:

$$ \Delta(\text{hop}) = \Pi_{X,Y}((\Delta(\text{link})(X, V) \bowtie_{V=W} \text{link}(W, Y)) \cup (\text{link}(X, V) \bowtie_{V=W} \Delta(\text{link})(W, Y)) \cup (\Delta(\text{link})(X, V) \bowtie_{V=W} \Delta(\text{link})(W, Y))) $$
Counting Algorithm [GMS93]

- Works for queries with UNION, negation, and aggregation (no joins)
- For each tuple in the view, keep track of the number of different derivations for that tuple
- When an update is made, run the same query to decide how much the count for each tuple changes

- \( \text{link} = \{(a, b), (b, c), (b, e), (a, d), (d, c)\} \)
- \( \text{hop} = \{(a, c), (a, e)\} \)
  - Counts maintained internally: 2 and 1 resp.
- Say, \((a, b)\) is deleted from link
  - The deletion "counts": \((a, c) \rightarrow -1, (a, e) \rightarrow -1\)
  - So we remove \((a, e)\), but keep \((a, c)\)

- General idea extensible to other types of queries, including joins
Outer-join Views
- Need to handle NULLs carefully
  - e.g., a deletion from one of the tables may require insert into the view with padded NULLs

Recursive Views
- Naturally more complex
Partial Information

- Not always possible to maintain a view without access to all the base relations

- No information: In some cases, can decide that an update does not affect a view
  - However, if it does -- need to possible use the base relations to do the update

- Self-maintainable Views
  - Can be maintained just with access to the view and constraints
  - In some cases, access to a subset of the tables is sufficient
Overview and Early Work

Maintenance of Materialized Views

Models and issues in data stream systems

Discretized streams: fault-tolerant streaming computation at scale

Apache Flink: Stream and Batch Processing in a Single Engine

Incremental, Iterative Data Processing with Timely Dataflow

MacroBase: Prioritizing Attention in Fast Data
Differences from stored relation model

- Data elements arrive one at a time
- System has no control over the order of arrival
- Data streams potentially unbounded in size
- Access to past elements not provided (unless explicitly stored)

Queries

- One-time queries: evaluated against a snapshot of the tables
- Continuous queries: evaluated continuously and itself produces a data stream
Network Traffic Management

- Network packet traces being collected in a network for two links: C = customer link, B = backbone link
  - src: IP address of the sender
  - dest: IP address of destination
  - id, len, time

- Query: Compute load on link B every minute

  \[ Q_1: \text{SELECT} \quad \text{notifyoperator}(\text{sum(len)}) \]
  \[ \text{FROM} \quad B \]
  \[ \text{GROUP BY} \quad \text{getminute}(\text{time}) \]
  \[ \text{HAVING} \quad \text{sum(len)} > t \]

  - Semantics: need to evaluate continuously as new tuples arrive
  - Could be supported through use of “triggers”, but too heavy-weight
  - Also, may want to employ approximation techniques for large volumes
Network Traffic Management

- Network packet traces being collected in a network for two links: C = customer link, B = backbone link
  - src: IP address of the sender
  - dest: IP address of destination
  - id, len, time

- Query: Isolate traffic for each flow on B

  \[
  Q_2: \quad \text{SELECT} \quad \text{flowid, src, dest, sum(len) AS flowlen}
  \]
  \[
  \quad \text{FROM} \quad (\text{SELECT} \quad \text{src, dest, len, time}
  \quad \text{FROM} \quad B
  \quad \text{ORDER BY} \quad \text{time})
  \quad \text{GROUP BY} \quad \text{src, dest, getflowid(src, dest, time)}
  \quad \text{AS flowid}
  \]

- getflowid() is a UDF that specifies how to group packets across time
- Somewhat clumsy syntax trying to express in SQL
Network Traffic Management

- Network packet traces being collected in a network for two links: \(C =\) customer link, \(B =\) backbone link
  - src: IP address of the sender
  - dest: IP address of destination
  - id, len, time

- Query: Fraction of traffic that can be attributed to \(C\)

  \[ Q_3: (\text{SELECT count (*) FROM } C, B) \]

  \[ \text{WHERE } \begin{align*} 
  C.\text{src} &= B.\text{src} \land C.\text{dest} = B.\text{dest} \\
  &\land C.\text{id} = B.\text{id} \end{align*} / \\
  (\text{SELECT count (*) FROM } B) \]

  - Potentially very large intermediate tables
Network Traffic Management

- Network packet traces being collected in a network for two links: C = customer link, B = backbone link
  - src: IP address of the sender
  - dest: IP address of destination
  - id, len, time

- Query: Top 5 percent traffic

\[ Q_4: \text{WITH Load AS} \]
\[ \text{(SELECT src, dest, sum(len) AS traffic FROM B GROUP BY src, dest)} \]
\[ \text{SELECT src, dest, traffic FROM Load AS } L_1 \]
\[ \text{WHERE (SELECT count(*) FROM Load AS } L_2 \]
\[ \text{WHERE } L_2\text{.traffic } \lt L_1\text{.traffic) } \gt \]
\[ \text{(SELECT 0.95\times count(*) FROM Load)} \]
\[ \text{ORDER BY traffic} \]
Querying over Data Streams: Simple Architecture

Data Stream Management System (DSMS)

Data Streams

Continuous Queries/Tasks

Query/task results
At its simplest, a continuous query/task needs to specify:

- Frequency of execution: how often to execute the query/task
  - All the time (e.g., an “anomaly detection” task)
  - Every so often (e.g., every minute, every day, etc)
- The “scope” of the query/task: what data it operates on at any execution instance (could be different for different inputs)
  - The entire stream (i.e., all data ever received)
  - A lower-sized, potentially bounded transformation of the stream (e.g., all distinct elements every received -- could be bounded in some cases)
  - A window over the data (e.g., last one hour) -- usually called a “sliding” window
- What to do
  - SQL query over the inputs (per the scope)
  - Pattern detection (e.g., look for $A \rightarrow B$ within 10 seconds $\rightarrow C$ within 10 seconds)
  - Arbitrary user-defined tasks (e.g., ML tasks)
Querying: Performance Issues

- Key “intuitive” goal: process a newly arrived tuple before the next tuple arrives
  - On average -- okay if there is a queue for short periods of time
  - If not, the backlog will keep building up and system may have to drop tuples (Not acceptable)

- Data streams potentially unbounded in size ➔ answering queries exactly may require unbounded memory
  - Using ”external memory“ (i.e., disks) doesn’t help
    - Too slow
    - Eventually run out of external memory as well
  - Sliding windows and transformations help bound the memory requirements
Querying Issues: Approximate Query Answering

- May have to consider approximations if not possible to solve the query/task exactly
  - By maintaining a “summary” of the data stream so far

- A generic “summary” of the data streams characterized by two functions
  - \texttt{updateSummary()} given a new tuple
  - \texttt{computeAnswer()} using the summary

- Example of a summary: “random sample”
  - Can update a random sample when a new tuple comes in (non-trivial but not difficult)
  - Can compute an aggregate like “average”, “sum” at any point \(\rightarrow\) an unbiased estimate of the sum/average over the entire stream
Querying Issues: Approximate Query Answering

- Random samples don’t work well for many queries
  - Bad estimates for “joins” -- too many missing tuples
  - Can’t handle queries like number of distinct elements, number of triangles in a graph, etc.

- Can use other summaries like “histograms” in some cases

- Sketches
  - Purpose-built summaries for specific tasks
  - Much work over the last two decades on new sketching techniques
  - Usually provide error guarantees
Querying Issues: Blocking Operators

- Standard operators like sorting, aggregations, are problematic in data streams context
  - Assuming we re-purpose an existing query processing architecture
  - Less of an issue if sliding windows are being used

- Non-blocking operators like “symmetric” hash joins preferred

- ”Punctuations”: Assertions about the data elements in the stream that are yet to arrive
  - e.g., you won’t see any more tuples with A = 10
  - Can be used to finish computations in some cases
  - An interesting, but relatively-less-explored concept
Consider an “ad hoc” (one-time) query that refers to data received in the past
  ◦ ... that may have been thrown away during continuous processing

Option 1: Don’t allow queries like this

Option 2: Store all data ever received somewhere
  ◦ Probably the most common approach we would see today
  ◦ People are loath to throw away any data

Option 3: Use summaries of data
  ◦ Depends on whether approximations are permissible
Querying Issues: Summary

- Too many different considerations for how querying may be handled
- No single system that can handle all such cases efficiently

Most modern systems focus on specific sets of use cases

- General purpose micro-batching systems (e.g., Spark Streaming): don’t do well with small batches
- Real-time anomaly, event, or pattern detection system: usually support a small set of patterns/queries/tasks, and build incremental techniques for those
- Approximate Aggregations: Use sketching or other summary techniques to monitor specific things (e.g., heavy hitters, distinct counts, etc), or build dashboards
- Materialized views maintenance: Incremental maintenance of a small set of views, sometimes in a lazy fashion -- no claims to real-time
- ...
Stanford STREAM System: Query Language

- Extend SQL with sliding windows, time-based or length-based

Calls: `customer_id, type, minutes, and timestamp`.

**Average call length across the last 10 long-distance calls made by each customer**

```sql
SELECT AVG(S.minutes) FROM Calls S [PARTITION BY S.customer_id ROWS 10 PRECEDING] WHERE S.type = 'Long Distance'
```

**Average call length across the long-distance calls, among the last 10 made by each customer**

```sql
SELECT AVG(S.minutes) FROM Calls S [PARTITION BY S.customer_id ROWS 10 PRECEDING] WHERE S.type = 'Long Distance'
```

**Average call length across the last 1000 calls made by Gold customers**

```sql
SELECT AVG(V.minutes) FROM (SELECT S.minutes FROM Calls S, Customers T WHERE S.customer_id = T.customer_id AND T.tier = 'Gold') V [ROWS 1000 PRECEDING]
```

Implicit assumption that customers is a static table. Otherwise, semantics can be tricky.
Stanford STREAM System: Timestamps

- Option 1: Assign timestamps when they enter the system
  - Not clear what happens if the DSMS itself is distributed across a network
  - Also, what about “composite” tuples where the base tuples have different timestamps?
    - Usually use the latest timestamp (may have the user specify, but more tricky)

- Option 2: Assign timestamps at the source
  - Clocks are usually not synchronized sufficiently, especially in IoT settings
  - Need to worry about delays in tuples getting to the DSMS (e.g., how do you a sliding window is “complete”?)

- Similar issues studied in the context of *temporal databases*
Similar to the standard operator model, but with continuously running operators

Each operator maintains “synopses” to handle large volumes of data

Adaptive operators to handle dynamicism (similar motivation as eddies)
  - Operators adapt to memory by using approximations
  - Lot of open questions (some looked at in followup work)

Scheduling of operators, and multiple query plans also complex
Algorithmic Issues

- **General setting:**
  - A stream of values: $x_1, x_2, \ldots, x_N, \ldots$
  - Each value seen only once, in that order
  - At all times $N$, compute function: $f(x_1, \ldots, x_N)$, i.e., the prefix of the stream of size $N$

- **Optimization goals:**
  - Memory required -- ideally logarithmic in $N$
  - Time required for each new tuple -- ideally logarithmic in $N$

- Most techniques maintain (and update) a small summary structure that allows computing $f$ in an unbiased manner
Algorithmic Issues: Random Sampling

- For some function $f$, random samples work
  - e.g., average, sum, etc.
- **Reservoir Sampling** algorithm to maintain a sample of size $k$:
  - First $k$ elements are the initial sample
  - When we see the $N^{th}$ element:
    - Choose a random integer between 1 and $N$
    - If $\leq k$, then replace existing element at that position with the $N^{th}$ element
  - Can be proven to maintain a random sample of the prefix at all times

- Somewhat slow -- can be made more efficient by instead (randomly) computing how many elements to skip
- Also can be modified to handle “weights”
Algorithmic Issues: Sketching Techniques

- Alon, Matias, Szegedy: Space Complexity of Approximating the Frequency Moments; STOC 1996
- Consider a stream: (1, 2, 3, 1, 5, 2, 1, 3, 4)
- Let $m_i$ be the frequency of $i$ in the stream
  - $m_1 = 3, m_2 = m_3 = 2, m_4 = m_5 = 1.$
- Frequency moment $F_k = \sum m_i^k$
  - $F_0 = 5 = \text{number of distinct elements in the stream}$
  - $F_1 = 9 = \text{total number of elements in the stream (trivial)}$
  - $F_2 = 19 = \text{comes in up many places (e.g. self-join size of a relation)}$
  - $F_\infty = \text{most frequent item’s multiplicity}$
- Flajolet-Martain technique is for $F_0$
Algorithmic Issues: Sketching Techniques

- Alon, Matias, Szegedy: Space Complexity of Approximating the Frequency Moments; STOC 1996
- Improves upon the $F_0$ computation
- Key contribution: a technique for $F_2$
  - Hash every element $x_i$ onto $\{-1, +1\}$
  - Keep a running total: $\text{SUM } x_i \times \text{hash}(x_i)$
  - In the end, take square of the ”sum” as the estimator
  - Can prove strong guarantees about it
  - Hash functions need some properties
- Lot of work since then on other types of computations, better guarantees, etc.
- Techniques widely used in Google etc., for a variety of purposes
Algorithmic Issues: More

- Histograms, Wavelets, Heavy Hitters
  - Used to get at other properties of data sets (e.g., distributions, most frequent elements, etc)
  - Lot of work on specific techniques for maintaining those incrementally

- Not always possible to use sketching
  - e.g., $F_\infty$ requires storing the entire stream
  - Can do better if the count is a high fraction of the entire stream (i.e., if there is significant skew)

- Key Takeaway: Sketching techniques can be used in many cases to drastically reduce the dataset sizes, and are quite practical
Overview and Early Work

Maintenance of Materialized Views

Models and issues in data stream systems

Discretized streams: fault-tolerant streaming computation at scale

Apache Flink: Stream and Batch Processing in a Single Engine

Incremental, Iterative Data Processing with Timely Dataflow

MacroBase: Prioritizing Attention in Fast Data
Motivation

- Need streaming computation models analogous to MapReduce for batch processing
  - Applications: detecting trends, real-time personalization, failure detection, spam detection, advertising statistics,

- Challenges
  - Computational high-level frameworks for distributed systems
  - Machine failures and stragglers (slow nodes) cause latency issues
  - Need to be able to recover quickly from faults

- Performance goals
  - Scalability to 100s of nodes, with low cost
  - Second-scale latency (fundamental limitation of micro-batching)
  - Second-scale recovery from faults and stragglers
Aurora (research), Storm, etc., followed a continuous operator model

- Better latencies and faster response -- but fault tolerance is tricky
- Especially in a distributed setting
- Can’t handle stragglers easily -- may block the whole system

(a) Continuous operator processing model. Each node continuously receives records, updates internal state, and emits new records. Fault tolerance is typically achieved through replication, using a synchronization protocol like Flux or DPC [34, 5] to ensure that replicas of each node see records in the same order (e.g., when they have multiple parent nodes).

(b) D-Stream processing model. In each time interval, the records that arrive are stored reliably across the cluster to form an immutable, partitioned dataset. This is then processed via deterministic parallel operations to compute other distributed datasets that represent program output or state to pass to the next interval. Each series of datasets forms one D-Stream.
D-Streams Processing Model

- Input data stream broken into batches (i.e., a form of “sliding windows”)
- Each batch seen as an RDD (resilient distributed dataset)
- Standard Spark operations done on RDDs

Create batches of 1s each
Compute counts for each URL

```java
pageViews = readStream("http://...", "1s")
one = pageViews.map(event => (event.url, 1))
counts = ones.runningReduce((a, b) => a + b)
```

Each operation creates a new D-Stream
Internally stored as a collection of RDDs

Figure 2: High-level overview of the Spark Streaming system. Spark Streaming divides input data streams into batches and stores them in Spark’s memory. It then executes a streaming application by generating Spark jobs to process the batches.
D-Streams Fault Tolerance

- Builds upon the fault tolerance of RDDs, through use of “lineage” graphs
- Additional “checkpointing” to limit recomputations

Figure 3: Lineage graph for RDDs in the view count program. Each oval is an RDD, with partitions shown as circles. Each sequence of RDDs is a D-Stream.
Timing/Timestamps Issues

- Implicit vs explicit timestamps
  - Implicit timestamps usually assigned when the record enters the system
  - Explicit timestamps usually assigned at the source

- Because of network delays, windows on explicit timestamps are difficult

- Options for doing windows on explicit timestamps
  - Wait for a limited “slack time” to see if more records arrive with the timestamp in a window
  - Issue “corrections” when older records arrive
    - The downstream operators need to be able to understand these corrections
D-Stream API

- `words.window("5s")`: Create a D-Stream with sliding windows of 5s
- Incremental aggregation
  - Variants of "reduceByWindow" operation

Figure 4: `reduceByWindow` execution for the associative-only and associative+invertible versions of the operator. Both versions compute a per-interval count only once, but the second avoids re-summing each window. Boxes denote RDDs, while arrows show the operations used to compute window \([t, t+5]\).
D-Stream API

- State tracking: Transform a stream of (key, event) into a stream of (key, state) records
- Doesn’t appear to be there in the latest D-Stream API

sessions = events.track(
    (key, ev) => 1,       // initialize function
    (key, st, ev) =>     // update function
        ev == Exit ? null : 1,
    "30s")            // timeout

counts = sessions.count() // a stream of ints

Figure 5: RDDs created by the track operation.
Other Issues

- Clean consistency semantics
  - Modulo the timestamp issues mentioned earlier
- Unification of batch and interactive processing

<table>
<thead>
<tr>
<th>Aspect</th>
<th>D-Streams</th>
<th>Continuous proc. systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>0.5–2 s</td>
<td>1–100 ms unless records are batched for consistency</td>
</tr>
<tr>
<td>Consistency</td>
<td>Records processed atomically with interval they arrive in</td>
<td>Some systems wait a short time to sync operators before proceeding [5, 33]</td>
</tr>
<tr>
<td>Late records</td>
<td>Slack time or app-level correction</td>
<td>Slack time, out of order processing [23, 36]</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fast parallel recovery</td>
<td>Replication or serial recovery on one node</td>
</tr>
<tr>
<td>Straggler recovery</td>
<td>Possible via speculative execution</td>
<td>Typically not handled</td>
</tr>
<tr>
<td>Mixing w/ batch</td>
<td>Simple unification through RDD APIs</td>
<td>In some DBs [15]; not in message queueing systems</td>
</tr>
</tbody>
</table>

Table 1: Comparing D-Streams with record-at-a-time systems.
Implementation Details

- Master: tracks the D-Stream lineage graphs and schedules
- All state is in the standard Spark RDDs

Figure 6: Components of Spark Streaming, showing what we added and modified over Spark.
Implementation Details

- **Parallel Recovery**
  - If a node fails, RDDs are recomputed using lineage graphs

- **Straggler mitigation**
  - Can use speculative backup copies
  - Determinism allows for exploring different options

- **Master recovery**
  - In standard Spark, "master" ("driver") is not fault-tolerant
  - Use checkpointing at the master to recover
  - Some issues with output operations (i.e., how to guarantee that outputs are not repeated)