CMSC 724: Database Management Systems
Data Streams and Dataflow Engines

Instructor: Amol Deshpande
amol@cs.umd.edu

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

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

Related Topics

- 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

View Maintenance

- ”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

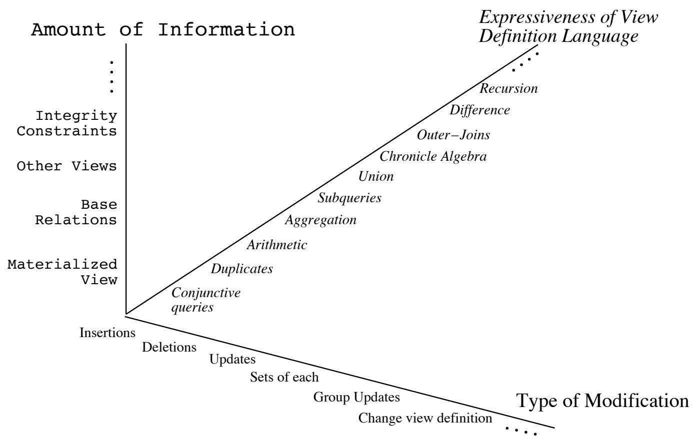


Figure 1: The problem space

Example 1

- Relation: part(part_no, part_cost, contract)
- View:

$$\text{expensive\_parts}(\text{part\_no}) = \Pi_{\text{part\_no}}\, \sigma_{\text{part\_cost}>1000}(\text{part})$$

- 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 → insert p1 into the view

Example 1

- Relation: part(part_no, part_cost, contract)
- View:

$$\texttt{expensive\_parts}(\texttt{part\_no}) = \Pi_{\texttt{part\_no}}\,\sigma_{\texttt{part\_cost}>1000}(\texttt{part})$$

- 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:

$$supp\_parts(part\_no) = \Pi_{part\_no}(supp \bowtie_{part\_no} part)$$

- 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: link(S, D)
- View to define one-hop neighbors

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

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

$$\begin{aligned}\Delta(\texttt{hop}) = \Pi_{X,Y}((\Delta(\texttt{link})(X,V) &\bowtie_{V=W} \texttt{link}(W,Y)) \cup \\ (\texttt{link}(X,V) &\bowtie_{V=W} \Delta(\texttt{link})(W,Y)) \cup \\ (\Delta(\texttt{link})(X,V) &\bowtie_{V=W} \Delta(\texttt{link})(W,Y)))\end{aligned}$$

Full Information: Non-recursive Views

- 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
  - link = {(a, b), (b, c), (b, e), (a, d), (d, c)}
  - hop = {(a, c), (a, e)}
    - Counts maintained internally: 2 and 1 resp.
  - Say, (a, b) is deleted from link
    - The deletion ”counts”: (a, c) → -1, (a, e) → -1
    - So we remove (a, e), but keep (a, c)
- General idea extensible to other types of queries, including joins

Full Information

- 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

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

Data Stream Model

- 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

$$
\begin{array}{lll}
Q_1: & \text{SELECT} & \texttt{notifyoperator}(\text{sum}(\text{len})) \\
 & \text{FROM} & B \\
 & \text{GROUP BY} & \texttt{getminute}(\text{time}) \\
 & \text{HAVING} & \text{sum}(\text{len}) > \; t
\end{array}
$$

  - 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

```
Q2: SELECT    flowid, src, dest, sum(len) AS flowlen
    FROM      (SELECT     src, dest, len, time
               FROM       B
               ORDER BY   time )
    GROUP BY  src, dest, getflowid(src, dest, time)
              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

$$
\begin{array}{lll}
Q_3\text{:} & (\text{SELECT} & \text{count }(*) \\
& \quad\text{FROM} & \text{C, B} \\
& \quad\text{WHERE} & \text{C.src} = \text{B.src and C.dest} = \text{B.dest} \\
& & \text{and C.id} = \text{B.id}) \;/ \\
& (\text{SELECT count }(*) \text{ FROM } B) &
\end{array}
$$

  - Potentially very large intermediate tables

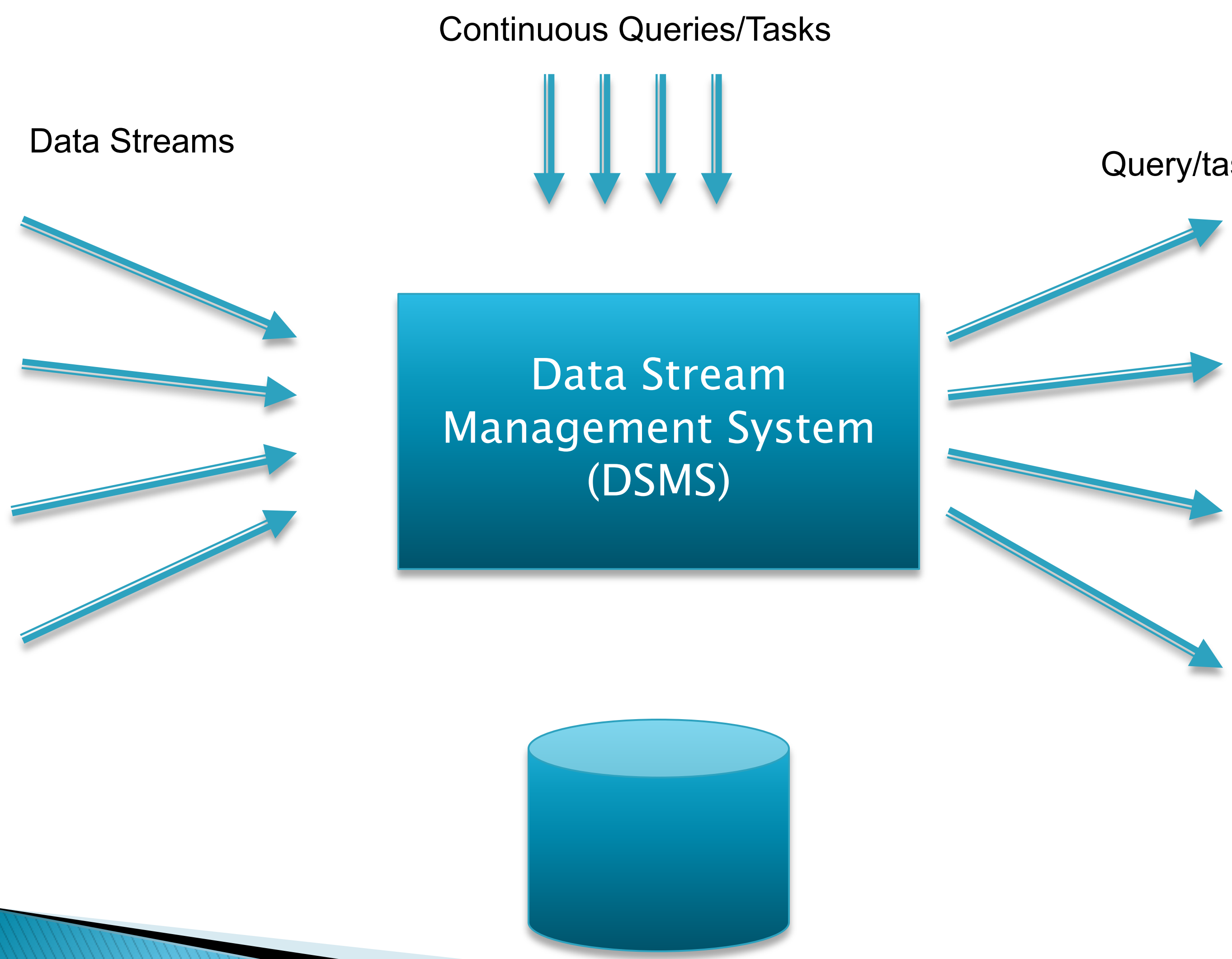
Network Traffic Management

- 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: WITH Load AS

```
    (SELECT     src, dest, sum(len) AS traffic
     FROM       B
     GROUP BY   src, dest)
SELECT      src, dest, traffic
FROM        Load AS L1
WHERE       (SELECT   count(*)
             FROM     Load AS L2
             WHERE    L2.traffic < L1.traffic) >
            (SELECT   0.95×count(*) FROM Load)
ORDER BY    traffic
```

Querying over Data Streams: Simple Architecture



Querying over Data Streams: Language

- 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 → B within 10 seconds → 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
  - updateSummary() given a new tuple
  - 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 → 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

Querying Issues: Queries referencing Past Data

- 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`.

```
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 10 long-distance calls made by each customer

```
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

```
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

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

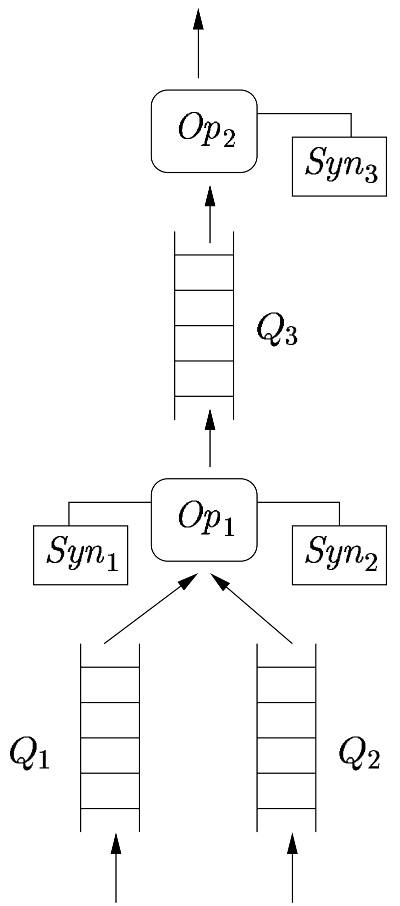
Stanford STREAM System: Timestamps

- 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*

Stanford STREAM System: Query Processing

- 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: x1, x2, …, x_N, …
  - Each value seen only once, in that order
  - At all times N, compute function: f(x1, …, 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 <= 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
  - $m1 = 3, m2 = m3 = 2, m4 = m5 = 1.$
- Frequency moment $F_k = \Sigma\, m_i^k$
  - $F0$ = 5 = number of distinct elements in the stream
  - $F1$ = 9 = total number of elements in the stream (trivial)
  - $F2$ = 19 = comes in up many places (e.g. self-join size of a relation)
  - F_∞ = most frequent item's multiplicity
- Flajolet-Martin 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: SUM x_i * 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_∞ 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

Outline

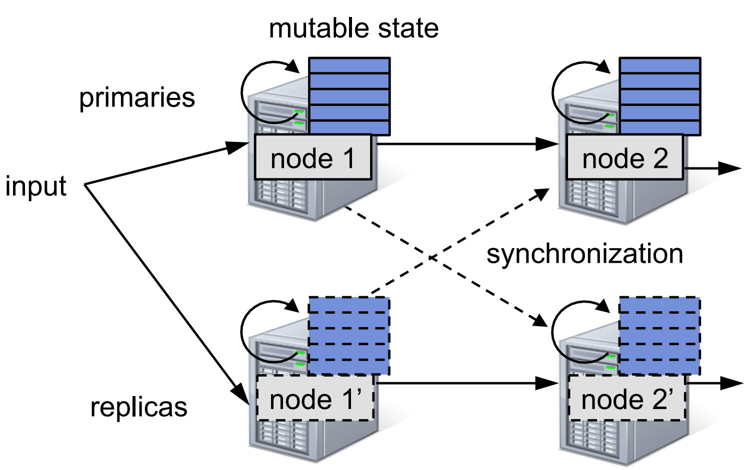
- Overview and Early Work
- Maintenance of Materialized Views
- Models and issues in data stream systems
- Discretized streams: fault-tolerant streaming computation at scale
- Apache Flink: Stream and Batch Processing in a Single Engine
- Incremental, Iterative Data Processing with Timely Dataflow
-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

Motivation

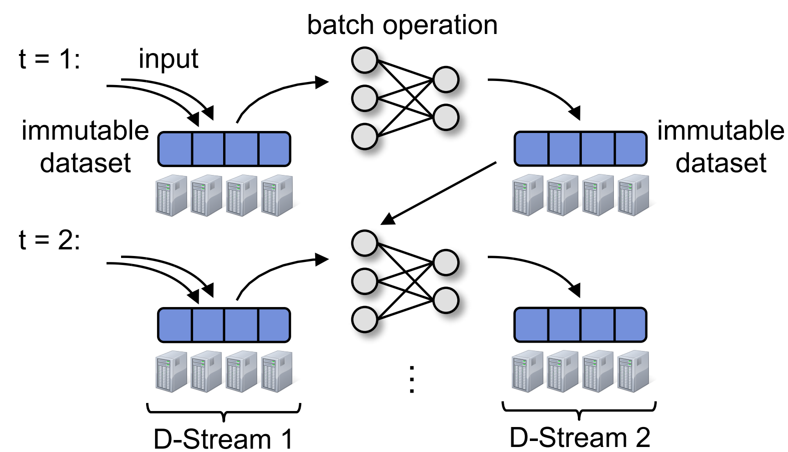
- Need streaming computation models analogous to MapReduce for batch processing
  - Applications: detecting trends, real-time personalization, failure detection, spam detection, advertising statistics,
- Challenges
  - Computational high-level frameworks for distributed systems
  - Machine failures and 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

Processing Models

- 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

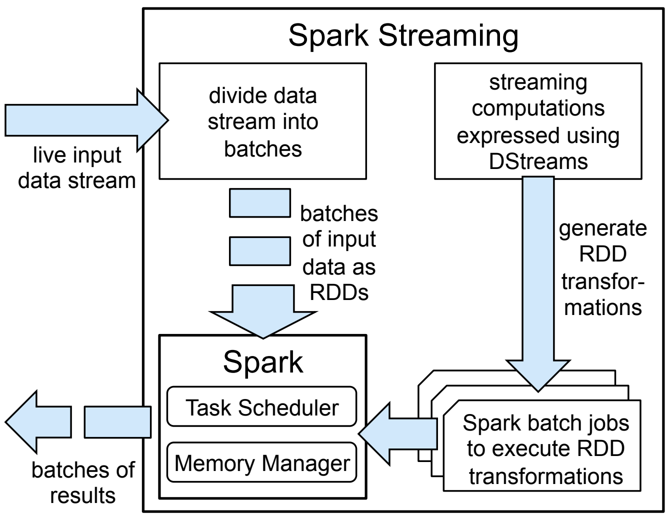


Figure 2: High-level overview of the Spark Streaming system. Spark Streaming divides input data streams into batches and stores them in Spark’s memory. It then executes a streaming application by generating Spark jobs to process the batches.

Create batches of 1s each
Compute counts for each URL

```
pageViews = readStream("http://...", "1s")
ones = 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

D-Streams Fault Tolerance

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

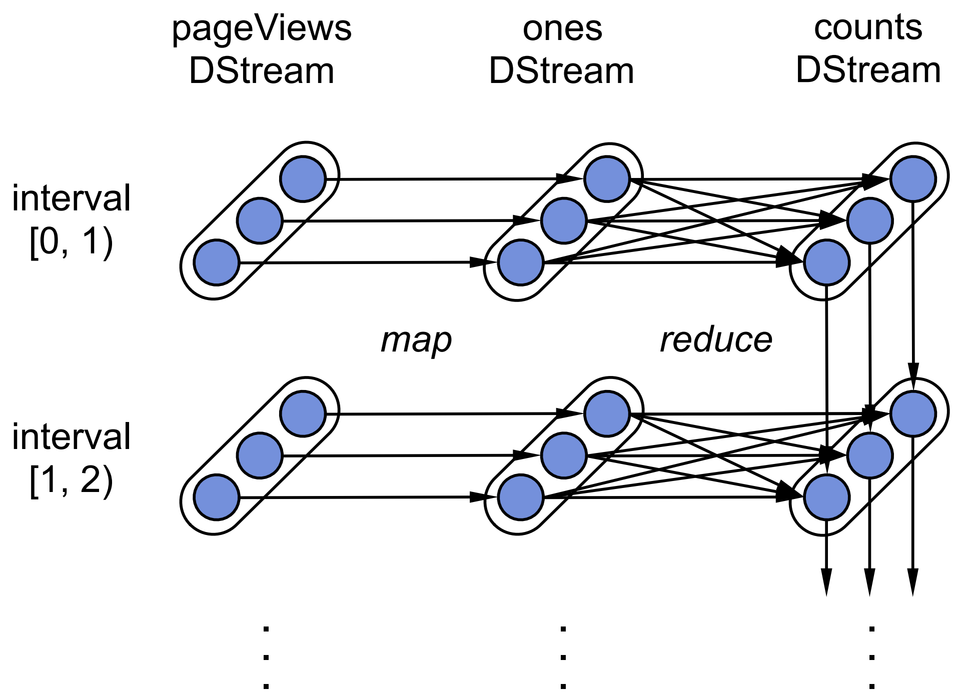


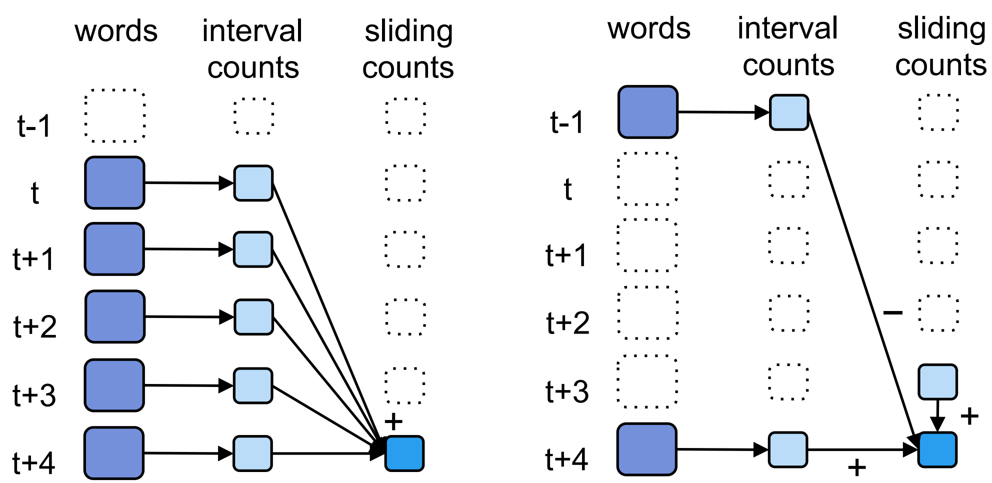
Figure 3: Lineage graph for RDDs in the view count program. Each oval is an RDD, with partitions shown as circles. Each sequence of RDDs is a D-Stream.

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



(a) Associative only (b) Associative & invertible

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
```

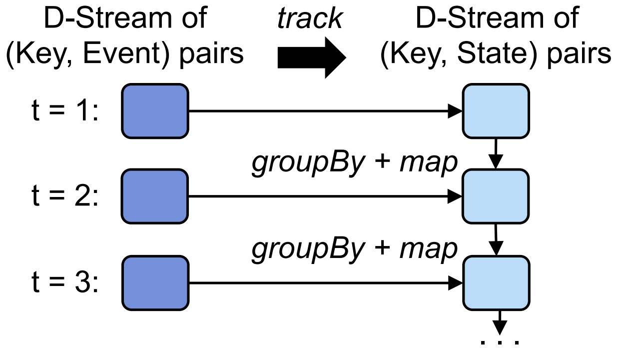


Figure 5: RDDs created by the *track* operation.

Other Issues

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

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

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

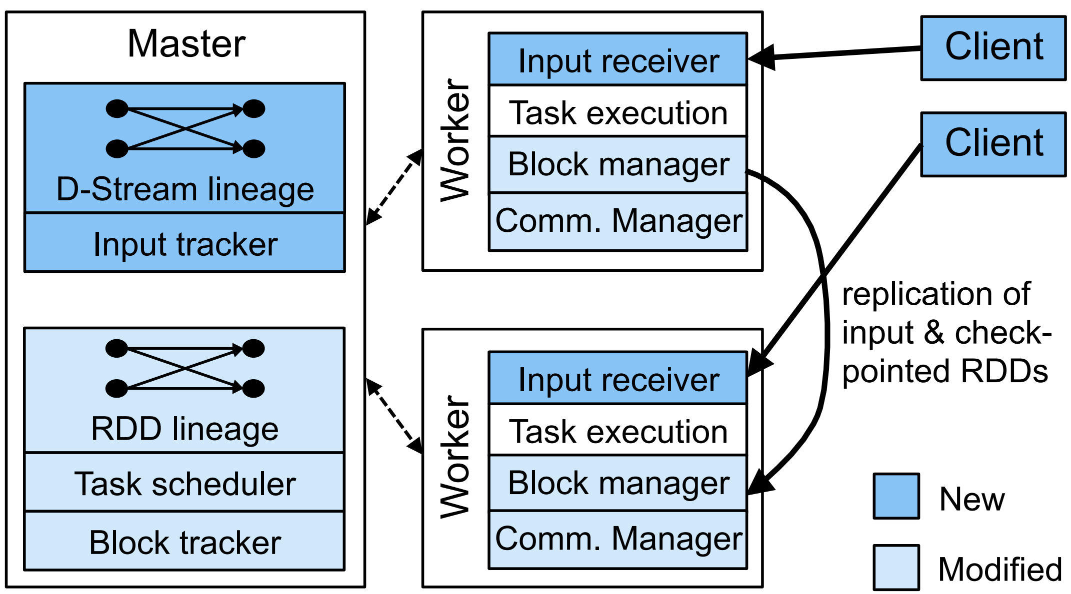


Figure 6: Components of Spark Streaming, showing what we added and modified over Spark.

Implementation Details

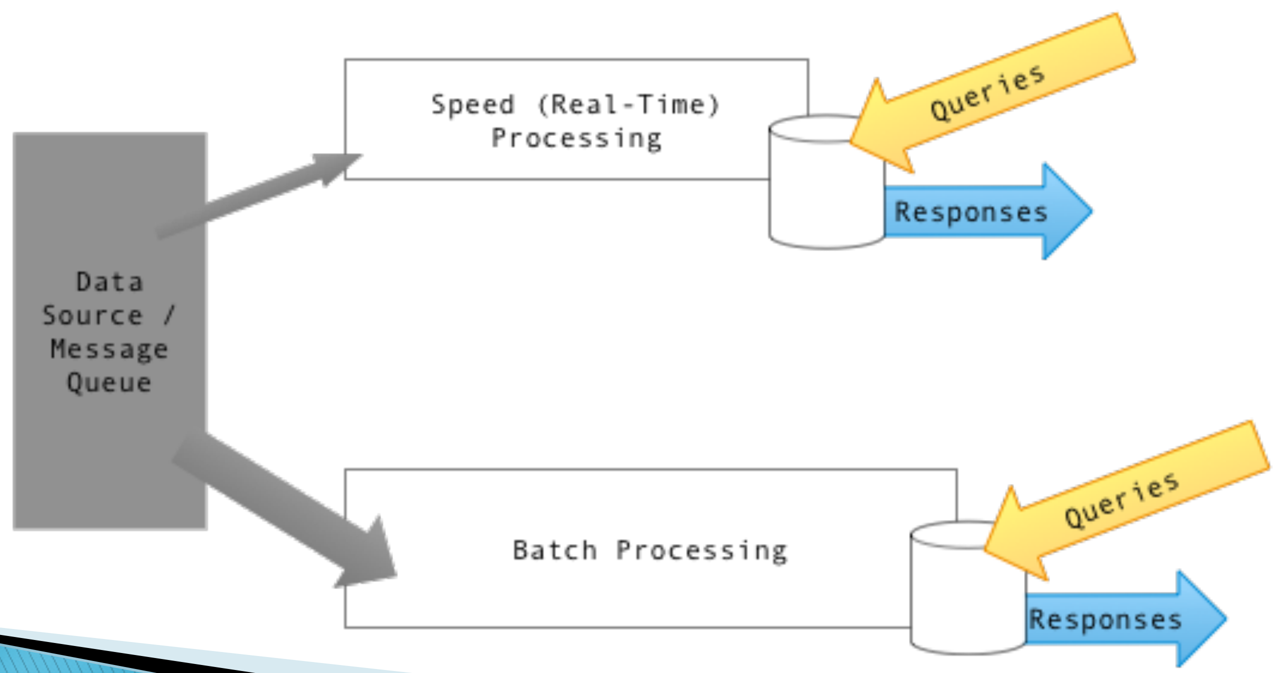
- Parallel Recovery
  - If a node fails, RDDs are recomputed using lineage graphs
- Straggler mitigation
  - Can use speculative backup copies
  - Determinism allows for a exploring different options
- Master recovery
  - In standard Spark, ”master” (”driver”) is not fault-tolerant
  - Use checkpointing at the master to recover
  - Some issues with output operations (i.e., how to guarantee that outputs are not repeated)

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

Motivation

- Large-scale data processing primarily batch-oriented
- Patterns like “Lambda Architecture” still have high latencies
- Need a unifying architecture/system that supports both batch processing and stream processing



Flink Architecture

- Two APIs supported by the same engine

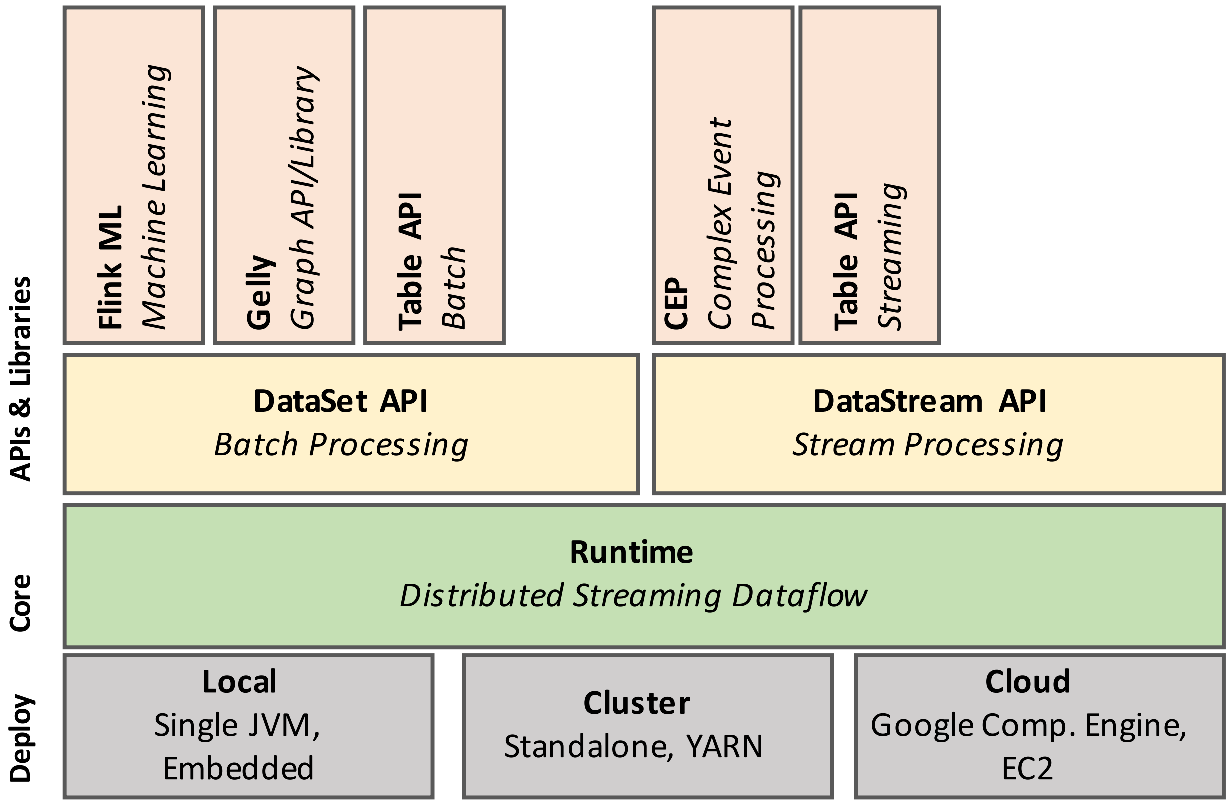


Figure 1: The Flink software stack.

Flink Process Model

- Client: program code → dataflow graph ("driver" in Spark)
- Job Manager: Coordinate execution
- Task Managers: Workers

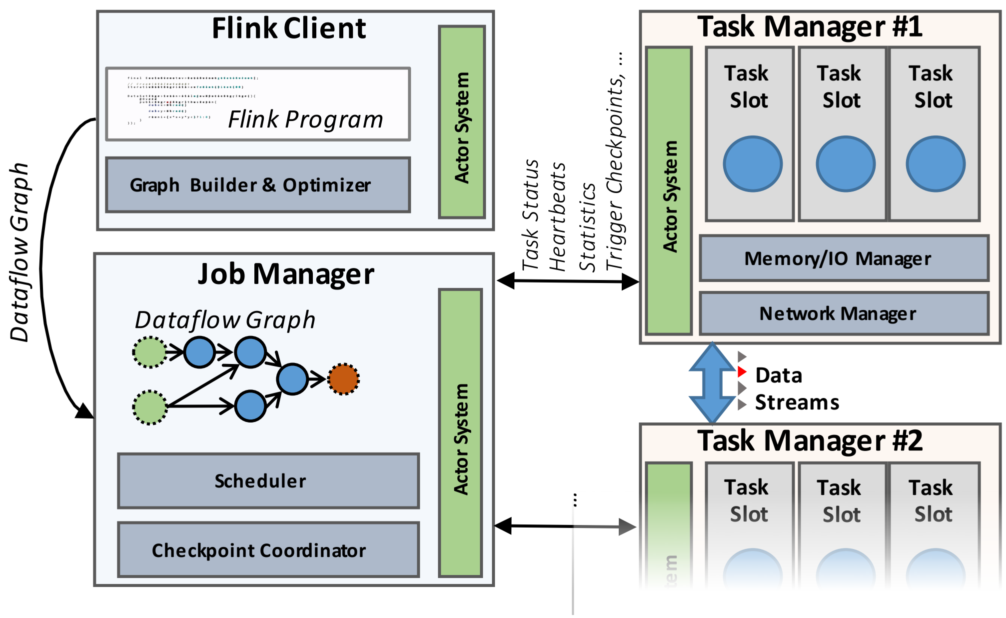


Figure 2: The Flink process model.

Streaming Dataflows Abstraction

- Directed Acyclic Graphs: Stateful Operators + Streams between them
  - Similar to a typical database query engine
  - Each operator may be partitioned across machines → streams are partitioned across machines
  - Data may need to be “shuffled” across machines (e.g., for a groupby)

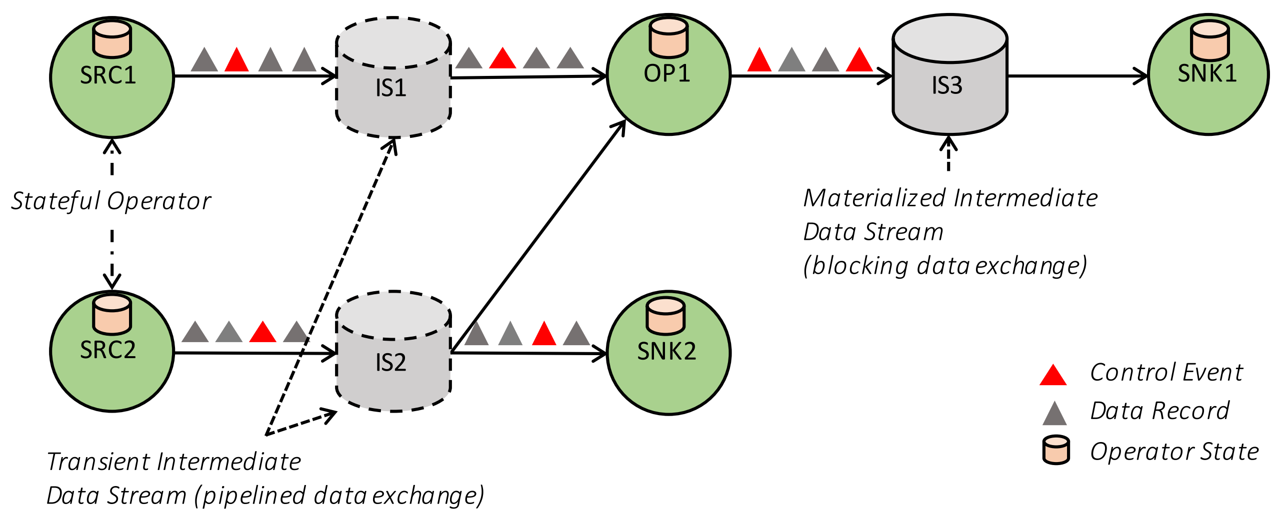


Figure 3: A simple dataflow graph.

Data Exchange

- Pipelined streams
  - Both consumer and producer running simultaneously
  - Helps avoid “materialization”
- …vs Blocking streams
  - Need to write out intermediate results somewhere (potentially on disks)
- Use “buffers” to send data around -- configurable buffer size and timeouts
  - Large buffers → high latencies, higher throughputs
  - Large timeouts → high latencies, higher throughputs

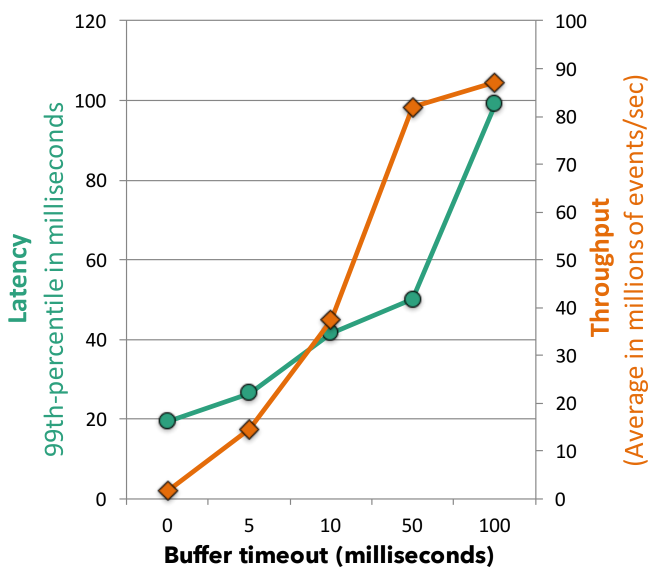


Figure 4: The effect of buffer-timeout in latency and throughput.

Control Events

- Special events designed to control different aspects
- Examples:
  - Checkpoint barriers: coordinate checkpoints
  - Watermarks: information about "event-time" (also see "Timely Dataflow")
  - Iteration barriers: for bulk-synchronous processing
- Control events need to be delivered "in order"
  - Okay for unary operators that work on a single stream
  - But not for operators that have more than one input (including different partitions of the same stream)
    - Left to the operator logic to deal with this

Fault Tolerance

- Two Issues
  - Exactly-once-processing semantics
  - Recovery from failures
- Checkpointing and Partial re-execution
  - Take snapshots on a periodic basis of the operator state, to bound recovery time
  - If a machine goes down, can replay from the last checkpoint
  - Assumes that the input sources are “replayable”
- Challenge
  - Taking a snapshot without a complete halt of the operators

Asynchronous Barrier Snapshotting

- Insert “barrier” control events into the stream
- Each operator aligns across all of its input, and then takes snapshot, and then forwards the barrier to its downstream
- Only need to checkpoint the operator state
  - And not the streams/queues

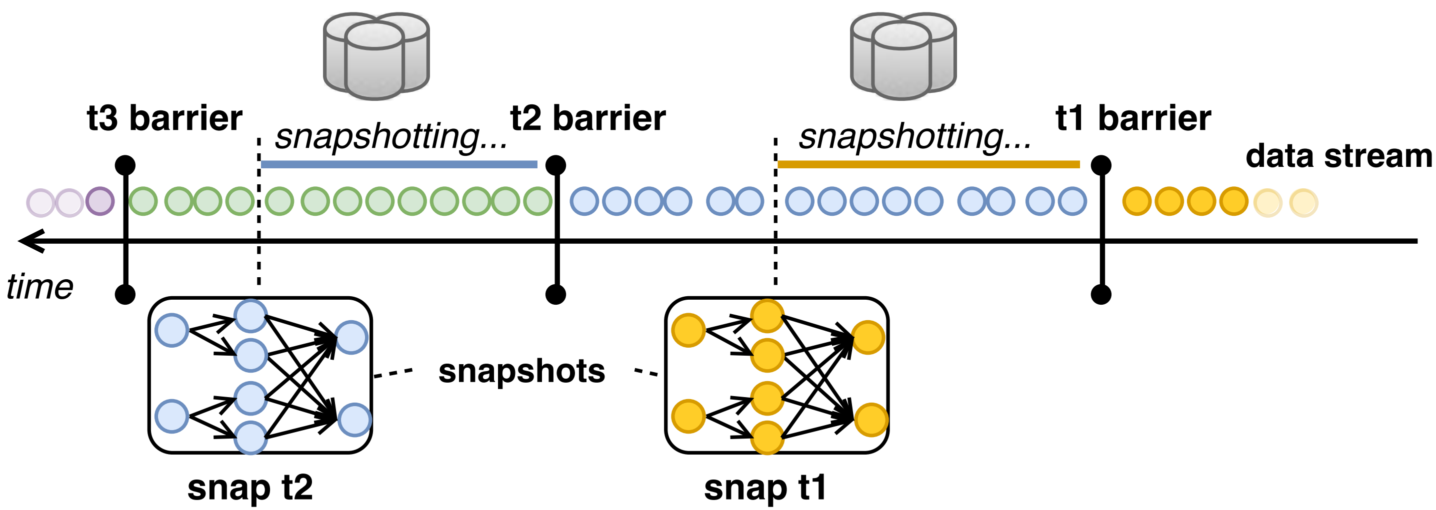


Figure 5: Asynchronous Barrier Snapshotting.

Iterative Dataflows

- Iterations needed for ML/Graph Analytics
- Supported through explicit mechanism and a feedback loop

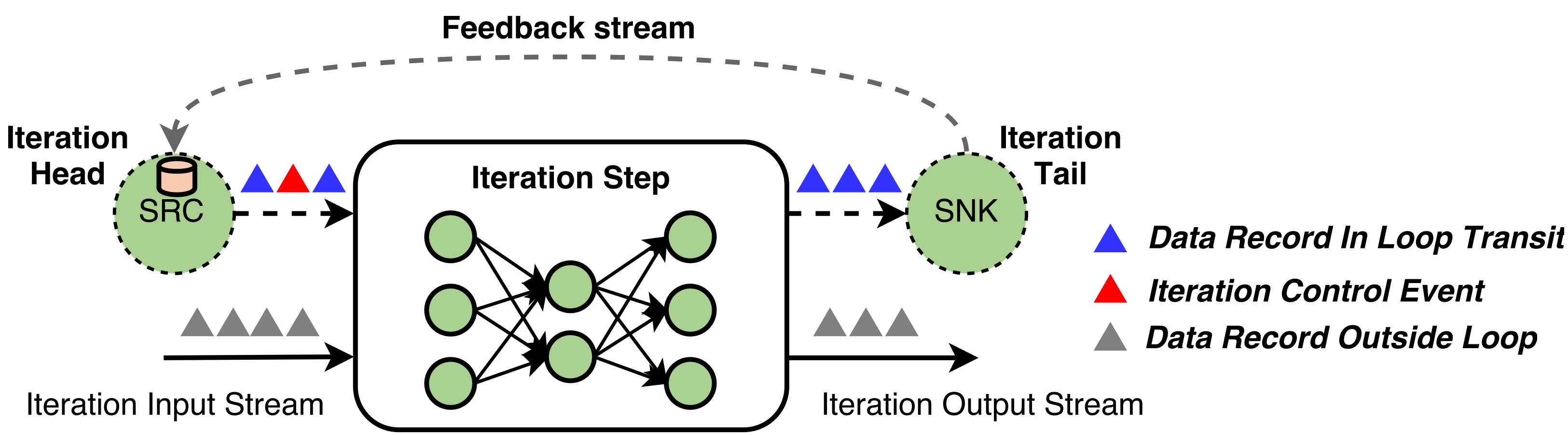


Figure 6: The iteration model of Apache Flink.

Stream Analytics

- Event-time vs processing-time (implicit vs explicit timestamps)
  - Deal with ”skew” with special watermark events
  - Data sources insert ”time” watermarks into the streams
  - Windowing operators reason about these watermarks to decide if the windows are “complete”
  - Using “processing-time” leads to lower latencies, but potentially inconsistent execution
- Also has a notion of “ingestion-time” (special case of event-time)
  - In-between event-time and processing-time
  - Leads to more consistent execution

Stream Analytics

- Stateful Processing
  - Operator state made explicit in the APIs
  - Checkpointing mechanism automatically handles such state
- Stream Windows
  - Flexible interface to define new types of windows
  - Supports standard sliding-time windows, etc.

```
stream
  .window(SlidingTimeWindows.of(Time.of(6, SECONDS), Time.of(2, SECONDS))
  .trigger(EventTimeTrigger.create())
```

```
stream
  .window(GlobalWindow.create())
  .trigger(Count.of(1000))
  .evict(Count.of(100))
```

Batch Analytics

- Simplified API, but the same dataflow engine
- Periodic snapshotting turned off -- can replay from the beginning
- Query Optimization
  - Incorporates some of the standard query optimization methods, primarily focusing on data movement
  - Operators are black-boxes -- hard to optimize
- Batch Iterations
  - Recall PageRank in GraphX
  - Can be done through use of control events, and support for iterations

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

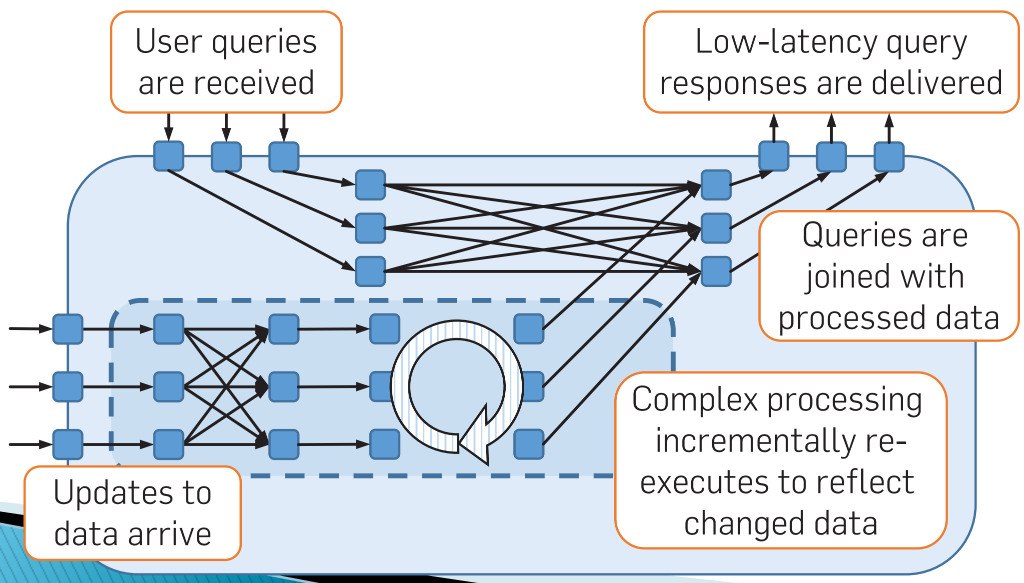
Motivation

- Common abstraction to support many different use cases, including low latency, high throughput, and iterative computations
- Challenges:
  - Asynchronous, distributed, tuple-at-a-time computation → low latencies, but consistency issues
  - Carefully coordinated execution → high throughputs, but high latencies
- Key ideas:
  - Timely dataflow, through use of "virtual timestamps"
  - Fault tolerance through checkpointing
- Implemented in the NAIAD system

Example Application

- Separate tasks would have been done using three different systems
  - Clustering algorithm to generate a model: using MapReduce/Spark
  - Handling recent updates: MillWheel/Storm/Flink, but approximate only
  - Queries: Another application that looks up user queries against the model

Figure 1. An application that supports real-time queries on continually updated data. The dashed rectangle represents iterative processing that incrementally updates as new data arrive.



Dataflow

- Starting point: Standard Stateful Dataflow Model
  - Represent computation as a DAG
  - Each node can maintain arbitrary state, private to the node
  - Support low latencies, and iterations (through cycles)
- Single-threaded node implementation
  - Simplifies implementation and optimization
- Data Parallelism
  - Split records across parallel subtasks based on a key

“Timely” Dataflow

- Main difference: use of logical timestamps

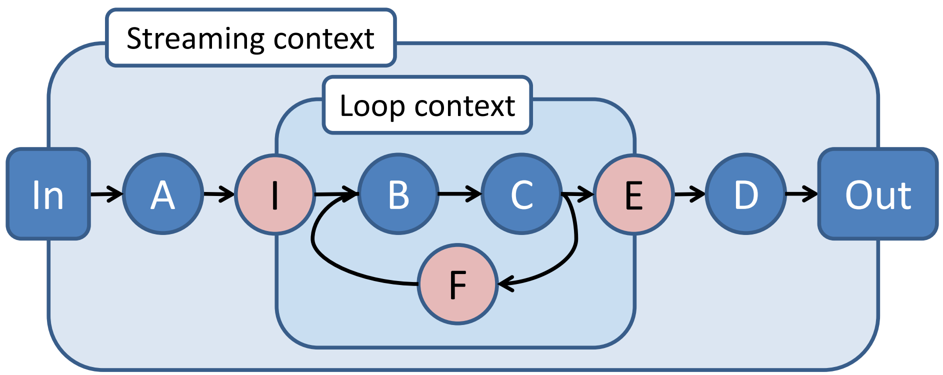


Figure 3: This simple timely dataflow graph (§2.1) shows how a loop context nests within the top-level streaming context.

External producer: labels each message with an “epoch”
inserts control messages saying “no more tuples from an epoch”

From SOSP 13 Paper

“Timely” Dataflow

- Nested Loops, each with a “feedback” vertex (F)
- Every message has a logical timestamp:

$$\text{Timestamp} : (\overbrace{e \in \mathbb{N}}^{\text{epoch}}, \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$$

- For each loop, ingress, egress, and feedback vertices modify timestamps

Vertex	Input timestamp	Output timestamp
Ingress	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k, 0 \rangle)$
Egress	$(e, \langle c_1, \ldots, c_k, c_{k+1} \rangle)$	$(e, \langle c_1, \ldots, c_k \rangle)$
Feedback	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k + 1 \rangle)$

Computation at Each Node/Vertex

- Each vertex implements two callbacks

$$v.\textsc{OnRecv}(e : \text{Edge},\ m : \text{Message},\ t : \text{Timestamp})$$
$$v.\textsc{OnNotify}(t : \text{Timestamp}).$$

- And may use two methods:

$$\mathit{this}.\textsc{SendBy}(e : \text{Edge},\ m : \text{Message},\ t : \text{Timestamp})$$
$$\mathit{this}.\textsc{NotifyAt}(t : \text{Timestamp}).$$

- Requires a guarantee that:
  - After invoking v.OnNotify(t), no calls will be made to: v.OnRecv(e, m, t'<=t)

Example Vertex Implementation

```
class DistinctCount<S,T> : Vertex<T>
{
  Dictionary<T, Dictionary<S,int>> counts;
  void OnRecv(Edge e, S msg, T time)
  {
    if (!counts.ContainsKey(time)) {
      counts[time] = new Dictionary<S,int>();
      this.NotifyAt(time);
    }

    if (!counts[time].ContainsKey(msg)) {
      counts[time][msg] = 0;
      this.SendBy(output1, msg, time);
    }

    counts[time][msg]++;
  }

  void OnNotify(T time)
  {
    foreach (var pair in counts[time])
      this.SendBy(output2, pair, time);
    counts.Remove(time);
  }
}
```

Figure 4: An example vertex with one input and two outputs, producing the distinct input records on `output1`, and a count for each one on `output2`. The distinct records may be sent as soon as they are seen, but the counts must wait until all records bearing that time have been received.

From SOSP 13 Paper

Achieving Timely Dataflow

- Question: How to “correctly” invoke the “onNotify()” callbacks?
  - i.e., how do we know that no more events with time “t” will be generated?
  - In a distributed setting, by any node?
- Dataflow graph puts constraints on the timestamps that could be generated in future
- Combine that with a count of how many unprocessed events are still there
- Implemented as an “out-of-band” mechanism
  - Unlike Flink, which uses special control messages sent in the same fashion as normal messages
  - Lower overheads in using “out-of-band”, especially #messages that are needed

Implementation

- Example Naiad Program

```
// 1a. Define input stages for the dataflow.
var input = controller.NewInput<string>();

// 1b. Define the timely dataflow graph.
// Here, we use LINQ to implement MapReduce.
var result = input.SelectMany(y => map(y))
                  .GroupBy(y => key(y),
                     (k, vs) => reduce(k, vs));

// 1c. Define output callbacks for each epoch
result.Subscribe(result => { ... });

// 2. Supply input data to the query.
input.OnNext(/* 1st epoch data */);
input.OnNext(/* 2nd epoch data */);
input.OnNext(/* 3rd epoch data */);
input.OnCompleted();
```

From SOSP 13 Paper

Implementation

- Uses deferred (lazy) evaluation
- Same program can be run on one machine, or multiple machines
  - Uses shared memory for local workers, and TCP for remote workers
- Run-time code generation, value types, to reduce overheads
- Layered programming abstractions
  - Raw API in addition to framework libraries

From SOSP 13 Paper

Graph Processing in Naiad

- Iterative graph algorithms difficult to do in a distributed fashion
  - Map-Reduce not a good abstraction for programming
  - Requires too much communication across the nodes (graphs are hard to partition)
- Gather-Apply-Scatter Abstraction (GraphX)
  - Gather: values from neighbors
  - Apply: an update to the node's state
  - Scatter: new value to the neighbors
- Can be done as a Join followed by an Aggregate in MR (or SQL)
  - Pagerank is 9 lines of code in GraphLINQ (not much longer in Spark RDD either though)

Figure 3. Illustration of a graph algorithm as a timely dataflow graph.

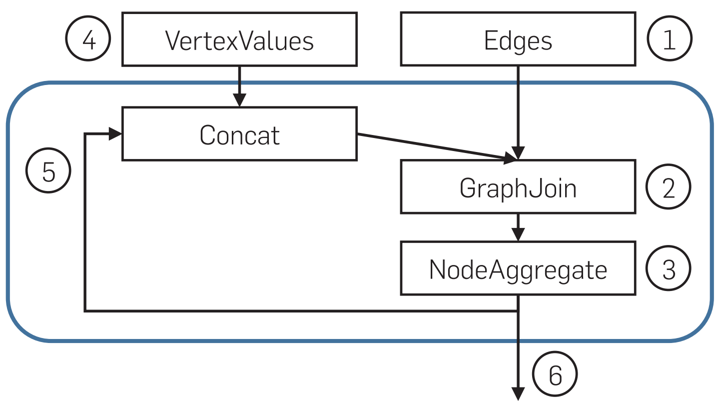
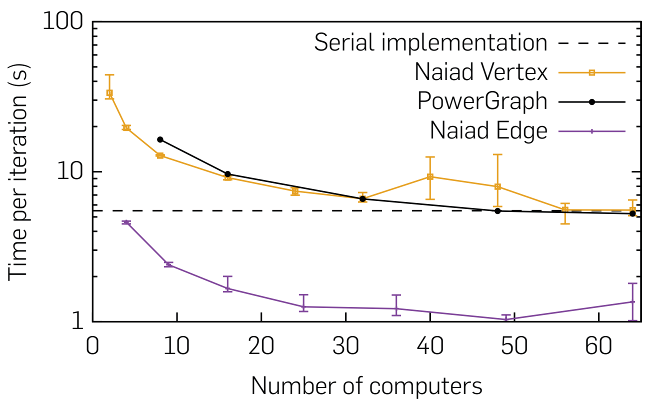


Figure 4. Time per iteration for PageRank on the Twitter follower graph, as the number of machines is varied.



Differential Dataflow

- Goal: Incremental update of the iterative computation
  - Generalization of incremental view maintenance
- Can be handled through partially ordered timestamps
  - Timestamps of the type: (epoch_no, iteration_nos)
- Same framework for both iterations and incremental changes → higher composability

Fault Tolerance

- Harder because of asynchrony and non-determinism
- Uses global checkpoints, but details still to be worked out
  - (Not sure if there was followup work)

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

Motivation

- Very high data volumes: > 12M events per second
  - Mostly machine generated: IoT, Sensors, Machine logs, etc.
- Dataflow processing engines leave the development of analytics/operators to the users
- Macrobase :
  - A layer on top of dataflow engines to support classification and explanation
  - … to prioritize attention to important events in fast data
  - Uses sampling and sketching techniques to handle the scale

Architecture/APIs

- Two core classes of operators
  - Classification: Label individual data points
  - Explanation: Group and aggregate multiple data points
- All operators must operate on streams

DATA TYPES	
Point := (array<double> metrics, array<varchar> attributes)	
Explanation := (array<varchar> attributes, *stats* statistics)	
OPERATOR INTERFACE	
Operator	*Type Signature*
Ingestor	external data source(s) → stream<*Point*>
Transformer	stream<*Point*> → stream<*Point*>
Classifier	stream<*Point*> → stream<(label, *Point*)>
Explainer	stream<(label, *Point*)> → stream<*Explanation*>
Pipeline	Ingestor → stream<*Explanation*>

Table 1: MacroBase's core data and operator types. Each operator implements a strongly typed, stream-oriented dataflow interface specific to a given pipeline stage. A pipeline can utilize multiple operators of each type via transformations, such as group-by and one-to-many stream replication, as long as the pipeline ultimately returns a single stream of explanations.

Architecture/APIs

- Extensibility Options
  - New domain-specific feature transformation in the beginning
  - New rules and/or labels for the supervised classification
  - Users can write their own operators, as well as new pipelines

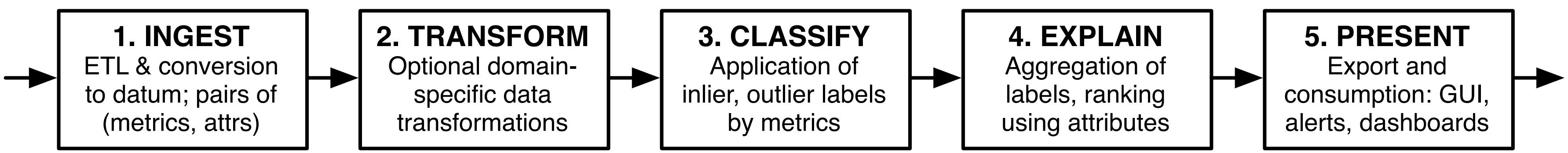


Figure 1: MacroBase's default analytics pipeline: MacroBase ingests streaming data as a series of points, which are scored and classified, aggregated by an explanation operator, then ranked and presented to end users.

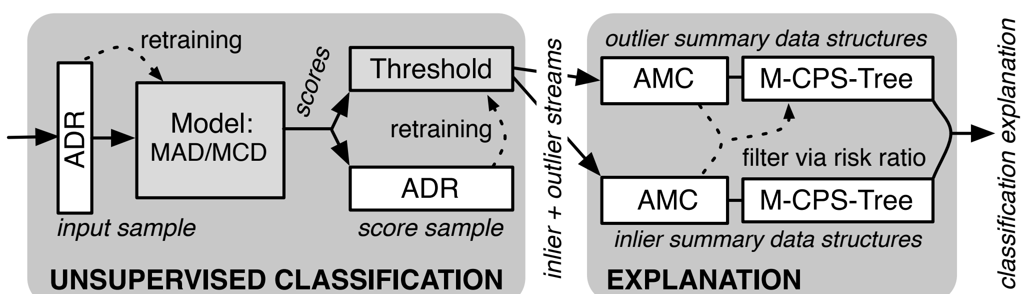


Figure 2: MDP: MacroBase's default streaming classification (Section 4) and explanation (Section 5) operators.

Details

- Classification
  - Main technique: Unsupervised density-based classification
  - New sampling technique (Adaptable Damped Reservoir)
  - Weights more recent points more heavily than older points
- Explanation
  - Use "risk ratios" to identify attributes that are common to outliers

$$\text{risk ratio} = \frac{a_o/(a_o+a_i)}{b_o/(b_o+b_i)}$$

Algorithm 2 MDP's Outlier-Aware Explanation Strategy

given: minimum risk ratio r, minimum support s, set of outliers O, set of inliers I

1: find attributes w/ support $\geq s$ in O and risk ratio $\geq r$ in O, I
2: mine FP-tree over O using only attributes from (1)
3: filter (2) by removing patterns w/ risk ratio $< r$ in I; return