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

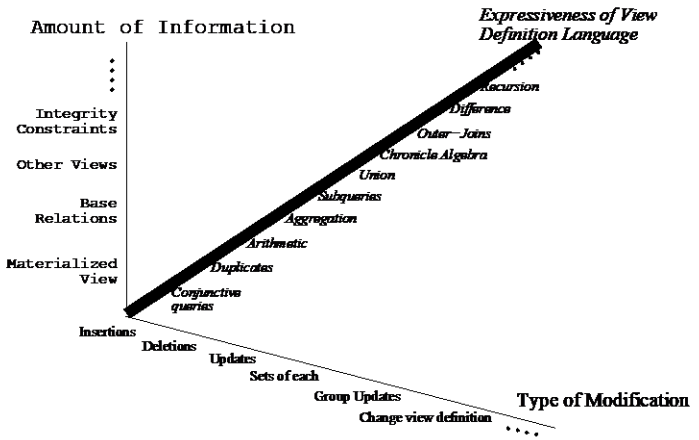


Figure 1: The problem space

Example 1

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

$$expensive_parts(part_no) = \Pi_{part_no}\, \sigma_{part_cost>1000}(part)$$

- 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:

$$expensive_parts(part_no) = \Pi_{part_no}\, \sigma_{part_cost>1000}(part)$$

- 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

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

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

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

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

```
Q1: SELECT     notifyoperator(sum(len))
    FROM       B
    GROUP BY   getminute(time)
    HAVING     sum(len) > l
```

  - 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

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

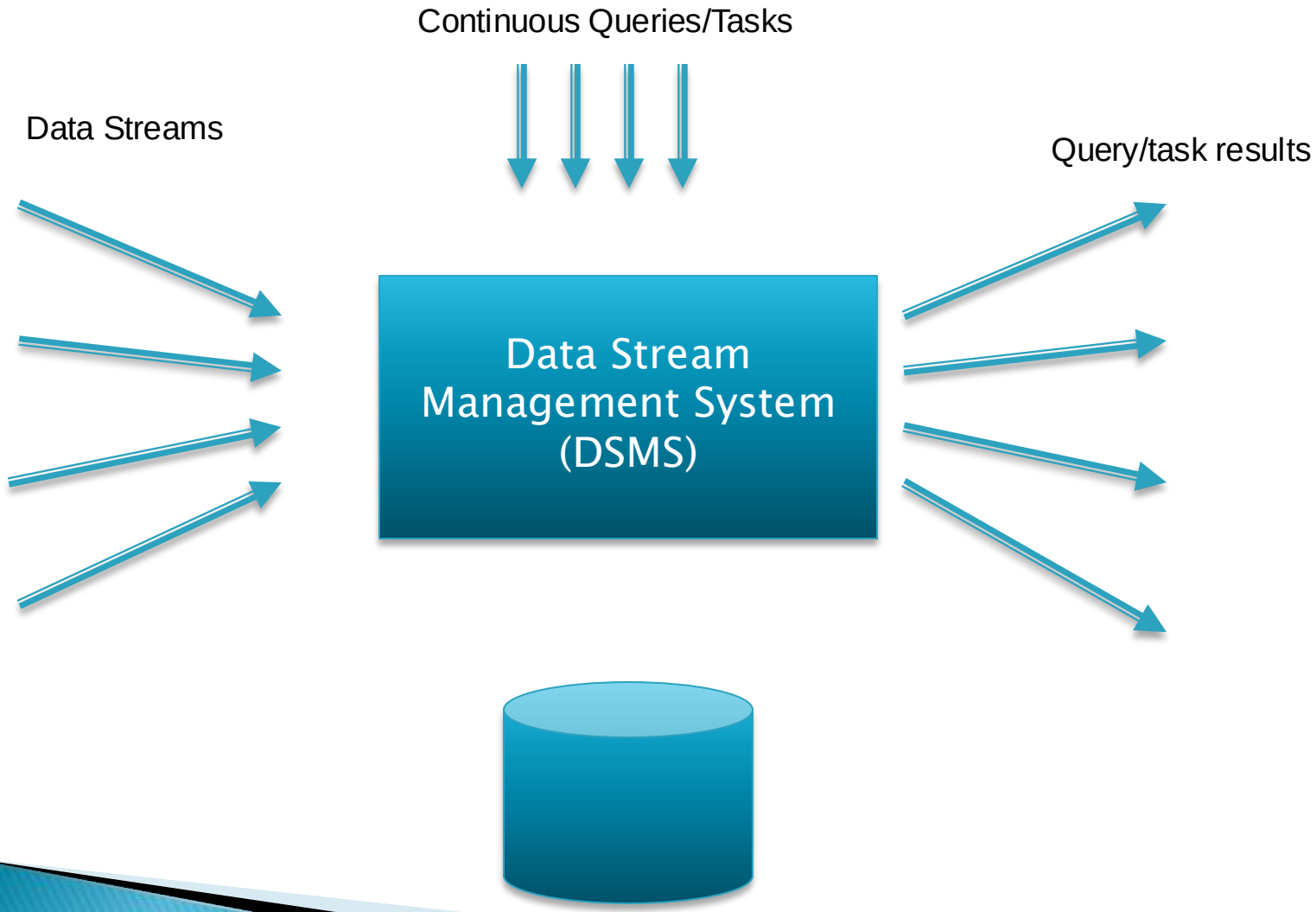
  - Potentially very large intermediate tables

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]
```

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

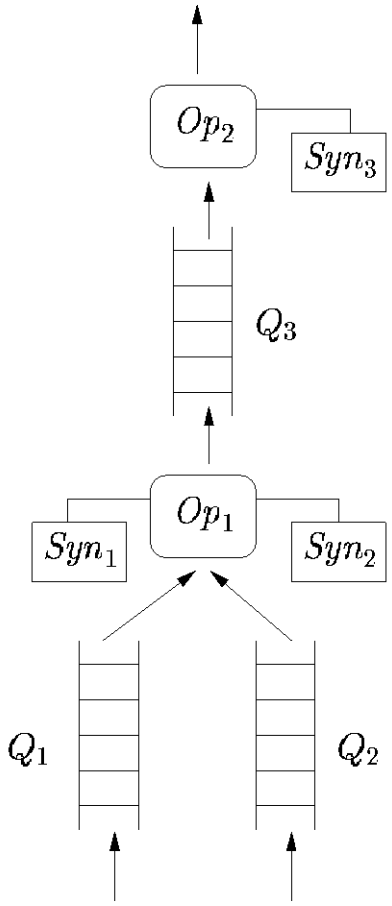
Implicit assumption that customers is a static table
Otherwise, semantics can be tricky

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*

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
  - *m*1 =3,*m*2 =*m*3 =2,*m*4 =*m*5 =1.
- Frequency moment *F_k* = Σ *m_i*^*k*
  - *F*0 = 5 = number of distinct elements in the stream
  - *F*1 = 9 = total number of elements in the stream (trivial)
  - *F*2 = 19 = comes in up many places (e.g. self-join size of a relation)
  - F_∞ = most frequent item's multiplicity
- Flajolet-Martin technique for F_0

Flajolet-Martin Technique for F_0

- Goal: Count distinct number of values in a sequence in *one pass* with minimum memory
  - N = Length of the sequence
  - n = Number of distinct values

- Naive Approach: Keep a list of all distinct values, and update incrementally
  - $O(n)$

- FM-Sketches: Approximate counting in $O(log(n))$ space

Flajolet-Martin Technique for F_0

- Algorithm:
  - Use a bitmap, B, of size k, where k is $\approx \theta(log2(n))$
  - Aren't we trying to estimate n ?
    - Use a rough upper bound. Even if you overestimate by a *factor* of 4, you only use 2 more bits.
  - Need a uniform hash function: $h(x)$ maps values in the sequence to $\{0,\cdots,2k-1\}$.
  - For each value, v in the sequence, find $h(v)$.
  - Let $l(h(v))$ denote the *least-significant 1 bit* in $h(v)$.
    - $k = 6$, $h(v) = 000100$, then $l(v) = 3$
    - $k = 6$, $h(v) = 000101$, then $l(v) = 1$.
  - Set $B(l(v)) = 1$
  - Note: Duplicate values will just set the same bit again ➔ "duplicate-insensitive"

Flajolet-Martin Technique for F_0

- Algorithm (Cntd):
  - At the end, let c be the least-significant (right-most) 0 in B
  - 1.2928×2^c is an estimator for the number of distinct values
  - Why ?
    - Choose a number, x, uniformly between 0 to $2k - 1$
    - $Prob(l(x) = c) = 1/2^{c+1}$
    - Hash function is assumed to map values in the sequence uniformly onto the above range as well
- Use multiple hash functions for more confidence
- Space: $O(log(n))$
- Choosing hash functions ?
  - Tricky: uniform hash functions take a lot of space
  - Much work on relaxing the requirement

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

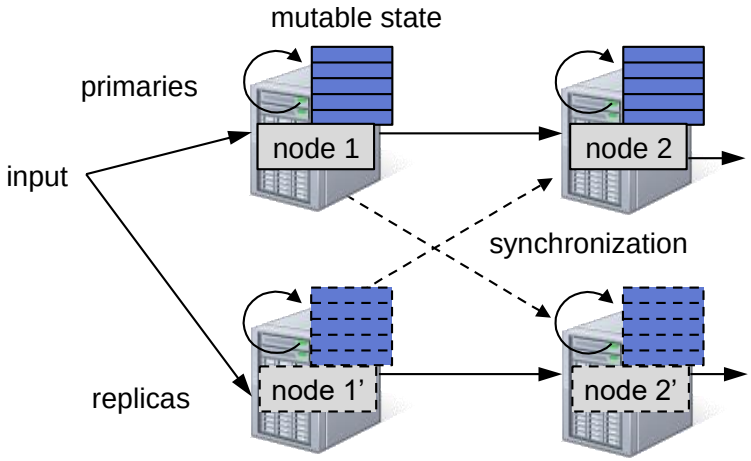
- Overview and Early Work
- Maintenance of Materialized Views
- Models and issues in data stream systems
- Discretized streams: fault-tolerant streaming computation at scale
- Apache Flink: Stream and Batch Processing in a Single Engine
- Incremental, Iterative Data Processing with Timely Dataflow
-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

Motivation

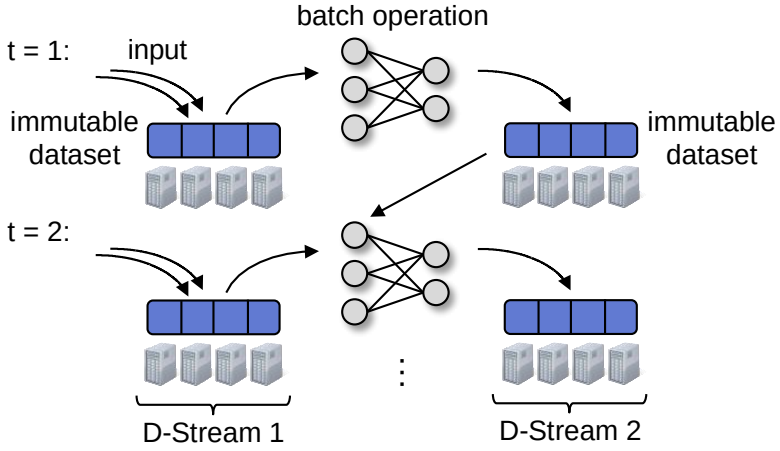
- Need streaming computation models analogous to MapReduce for batch processing
  - Applications: detecting trends, real-time personalization, failure detection, spam detection, advertising statistics,
- Challenges
  - Computational high-level frameworks for distributed systems
  - Machine failures and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

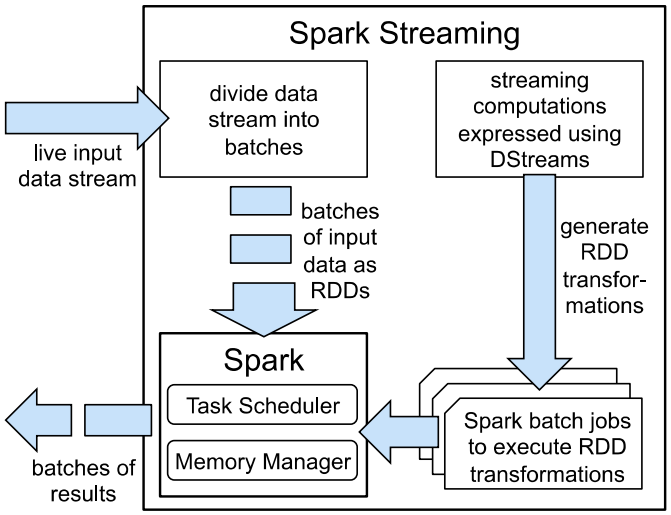


Figure 2: High-level overview of the Spark Streaming system. Spark Streaming divides input data streams into batches and stores them in Spark’s memory. It then executes a streaming application by generating Spark jobs to process the batches.

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

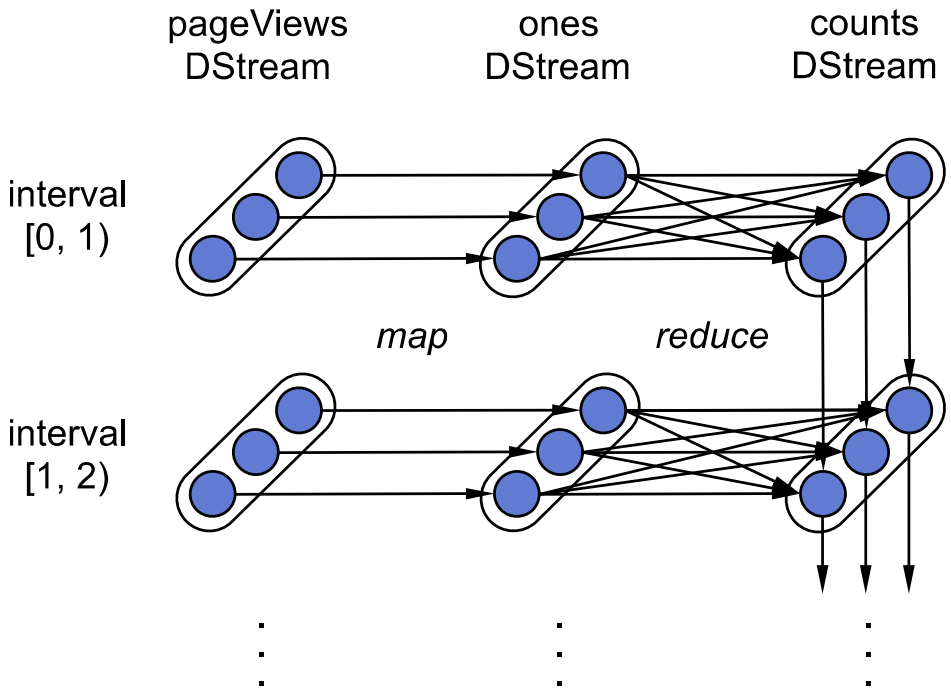


Figure 3: Lineage graph for RDDs in the view count program. Each oval is an RDD, with partitions shown as circles. Each sequence of RDDs is a D-Stream.

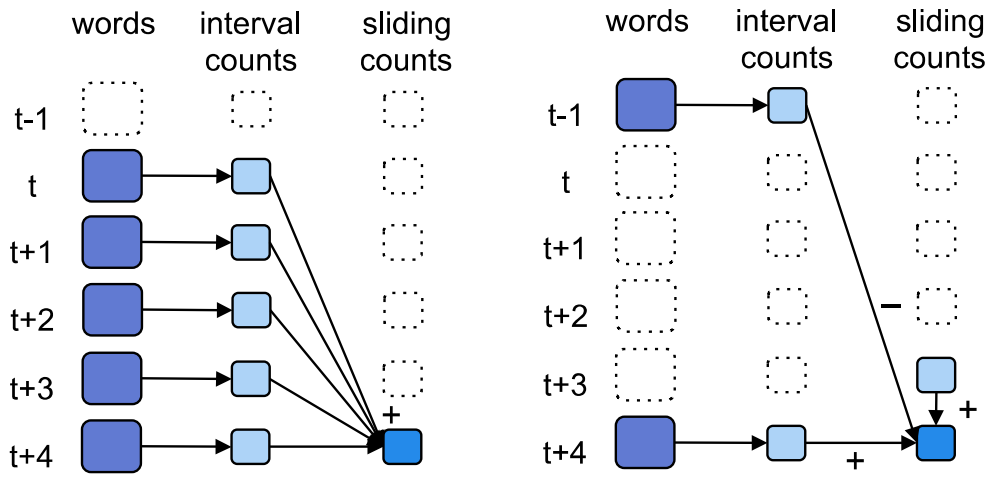
Timing/Timestamps Issues

- 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 $[t, t+5)$.

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
Consistency	Records processed atomically with interval they arrive in	Some systems wait a short time to sync operators before proceeding [5, 33]
Late records	Slack time or app-level correction	Slack time, out of order processing [23, 36]
Fault recovery	Fast parallel recovery	Replication or serial recovery on one node
Straggler recovery	Possible via speculative 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

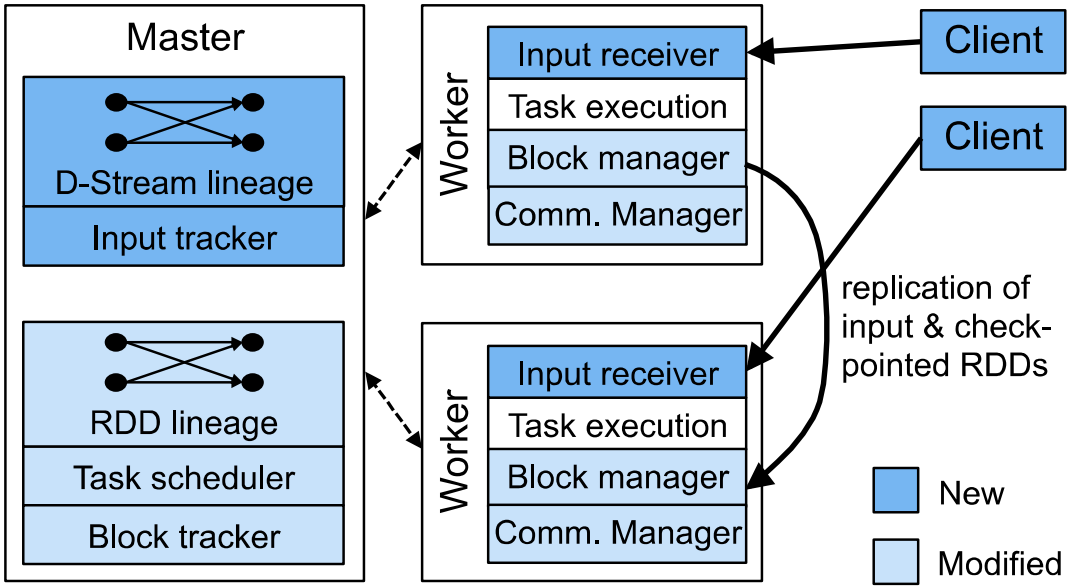


Figure 6: Components of Spark Streaming, showing what we added and modified over Spark.

Implementation Details

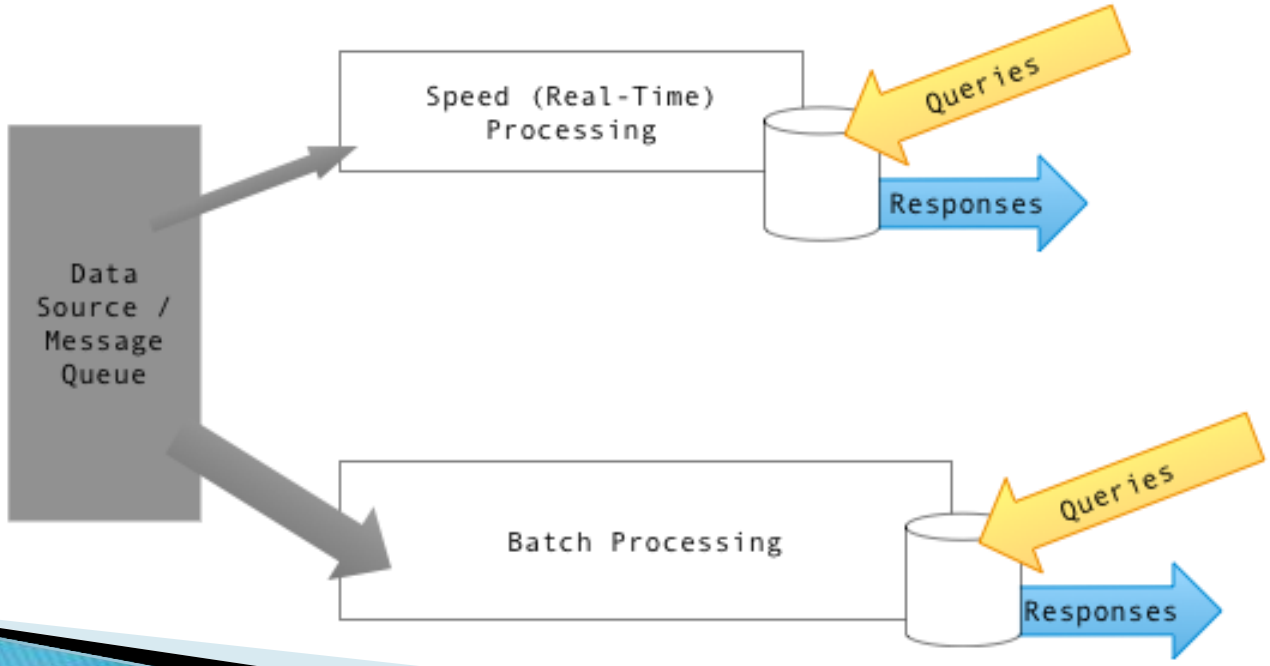
- Parallel Recovery
  - If a node fails, RDDs are recomputed using lineage graphs
- Straggler mitigation
  - Can use speculative backup copies
  - Determinism allows for a exploring different options
- Master recovery
  - In standard Spark, ”master” (”driver”) is not fault-tolerant
  - Use checkpointing 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
- Watermarks (VLDB 2021 Paper)
- Incremental, Iterative Data Processing with Timely Dataflow

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

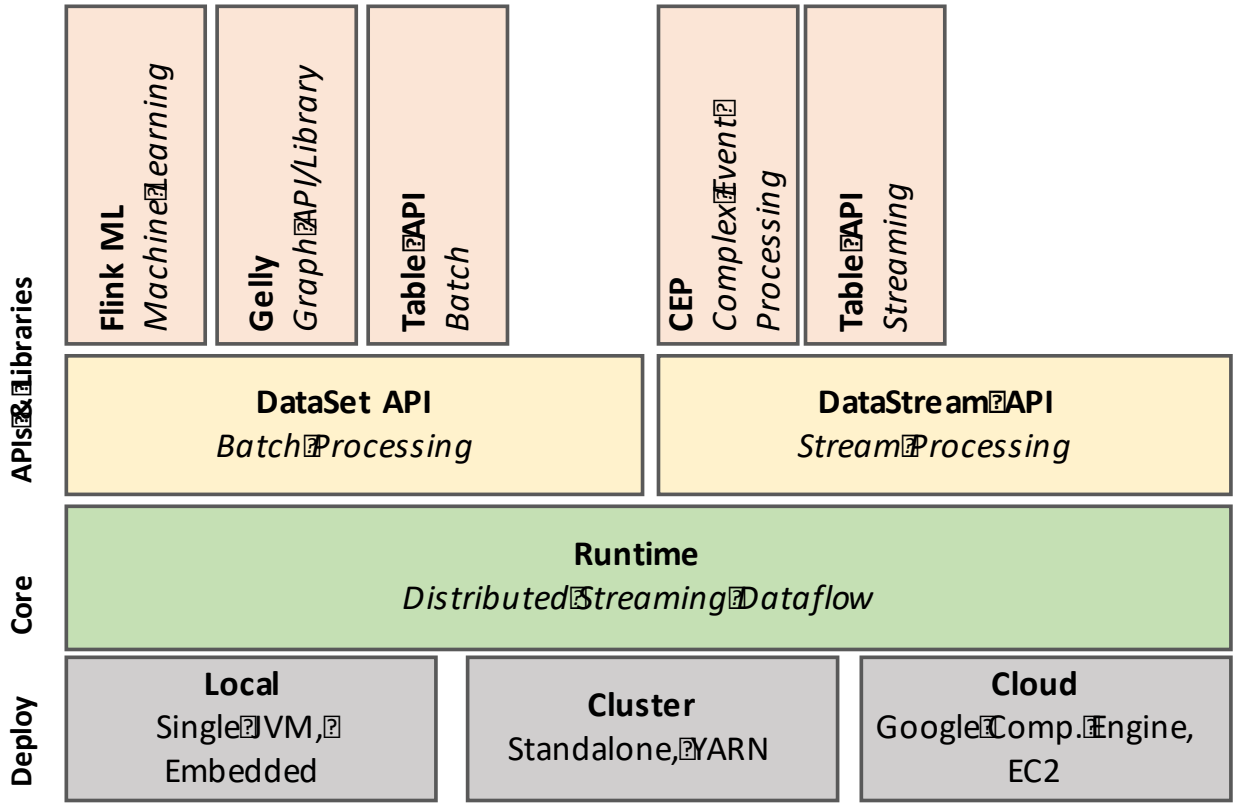


Figure 1: The Flink software stack.

Flink Process Model

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

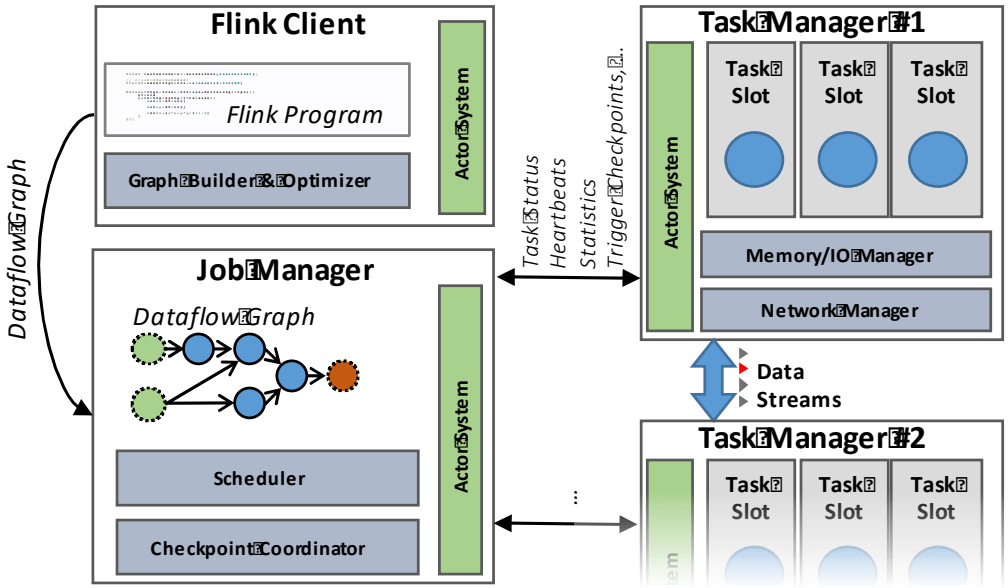


Figure 2: The Flink process model.

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)

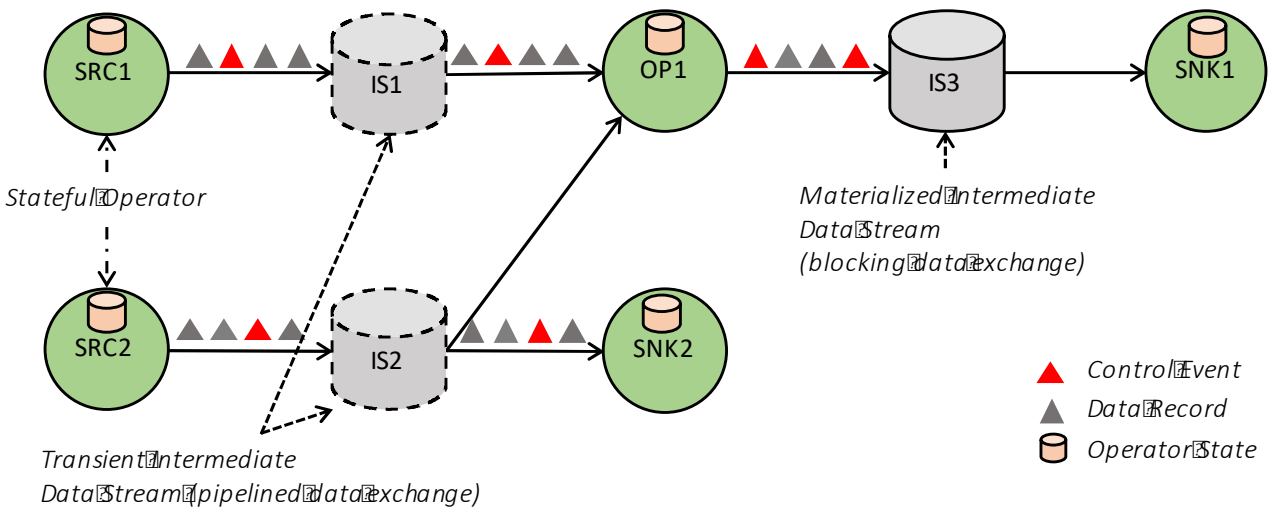


Figure 3: A simple dataflow graph.

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

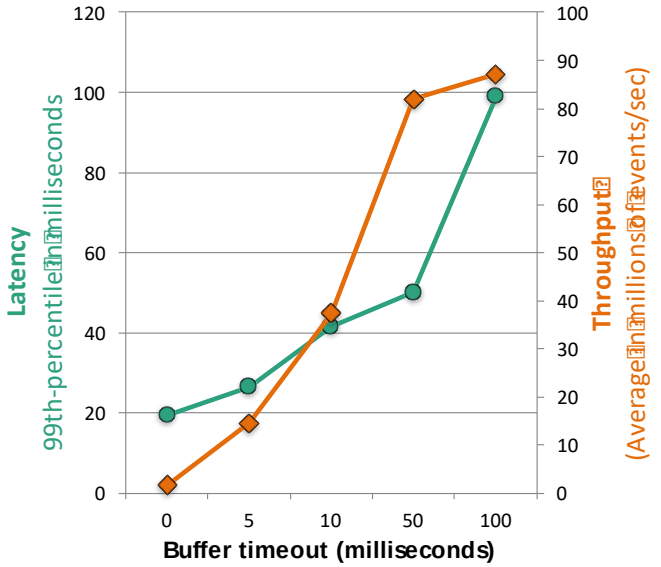


Figure 4: The effect of buffer-timeout in latency and throughput.

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

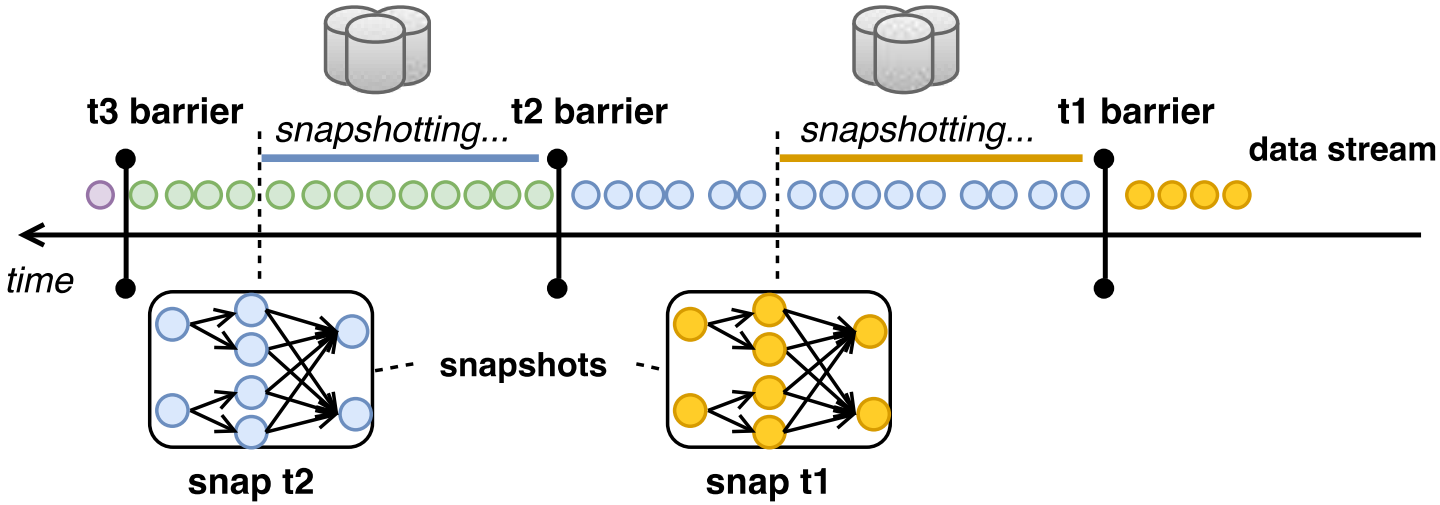


Figure 5: Asynchronous Barrier Snapshotting.

Iterative Dataflows

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

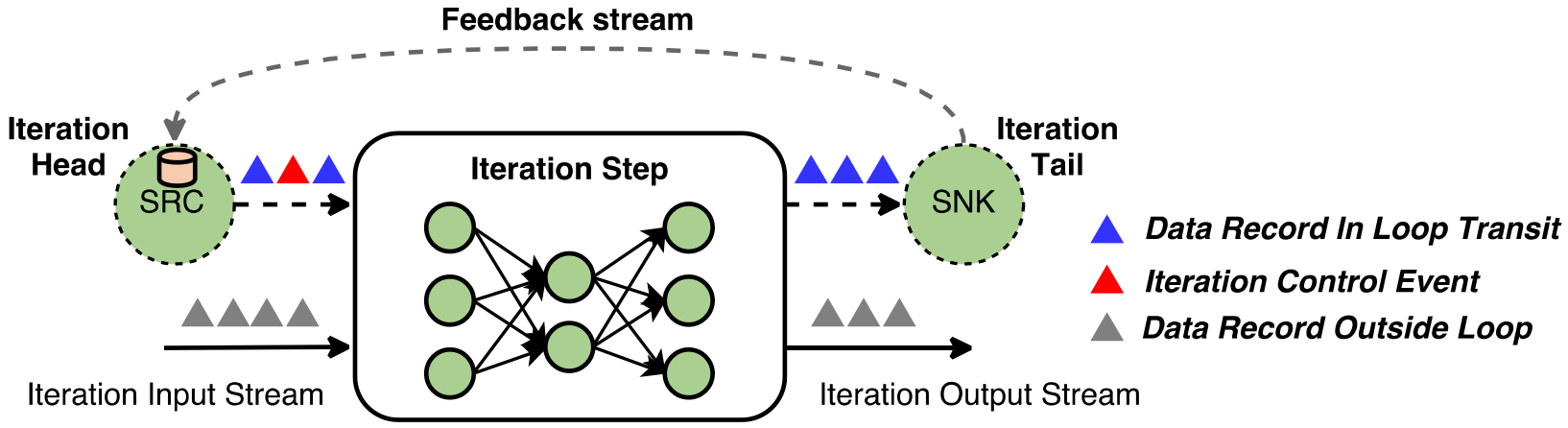


Figure 6: The iteration model of Apache Flink.

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

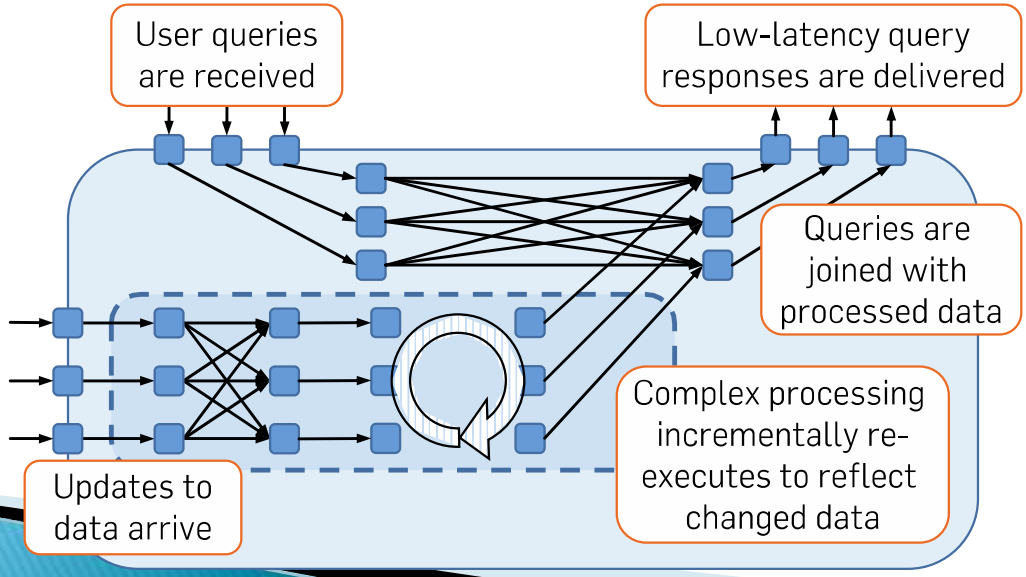
Motivation

- 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

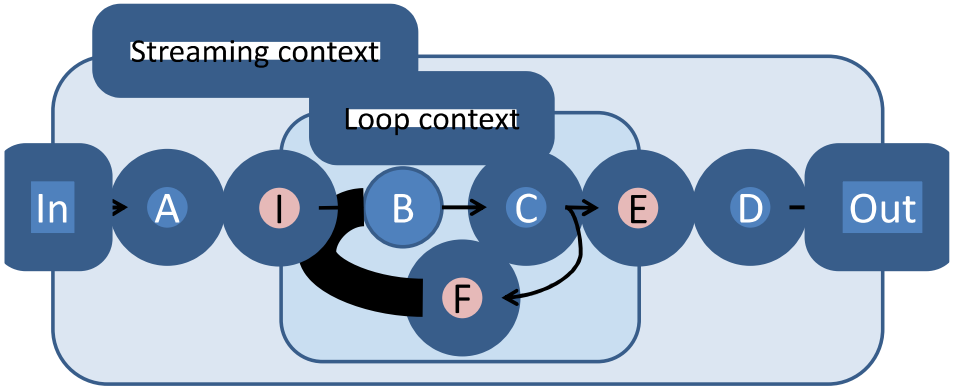


Figure 3: This simple timely dataflow graph (§2.1) shows how a loop context nests within the top-level streaming context.

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

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

- And may use two methods:

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

- 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

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.

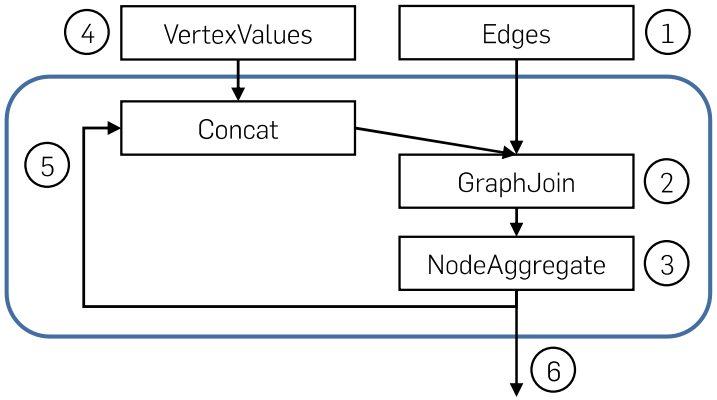
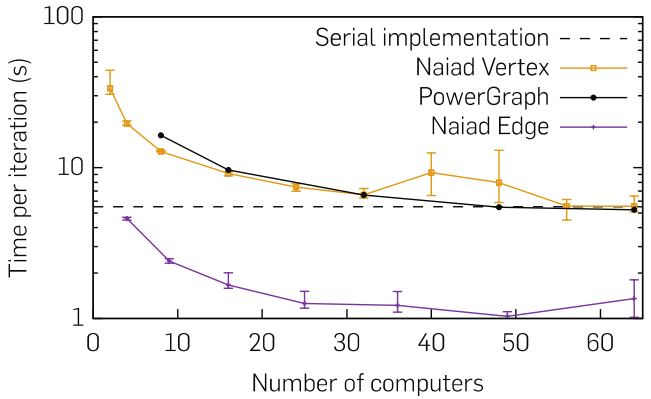


Figure 4. Time per iteration for PageRank on the Twitter follower graph, as the number of machines is varied.



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

- Part 1:
  - Overview and Early Work; Maintenance of Materialized Views
  - Models and issues in data stream systems
  - Discretized streams: fault-tolerant streaming computation at scale
- Watermarks (VLDB 2021)
- Incremental View Maintenance (VLDB 2023 Best Paper Award)

Motivation

- Event time vs Processing time
  - Application time vs System time (SQL:2011)
  - Valid time vs Transaction time (temporal databases literature)
- Disorder in distributed dataflow systems
  - Producers sending data at varying rates
  - Events flow along distinct paths, with different delays
- Watermarks
  - Quantitative markers that indicate no future events with a lower timestamp
  - Used by most systems today to deal with disorder

Watermarks

- Formal definition

> More formally, for a node N of a dataflow graph G and a totally-ordered time domain T, let I be the sequence of input elements arriving at N. The arrival timing of these elements is given by a processing time function $p : I \to T$. Each element is tagged with an event timestamp, given by the function $e : I \to T$. A *watermark* for N is a function $w : T \to T$ satisfying the following properties.
>
> - **Monotonicity**: w is monotone. It must never "move backwards".
> - **Conformance**[1]: for all $i \in I$, $w(p(i)) < e(i)$. A conformant watermark must bound event times from below.
> - **Liveness**: $w(t)$ has no upper bound.

- Watermarks eventually make forward progress, but allow unbounded "lag" ➔ May be useful to put a constraint on *t-w(t)*

Key Uses

- Readiness
  - Consumer of the output may not support updates
  - Partial results may be unstable or misleading
    - e.g., when detecting anomalies
  - Computation may be non-incremental
    - e.g., ML training – expensive to redo if a data element arrives late
- Obsolescence
  - When can state be discarded? Need to ensure any new late events that arrive can be handled
- Health Monitoring
  - Watermarks help with debugging performance or other issues

Alternative Approaches

- Order-agnostic processing
  - Process when it comes
  - Need ability to handle corrections etc.
- Ordered processing
  - Assume no disorder
- Watermarks
- Timestamp frontiers
  - From timely dataflow: generalization of watermarks
- Punctuations
  - A more general technique that applies to any attributes (e.g., no more tuples with A > 10)

Example

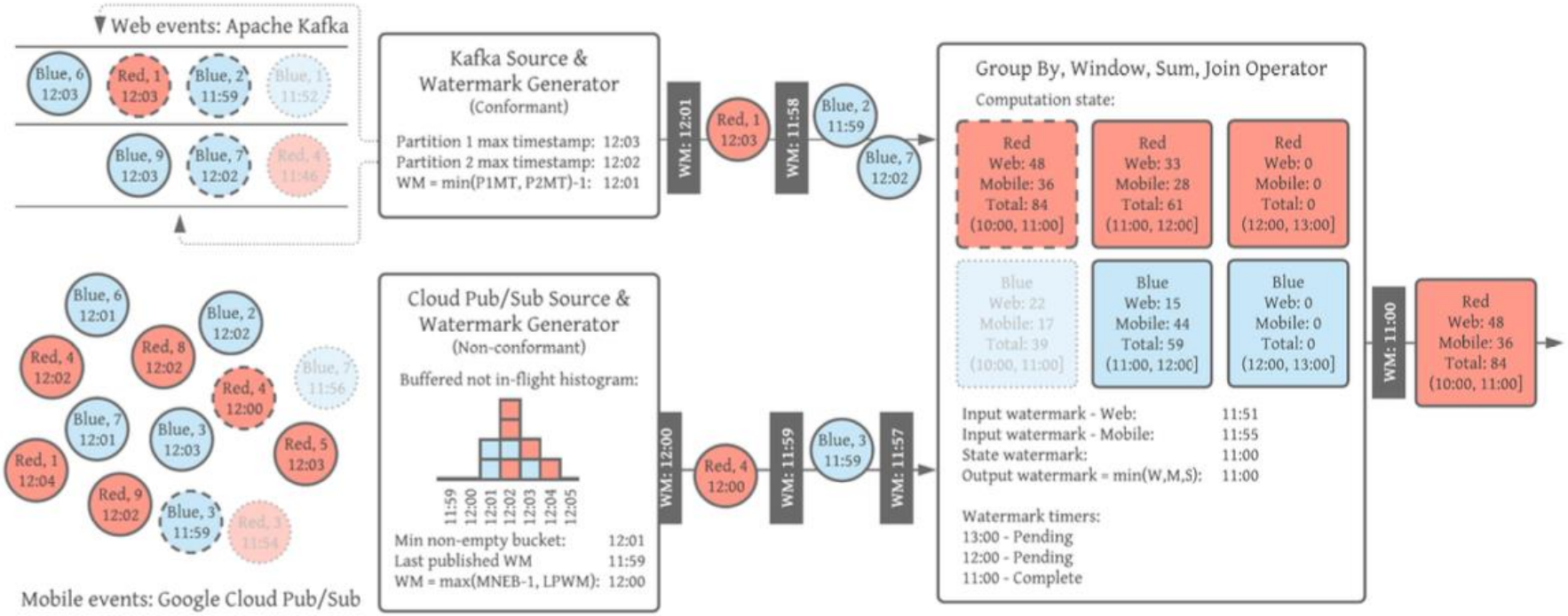


Figure 1: Example streaming analytics pipeline at time t_0, computing hourly per-team score aggregates. Individual score events for each team member are represented as circles containing the team name, score, and event time. Watermark updates are black rectangles. Summary aggregates are rounded rectangles. Sources, watermark generators, and operators are labeled as such. Faded events with dotted outlines have been processed already by the corresponding operator, and events with dashed outlines are in-flight; these markings help illustrate progress over time when comparing against Figure 2.

Example

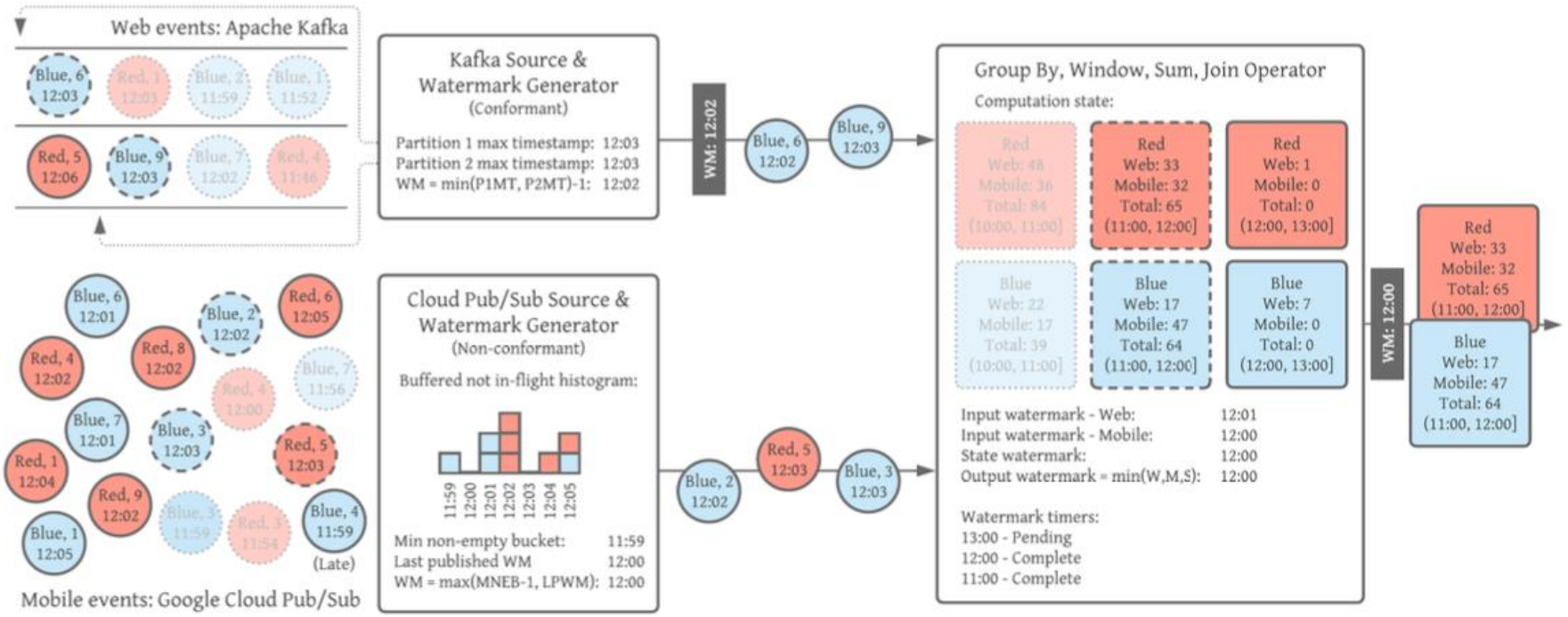


Figure 2: Example pipeline at time $t_1 > t_0$. In this figure, time has advanced such that the input watermark has reached 12:00 at the aggregation operator. As a result, the hourly summaries for the (11:00, 12:00] windows have been emitted downstream, and the output watermark advanced to 12:00 as well. Sources and watermark generators have progressed, with Kafka offsets and watermarks increased per partition, and the histogram of pending Pub/Sub timestamps evolved. Some new data has arrived. Note the late (Blue, 4, 11:59) event in the mobile source, rendering that watermark non-conformant.

Watermark Generation

- Single source that is generating data in an ordered fashion can add watermarks in the data
  - e.g., a mobile app that's sending data to Kafka, or a web server that's logging incoming requests
- Assume bounded disorder
  - When a node sees an event with time t, it moves the watermark to t – D, where D is configurable "lateness" parameter
  - Can also use "timeouts"
  - Problematic as they introduce delays when not needed
- Adapt watermark delay using statistical models

Watermark Propagation and Use

- Assume all input nodes can introduce watermarks
- For any other node:
  - Watermark > any of the watermarks on its incoming edges
  - Specific values depend on the computation inside the node
- Stateless nodes that perform element-wise computation (e.g., “map”)
  - Watermark = watermark of its incoming edge
- Aggregation/Join nodes
  - Watermark should not exceed max of its incoming edges

- Using Watermarks
  - Typically through use of a callback (watermark timers, or triggers)
  - What if data arrives late (non-conformant watermarks)? ➔ ignore, or send corrections

Watermarks in Flink vs Google Cloud Dataflow

- Generation
  - Flink: Source nodes or special watermark generation nodes
  - Flink: Publishes either periodically or directly – either case, added directly to the stream itself
  - Cloud Dataflow: Uses Apache Beam functionality similarly to Flink
  - Cloud Dataflow: Nodes periodically publish watermarks to a global watermark aggregator (i.e., out-of-band)
- Propogation:
  - Flink: Special control/metadata messages in the stream itself
  - Cloud Dataflow: Watermarks managed out of band, by tracking the minimum watermarks across different partitions etc.
- Reacting to Watermarks
  - Through use of watermark timer callbacks

Design Tradeoffs

- Watermark reporting latency: Higher for Cloud Dataflow
- Fault Recovery:
  - Slower in Flink due to the way checkpointing works
- Consistency
  - Trickier in Cloud Dataflow due to dynamic range partitioning and coordination with watermark aggregator
- Idle workers
  - A source with no new watermarks can cause problems for Flink
- Bottlenecks
  - Problem for both
- Watermark skew
  - Need additional techniques to handle this

Outline

- Part 1:
  - 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, Naiad/Differential Dataflow
  - MacroBase: Prioritizing Attention in Fast Data
- Watermarks (VLDB 2021)
- Incremental View Maintenance (VLDB 2023 Best Paper Award)
- New SQL Proposal (SIGMOD 2019)
- Serverless computing (OOPSLA 2021)

IVM Background

- Goal: Incrementally maintain the result of an SQL query (i.e., a materialized view)

R

A	B
a1	b1
a2	b2
a1	b3

S

B	C
b1	c1
b2	c2
b2	c3

Create View V as
select *
from R natural join S



V

A	B	C
a1	b1	c1
a2	b2	c2
a2	b2	c3

IVM Background

Example 1: select * from R where A = a1

- Note: SQL semantics are no duplicate elimination
- Straightforward – can compute **ΔV1 from ΔR1**

R^0

A	B
a1	b1
a2	b2
a1	b3

ΔR^1

A	B
a1	b4
a2	b3
a1	b3

R^1

A	B
a1	b1
a2	b2
a1	b4
a2	b3

V^0

A	B
a1	b1
a1	b3

ΔV^1

A	B
a1	b4
a1	b3

V^1

A	B
a1	b1
a1	b4

IVM Background

Example 2: select distinct B from R where A in [a1, a2]

- Need access to the original relation to compute the change
- Can't remove b1 – there is another "derivation"

R^0

A	B
a1	b1
a2	b1
a1	b3

ΔR^1

A	B
a1	b4
a1	b3
a1	b1

R^1

A	B
a2	b1
a1	b4

V^0

B
b1
b3

ΔV^1

B
b4
b3

V^1

B
b1
b4

IVM Background

Example 2: select distinct B from R where A in [a1, a2]

- But we can keep track of the # of derivations

R^0

A	B
a1	b1
a2	b1
a1	b3

ΔR^1

A	B
a1	b4
a1	b3
a1	b1

R^1

A	B
a2	b1
a1	b4

V^0

B	
b1	2
b3	1

ΔV^1

B
b4
b3

V^1

B	
b1	1
b4	1

IVM Background

Example 3: select * from R natural join S

Need access to original R^0 and S^0

R^0

A	B
a1	b1
a2	b1
a1	b3

S^0

B	C
b1	c1
b2	c2
b2	c3

ΔR^1

A	B
a1	b4
a2	b2
a1	b3

ΔS^1

A	B
b1	c2
b2	c4
b2	c3

V^0

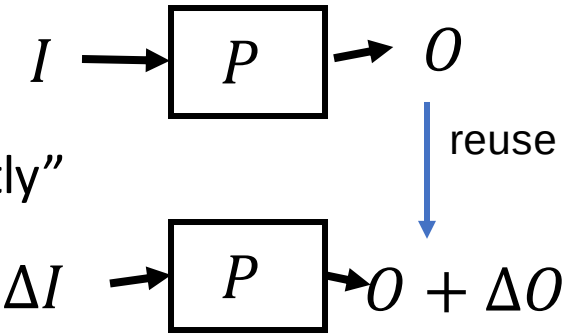
A	B	C
a1	b1	c1
a2	b2	c2
a2	b2	c3

ΔV^1

A	B	C	
a1	b1	c2	R^0 JOIN ΔS^1
a2	b2	c4	ΔR^1 JOIN ΔS^1
a2	b2	c2	ΔR^1 JOIN S^0
a2	b2	c3	

What is incremental computation?

- Consider a program P



- Change I "slightly"



- In general, this is hard to even define precisely

- But it works great for databases:
  - Databases have a natural notion of "changes"
  - P is a query, I is a set of tables, and ΔI is a transaction, O is a view
  - Usually $|\Delta I| \ll |I|$ and $|\Delta O| \ll |O|$
  - Database people call this "Incremental View Maintenance"

Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

A simple streaming language called DBSP

- Four (4) operators
- A new definition of incremental computation using DBSP
- An algorithm converting an **arbitrary** DBSP program to an **incremental** DBSP program
- A recipe implementing queries in DBSP
  - 2 operators needed for SQL
  - 2 for recursive queries (not in this talk)

Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

DBSP is powerful

- It can model a rich class of database queries:
  - Relational (select, project, join, union, difference)
  - Computations on sets, bags, multisets
  - Nested relations (group-by, unnest)
  - Aggregation
  - Recursion (monotonic and non-monotonic)
  - Stratified negation
  - Streaming computations (joins, window aggregates)
  - More?

SQL

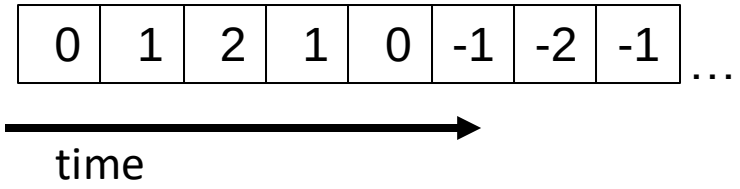
Datalog

- Corollary: we get automatically incremental evaluation for all these

Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

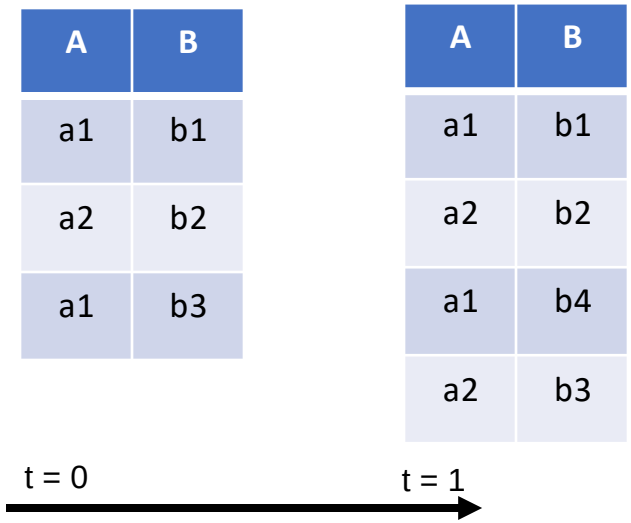
Streams

- Infinite vectors

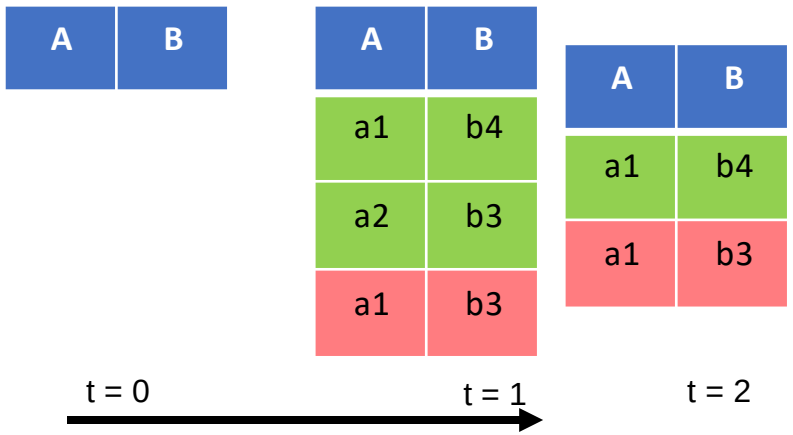


- S_A = streams with elements of type A
- Assume that A has $+$, $-$, and 0 (zero)
  - (A is a commutative group)

Database: stream with type: set of tuples

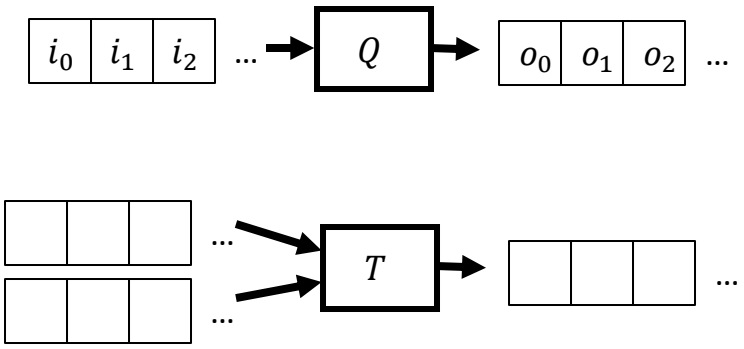


Deltas/Transactions: stream of weighted set of tuples



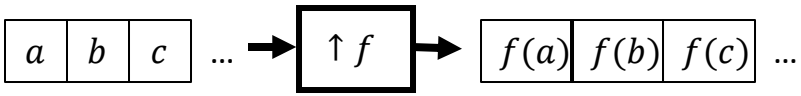
Stream operators

- Arrows = streams
- Boxes = operators



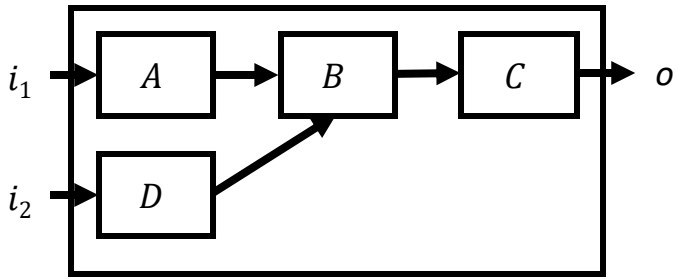
Lifting

Convert a function $f : A \to B$
into a stream operator $\uparrow f : S_A \to S_B$



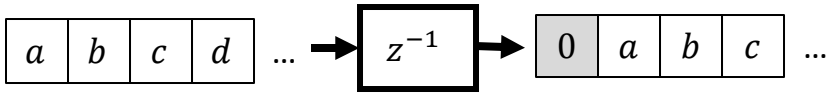
Chaining operators into circuits

"Dataflow graphs"

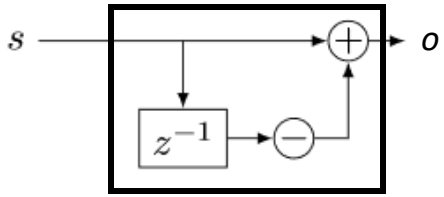


Delay (z^{-1})

- Output is input stream delayed by one step
- First value is 0

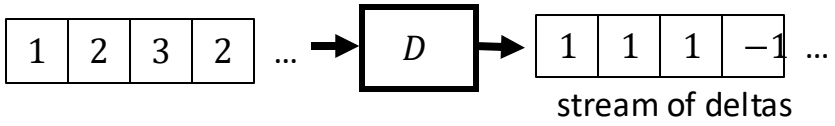


Computing changes (deltas)

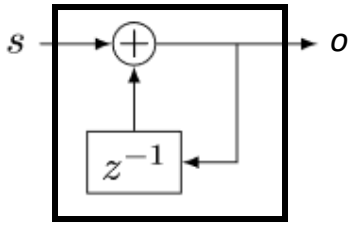


$o[t] = s[t] - s[t-1]$

o is the *stream of changes* of *s*

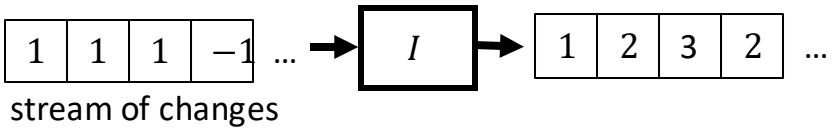


Integration



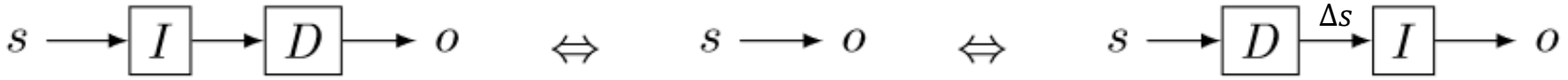
- $o[0] = s[0]$
- $o[t] = s[t] + o[t-1]$
- $o[t] = s[0] + s[1] + \ldots + s[t]$

- If *s* is a *stream of changes…*
- … then *o* is the original stream



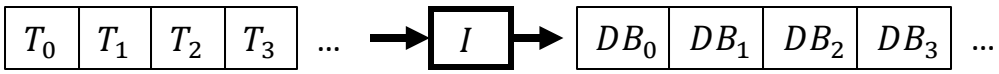
Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

D and I are inverses



All databases are streaming databases!

- Consider a database DB, a set of tables
- A committed transaction is a **change** to DB
- The set of linearized transactions define T, a **stream of changes** to DB
  - $T[k]$ is the k-th transaction
- DB is a **stream** of database snapshots
  - $DB[k]$ is the contents of the database after k transactions have been committed
- $DB[k] = I(T)[k]$
  - A database (stream) is the integral of a transaction stream



Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

Views are lifted queries

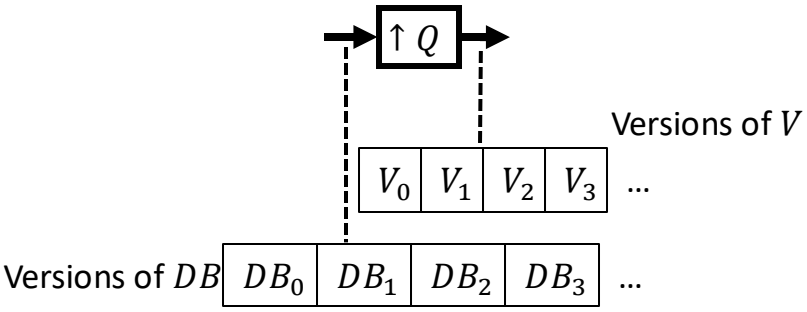
- Let Q be a query defining a view V, e.g.:

```
CREATE VIEW V AS
SELECT * FROM T
WHERE Age >= 10
```

$$V[k] = Q(DB[k])$$

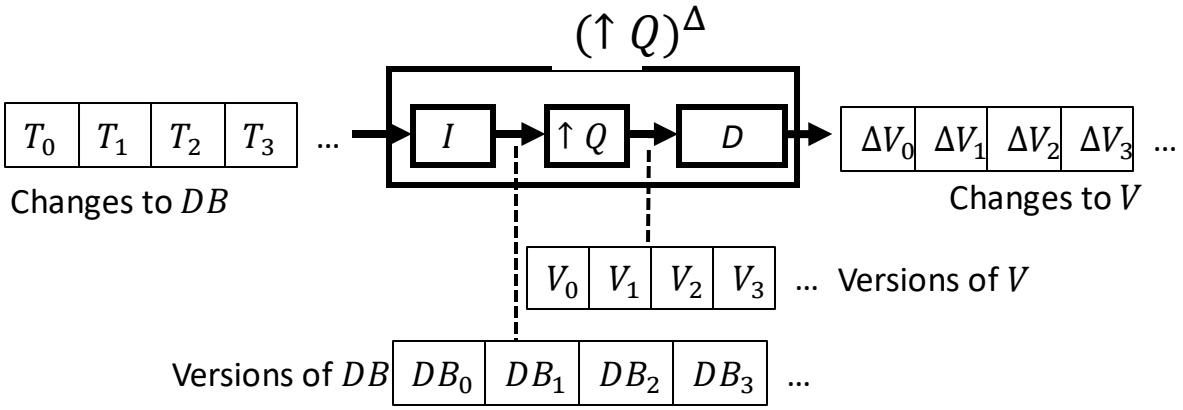
- V is a stream of view snapshots:

$$V = \uparrow Q(DB)$$

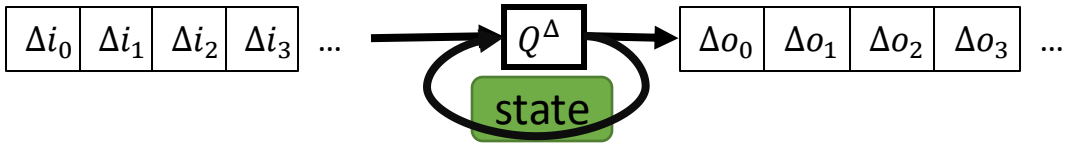


Incremental view maintenance of view V

- This is our definition of IVM
- The standard DB definition does not include the last D
- This definition is much better:
  - It is compositional
  - Inputs and outputs are both **deltas**



Incremental queries are streaming systems

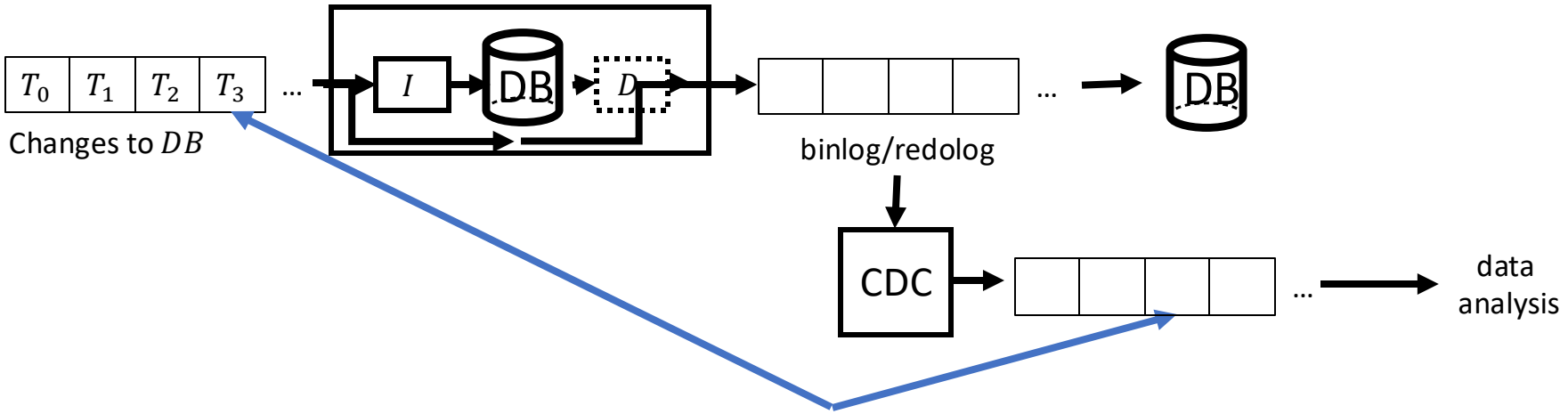


Q^Δ is a **streaming system**:

- it runs forever
- it maintains internal state (even if Q is stateless)
- the state is stored in the delay operators (inside I and D)

Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

Change Data Capture and DB replication



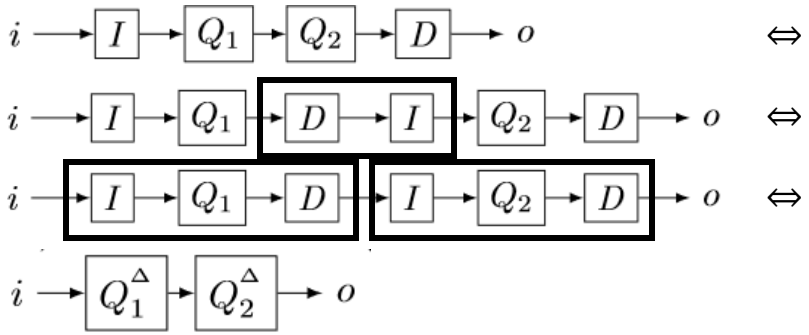
- *These streams have contents of the same nature*
- *They should use the same formats*
- *The DB should have APIs to publish streams of changes for any view*

Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

The chain rule (structural recursion)

$$(Q_2 \circ Q_1)^{\Delta} = Q_2^{\Delta} \circ Q_1^{\Delta}$$

- Proof by pictures:



Incremental operators

- $(Q_2 \circ Q_1)^{\Delta} = Q_2^{\Delta} \circ Q_1^{\Delta}$
- If we know Q^{Δ} for all **primitive** operations…
  …then we can incrementalize **any query**
- For all standard primitive DB operations
  we know efficient incremental versions

Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

Let’s apply this to a DB

We need +, -, and 0 operations for a database



Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

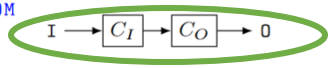
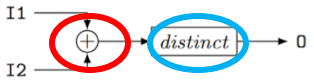
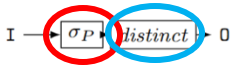
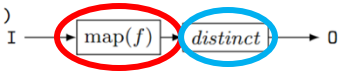
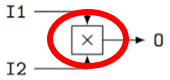
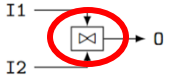
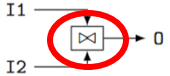
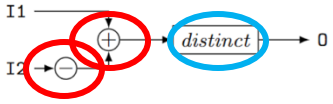
$\mathbb{Z}$-sets (also called $\mathbb{Z}$-relations)

- Each row has an integer weight
  - The weight can be positive, zero, or negative
- Can represent **both** tables and **changes** to tables
  - Positive weight => row added
  - Negative weight => row removed
- Generalize sets and multisets
  - Classic DB table = $\mathbb{Z}$-set where all weights are 1
- Form a commutative group

Name	Age	Weight
Mike	10	1
John	12	3
Amy	8	-1
Chris	10	2

tuples

Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

Operation	SQL example	DBSP circuit
Composition	`SELECT DISTINCT ... FROM (SELECT ... FROM ...)`	
Union	`(SELECT * FROM I1) UNION (SELECT * FROM I2)`	
Projection	`SELECT DISTINCT I.c FROM I`	
Filtering	`SELECT * FROM I WHERE p(I.c)`	
Selection	`SELECT DISTINCT f(I.c, ...) FROM I`	
Cartesian product	`SELECT I1.*, I2.* FROM I1, I2`	
Join	`SELECT I1.*, I2.* FROM I1 JOIN I2 ON I1.c1 = I2.c2`	
Intersection	`(SELECT * FROM I1) INTERSECT (SELECT * FROM I2)`	
Difference	`SELECT * FROM I1 EXCEPT SELECT * FROM I2`	

Relational algebra using $\mathbb{Z}$-set operations

- Recursively defined on query structure
- Many operations require a $distinct$
  - (But many can be removed)
- The other operations are all linear or bilinear

Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

Algorithm for incremental view maintenance

1. Translate recursively a DB query into a circuit Q on $\mathbb{Z}$-sets

   1.5 Eliminate some *distinct* calls

2. Lift circuit to compute on streams: $\uparrow Q$
3. Build incremental version of lifted circuit:

$$(\uparrow Q)^{\Delta} \stackrel{\text{def}}{=} D \circ \uparrow Q \circ I$$

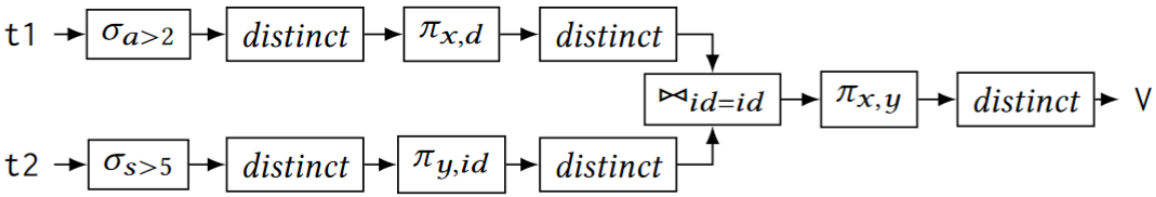
4. Optimize Q^{Δ} using structural recursion:

$$(Q \circ R)^{\Delta} = Q^{\Delta} \circ R^{\Delta}$$

This algorithm is deterministic. There are no heuristics.

Example

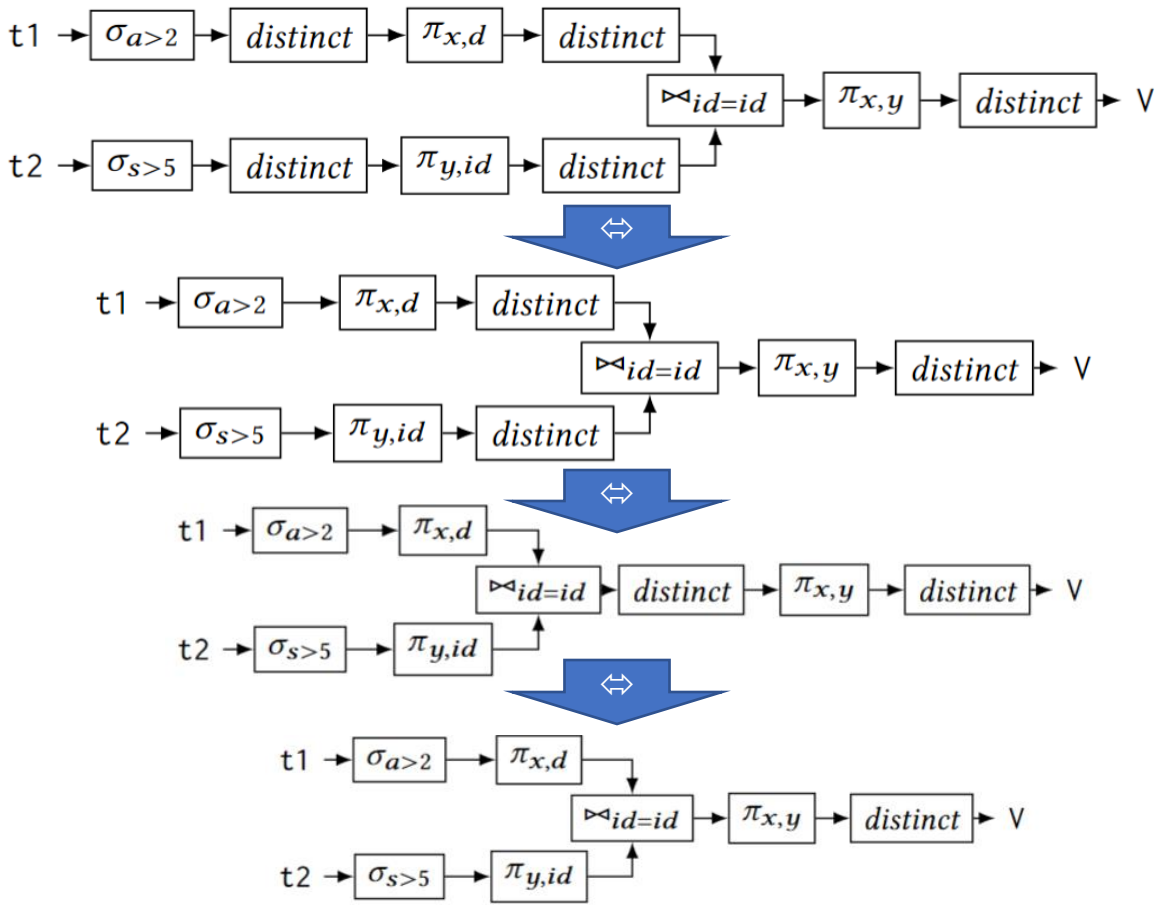
```
CREATE VIEW V AS
SELECT DISTINCT a.x, b.y FROM (
  SELECT t1.x , t1.id
  FROM t1
  WHERE t1.a > 2
) a
JOIN (
  SELECT t2.id , t2.y
  FROM t2
  WHERE t2.s > 5
) b ON a.id = b.id
```



Inputs and outputs are collections, not streams

Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

Example: Remove `distinct` calls



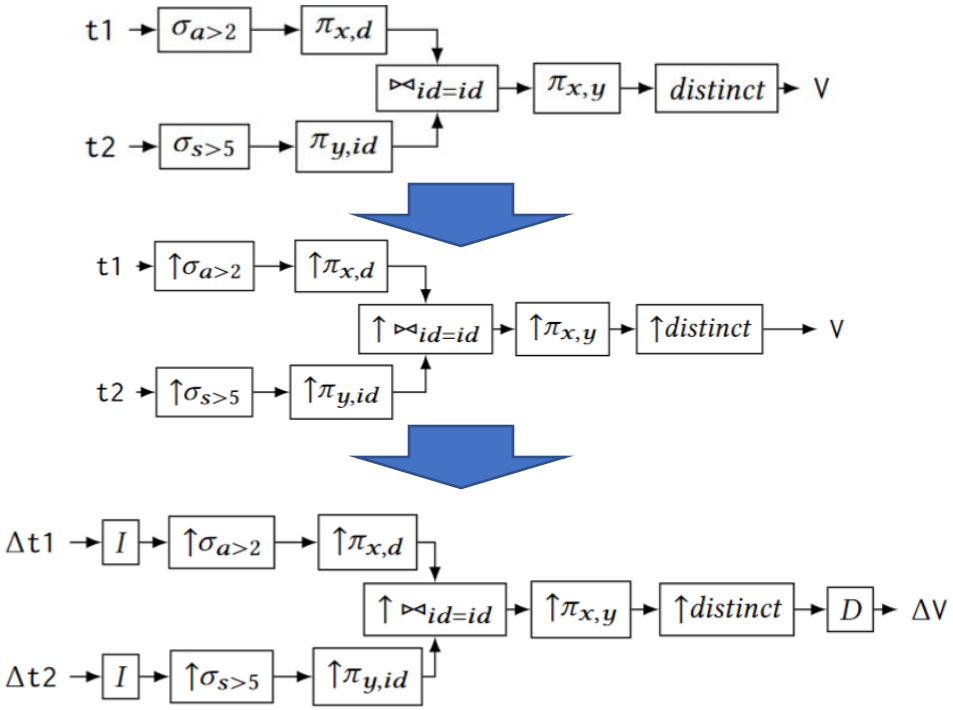
Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

Example: Lift and incrementalize

Inputs and outputs
are collections (z-sets)

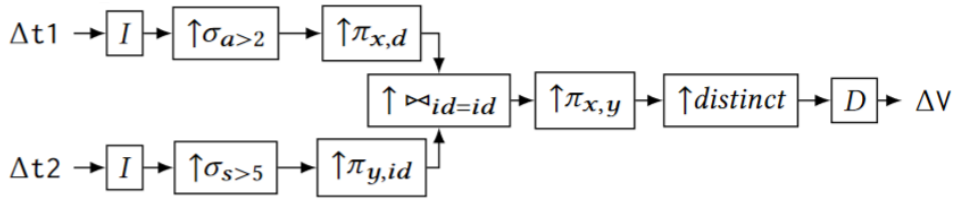
Inputs and outputs
are streams

Inputs and outputs
are streams of changes

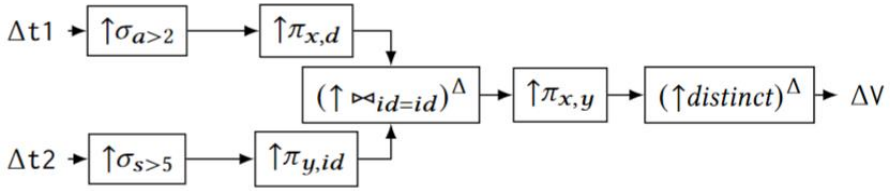


Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

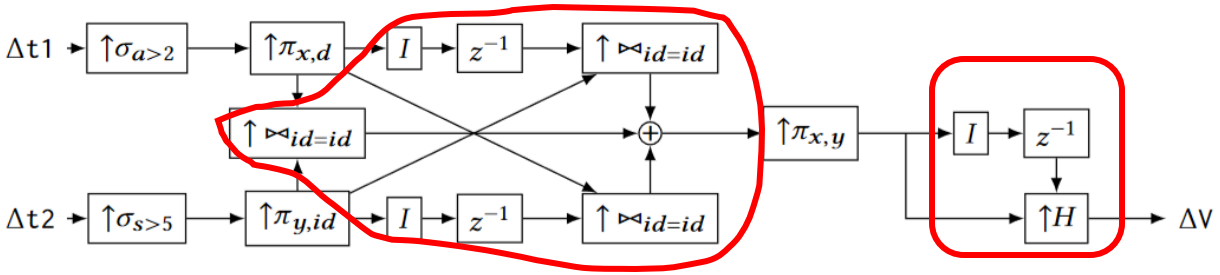
Example: optimize incremental circuit



Chain rule and linearity



Expand join and distinct

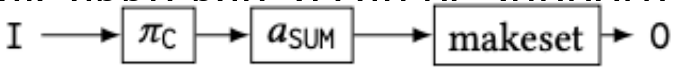


Slides adapted from Mihai Budiu (https://mihaibudiu.github.io/work/index.html)

Richer Query Languages

- Multisets (i.e., duplicate-free semantics of SQL)
  - Z-Sets are naturally multisets – need distinct to handle set semantics

- Aggregations
  - Count, Sum, Avg, etc., (algebraic aggregates) can be supported through some additional constructs



  - ”select sum(c) from I” ➔
  - Min/Max/Median etc., are not linear, so require brute-force approach
    - Recent work on optimizing these

- Grouping, Group-by Aggregate, Flatmap (unnest) easy to support

More

- Recursion also naturally supported
  - Not considered but much of the prior work on this topic
- Ongoing implementation efforts, through a startup
  - https://www.feldera.com
  - https://github.com/feldera/feldera
  - Uses the Calcite Query Optimizer to support a broad subset of SQL