Some Reminders for a Seamless Online Class…

- Please turn on your video
- Mute yourself (press and hold spacebar when you'd like to talk)
- Don't do anything you wouldn't do in an in-person class
- I will occasionally check the chat for messages if you'd like to share there instead
- Please say your name before you speak

Recap

- Data-savviness is the future!
- "Classical" relational databases
    - Notion of a DBMS
    - The relational data model and algebra: bags and sets
    - SQL Queries, Modifications, DDL
    - Database Design
    - Views, constraints, triggers, and indexes
    - Query processing & optimization
    - Transactions
- Non-classical data systems
    - Data preparation:
        - Semi-structured data and document stores
        - Unstructured data and search engines
    - Data Exploration:
        - Cell-structured data and spreadsheets
        - Dataframes and dataframe systems
        - OLAP, summarization, and visual analytics
    - Batch Analytics:
        - Compression and column stores
        - Parallel data processing and map-reduce
        - **Streaming, sketching, approximation**

Today's Lecture

- We'll start with how we can deal with large, but finite data…
- And then move to possibly infinite data

Sampling and Approximation

- When we are trying to explore large volumes of data (think TB/PB), the sheer size of the data can be a detriment to exploration
  - Imagine having to wait for hours for every query result to be returned, e.g., the total sales by category

- One common approach to address this issue is via sampling
  - Use a sample of the overall dataset to get a "sense" of the underlying patterns or trends with the understanding that the pattern or trend may be approximate

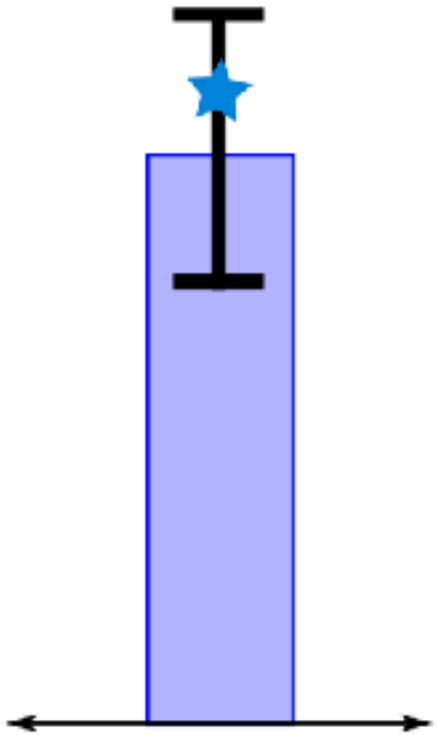
So how does sampling help us?

- If we have a query:
  - SELECT AVG(Salary) FROM Employee
  - On a table with 10^8 rows

- Then, we can use a 1% uniformly sample of 1M rows to estimate the average salary.
  - That is, each row has a 1% chance of making it to the sample, independently of other rows
  - This estimate is approximate
  - However, as the sample size becomes larger and larger, the estimate becomes closer to the actual value
    - Naturally, if it hits 100% it is equal to the actual value

How much do we believe the estimate?

- When we are computing an aggregate using a sample, we need to understand how much we can believe the estimated aggregate (or *empirical* aggregate from the sample)
- Thankfully, probability tools help us with this issue
- Usually, the degree of belief in an estimate is expressed in the form of a *confidence interval* around the empirical aggregate
  - We can make a claim that there is only a small probability that the confidence interval fails to enclose the true aggregate δ



Confidence Interval Computation

- Many many ways to compute confidence intervals
- First off, they come in many flavors
  - The conservative kind which provide true guarantees
  - The asymptotic kind which only provide guarantees "in the limit"
- The specific approach is unimportant, but here is one example

Proposition 1.2 (Hoeffding's inequality). *Let $\mathcal{X} = (x_1, \ldots, x_N)$ be a finite population of N points and $X_1, \ldots, X_n$ be a random sample drawn without replacement from $\mathcal{X}$. Let*

$$a = \min_{1 \le i \le N} x_i \quad \text{and} \quad b = \max_{1 \le i \le N} x_i.$$

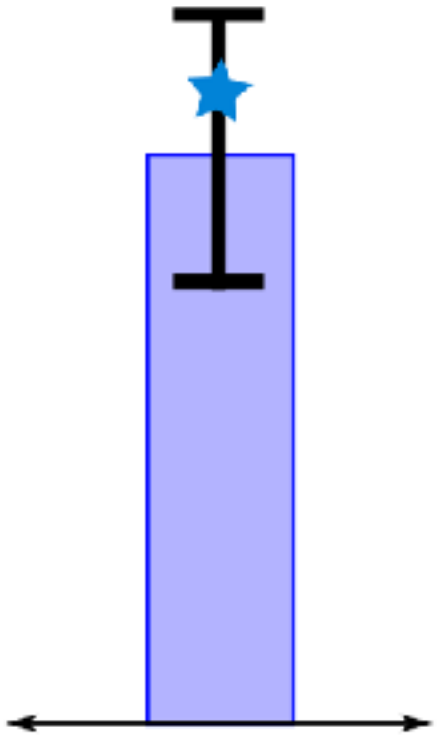
Then, for all $\varepsilon > 0$,

$$\mathbb{P}\left(\frac{1}{n} \sum_{i=1}^{n} X_i - \mu \ge \varepsilon \right) \le \exp\left(-\frac{2n\varepsilon^2}{(b-a)^2} \right), \tag{1}$$

where $\mu = \frac{1}{N} \sum_{i=1}^{N} x_i$ is the mean of $\mathcal{X}$.

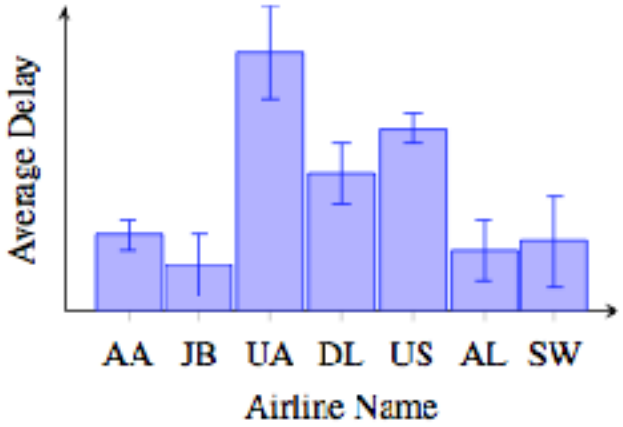
Takeaways from Hoeffding's Inequality

- The confidence interval crucially depends on
  - The number of samples drawn
  - A-priori known bounds for the values the aggregate can take
    - e.g., range of possible salaries



So what can sampling help us with?

- SELECT AGG(X) FROM R
  - Aggregates like SUM, AVG, COUNT
    - Doesn't work with MAX, MIN. Why?
- Q: What if we want to estimate a GROUP BY?
  - SELECT AVG(Delay) FROM R GROUP BY Airline
  - Valuable for visualization
  - Can we simply take a 1% overall sample?

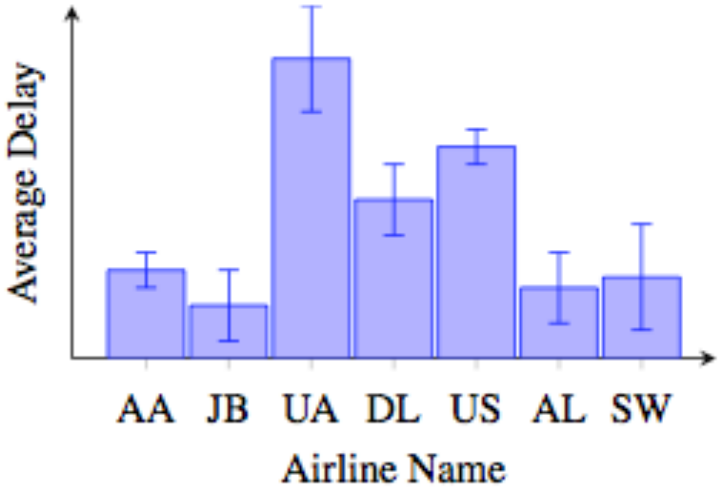


- No! Some rare groups will not be represented in the sample!

Solution: Stratified Sampling

- Ensure that a fixed # of samples are present for each group that you want to compute an estimate for.
- For example, here, ensure 10,000 samples for each of AA, JB, UA, …

- Challenge: there may be unanticipated groupings that you may want to produce visualizations for
- Approach: OLAP-style data cube materialization
  - Except that here, instead of materializing the aggregates, we are materializing a sample
  - Pick the finest-granularity that you can afford to keep samples for
  - e.g., if we may want to generate visualizations by Airline or by date or by both, we may want to ensure that there are samples for every combination of (Airline, date)

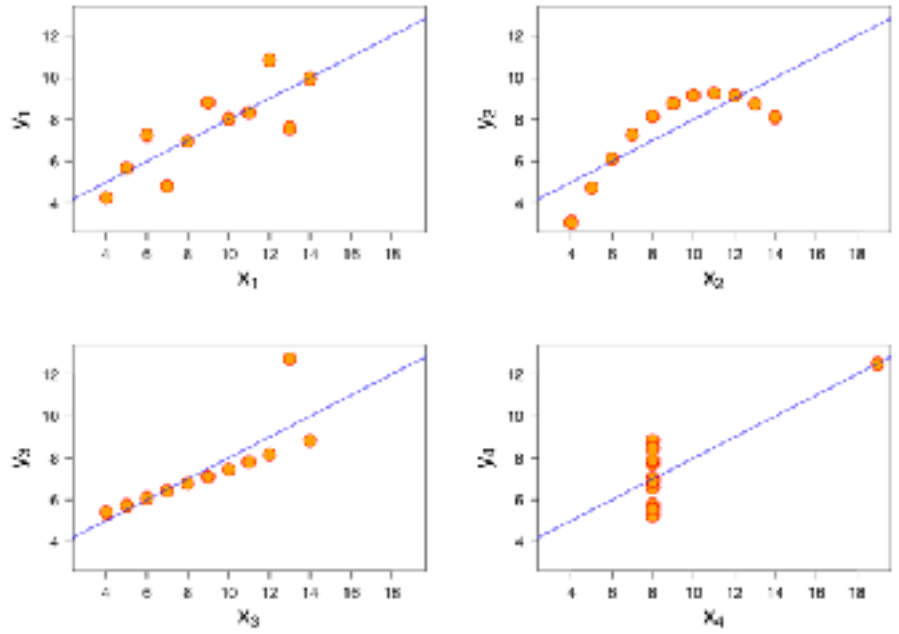


Adding on other SQL keywords…

- What about WHERE clause(s)?
  - Unless we have accounted for the same attributes as part of the cube, it is possible that we may not have enough samples that obey the where clause
  - e.g., WHERE Departure time < 03.00
- What about joins?
  - Joins are fairly tricky
  - If we took a 1% sample of R and of S
    - On average, we'll have a 0.01% sample of the join of R and S
  - In general, preferable to apply sampling to denormalized data for this reason

The Dangers of Sampling

- Sampling is good for identifying aggregated trends
- Anscombe's quartet (1975)
  - Four charts with the same
    - Mean for x, y
    - Variance for x, y
    - Correlation between x, y
    - …
- Even mean and variance obscures obvious patterns
- More generally, if we are sampling, we may miss outliers



Taking a sample…

- Say I have a 1B row CSV containing flights. I want to take a 1% sample of this CSV of 10M rows to compute the average delay. One option is to take the first 10M rows and throw out the rest.
  - Q: Is this OK?
  - A: no, because the layout may not be random
    - Specifically, if organized by day, the earliest 1% may be very different from the latest 1%

How to Perform Sampling in RDBMSs

- Two mechanisms for sampling (SQL:2003 standard)
  - SELECT * FROM R TABLESAMPLE BERNOULLI(percentage p)
  - SELECT * FROM R TABLESAMPLE SYSTEM(percentage p)
- BERNOULLI
  - Takes a p% uniformly random sample of R
- SYSTEM
  - Samples a number of pages to get a p% sample.
  - Pages are selected uniformly randomly, but we read all the tuples on a given page
- Downsides?
  - BERNOULLI is slower due to more random accesses; SYSTEM is faster but less "random"
- Similar ideas can be applied to other systems, e.g., a 1% sample on MapReduce or Spark or on Dataframes

So far, we've seen sampling on bounded data…

- What if the data is truly unbounded?

- Enter streaming systems
  - Streaming systems make no assumption about the finiteness of underlying data
  - In one sense, batch systems are one extreme within streaming S

Some material drawn from Streaming Systems, Akidau et al.
+ lecture slides on Data Stream Processing Systems by Kalavri

Examples of Streaming Scenarios

- IOT sensor measurements
- Internet page visit logs, search queries, …
- Financial transactions
- Traffic records

Characteristics of Data Streams

- Data arrives continuously
  - There is no real "ending"
  - Often can't store everything
- There is no control over how and when the data is produced
  - The arrival rate is unknown
- Sequential and single pass
  - Can look at each data item "once"

Simple Streaming Query

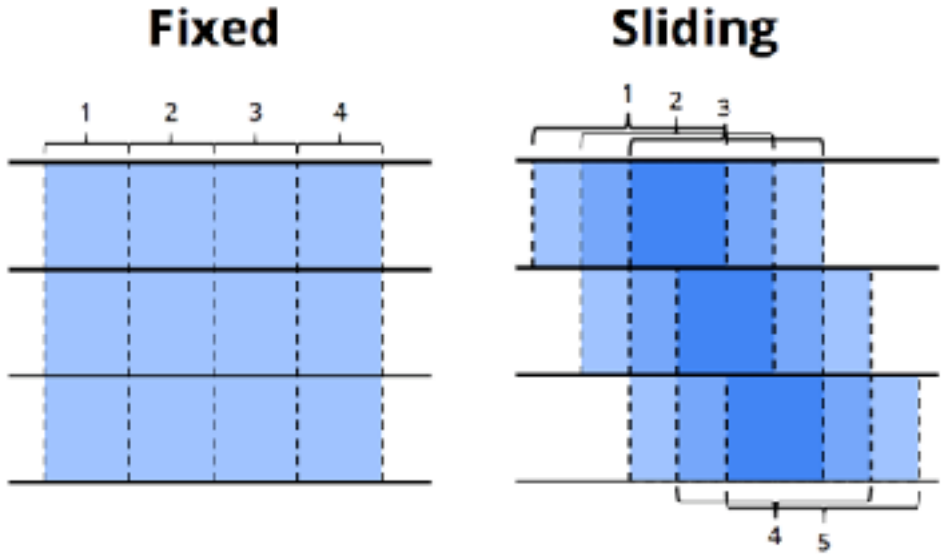
- Say we have a number of temperature measurements coming in, and we want to keep a track of the overall average

- Simple approach to do this: maintain:
  - A count of # of measurements seen so far, and
  - The total sum of measurements seen so far
- This is quite compact: much smaller than the total # of measurements
- For each new measurement, we modify the count and the sum
- Average at any point can be computed by taking sum and dividing by count

- Think of this as a continuously maintained materialized view
  - In the streaming literature, this is called a *standing query* or a *continuous query*

One Step Further

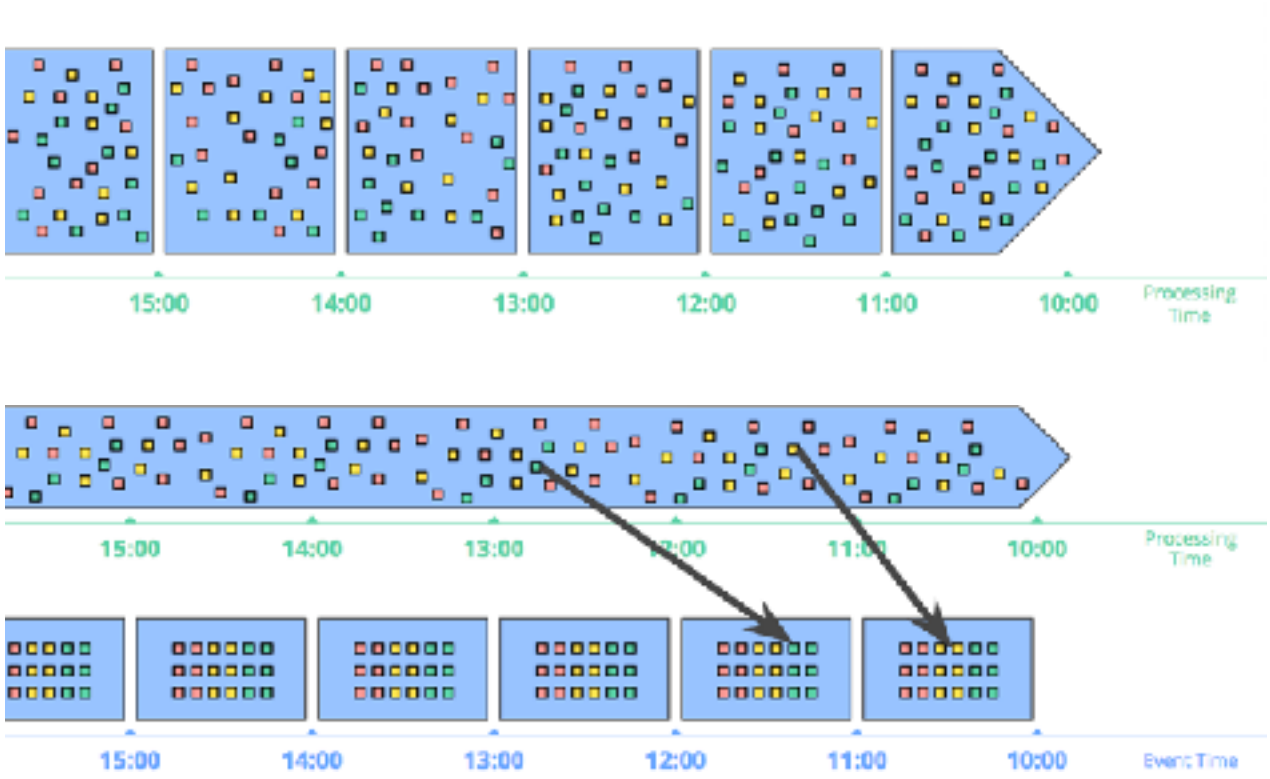
- Often average across all time of temperatures is not very useful…
- Maybe we may want to understand the recent "trend"
- We analyze data at the granularity of *windows*
  - Fixed windows: e.g., every hour
  - Sliding windows: e.g., at time t, provide the average between [t-1, t)
- Q: How do we compute average temperature over time for fixed or sliding windows?

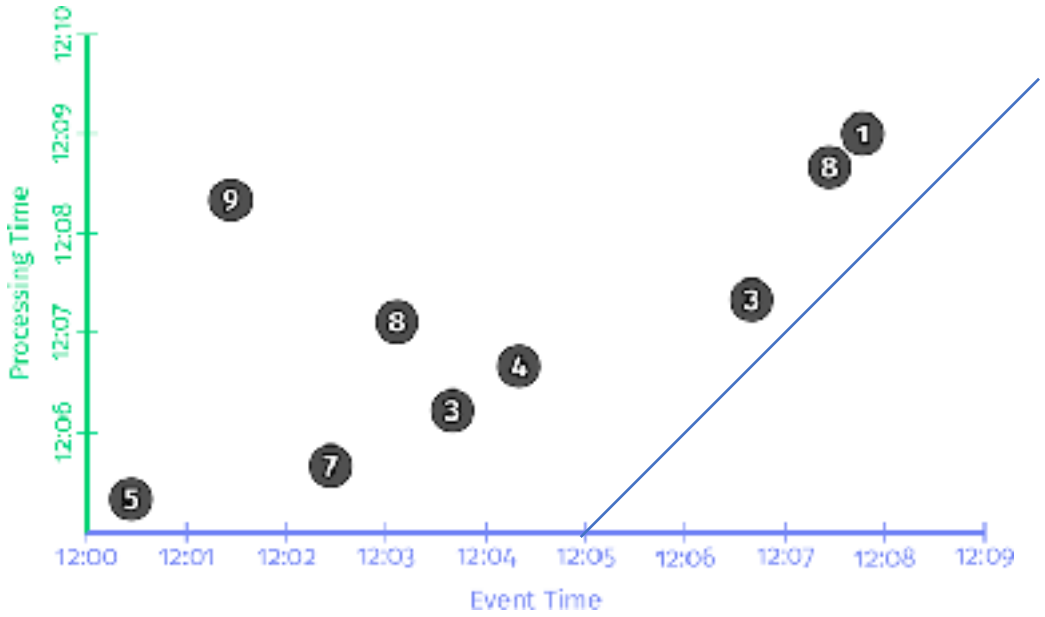


Timestamps

- Data has an *event* timestamp (when it happened) and a *processing* timestamp, which is when the record is processed by the system
- So far, we have assumed that the event time = processing time
- This may not be true in general
  - For example, some tuples may be delayed because of network delays or outages
  - In such a case, the processing time may be >> event time
- If our fixed/sliding windows are defined by processing times
  - e.g., provide the average of the last *k* temperatures received
  - Then it is easy: simply maintain the last *k* temperatures
  - Often, however, the windows are more meaningfully defined by event times

Event and Processing Times

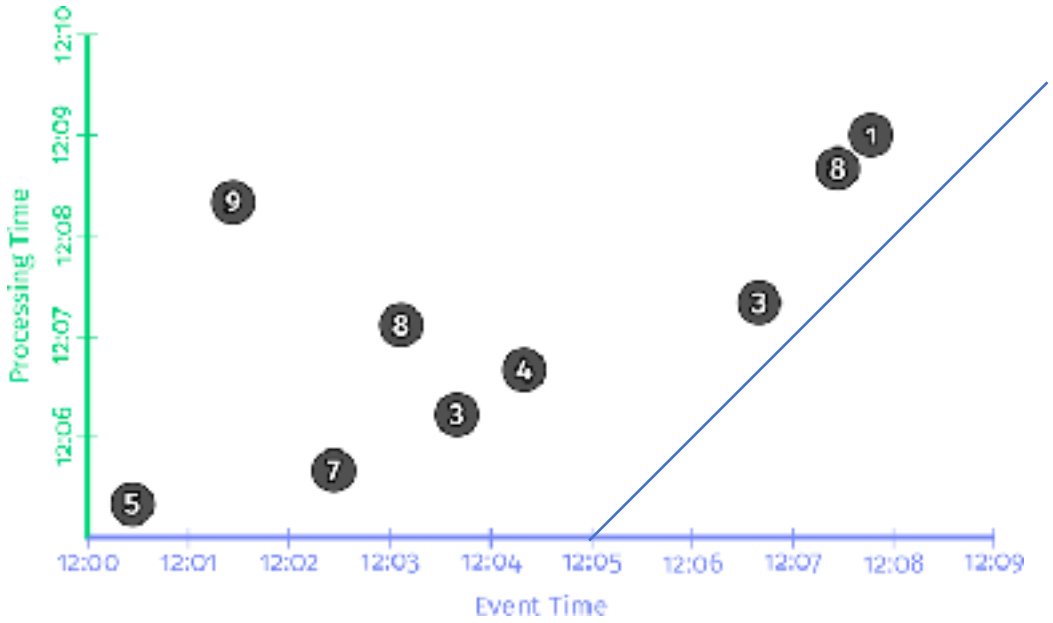




How do we handle these discrepancies?

- Triggers!
  - We update materialized views based on triggers
  - Say we want to do a fixed-window on events
    - e.g., average value for every 2 minutes
  - We have various materialized aggregates, one per 2 minute interval
  - These aggregates can be updated as new data comes in

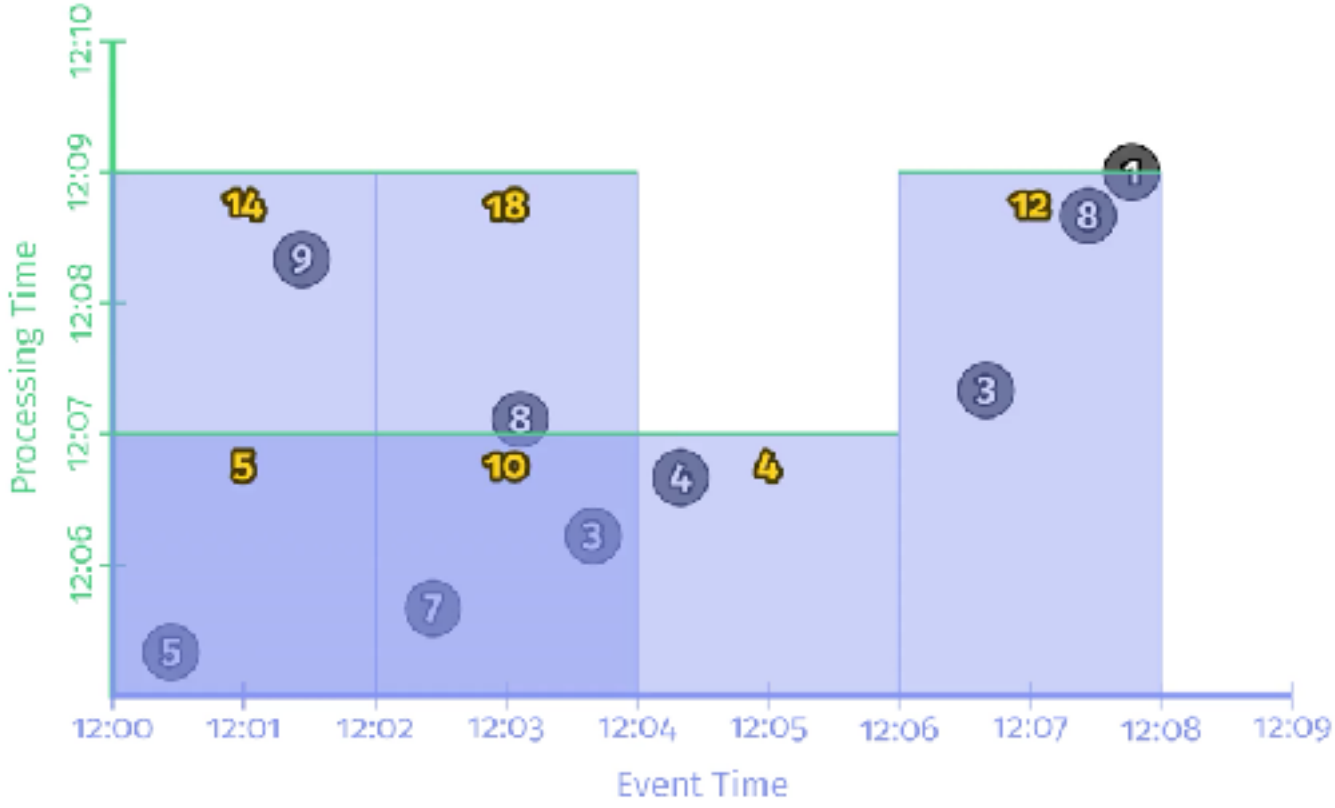
- Two forms of triggers:
  - Periodic updates
  - Completeness-based



Periodic Update Triggers

- One aggregate per 2 minute event window
- Updated every two minutes of processing
- Here, showing the total per event window
  - Purple means it is "published"
  - e.g., an output tuple is produced

- Can instead do the update based on time elapsed from when the last record was processed per window
  - If enough time has elapsed, can "publish" a new updated materialized result

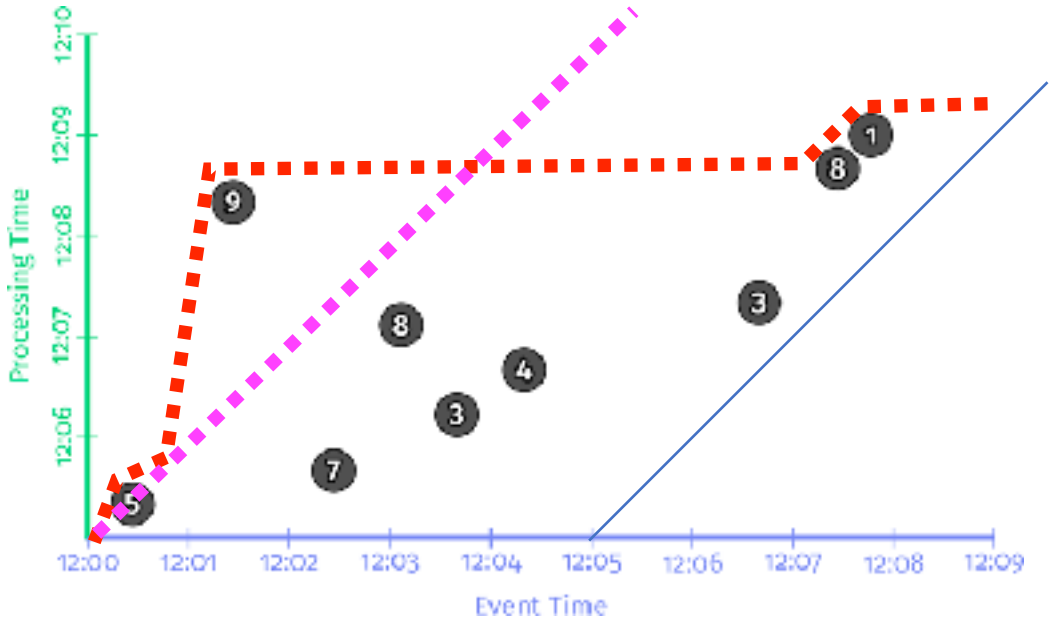


Downsides

- We don't know when we're done for event windows in the past
  - A new tuple can come that is extremely delayed and impact an aggregate very long ago
  - Also, we don't know if we should stop maintaining aggregates for such windows
- To deal with this, we need some understanding of how to relate the event time with the processing time
- Enter completeness-based triggers

Completeness-based Triggers

- We can set a cutoff wherein we believe we know all prior data should have arrived by that point
  - For example, we can say that tuples will arrive for processing at most 1 hour after the event
  - This functional relationship between event time and processing time is known as a *watermark*
  - This can be a heuristic, or could be accurate
  - Red: a perfect watermark
  - Purple: a heuristic watermark
    - Q: what does this watermark say (in words)?

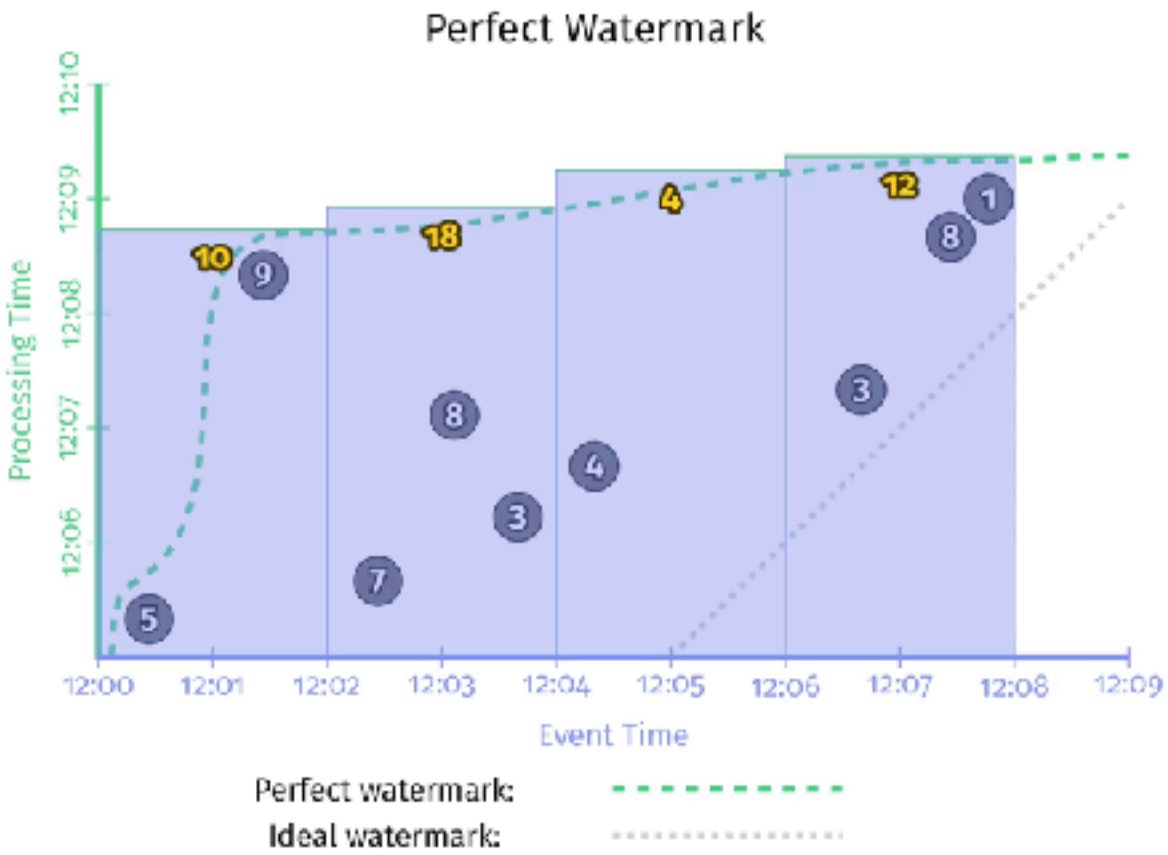


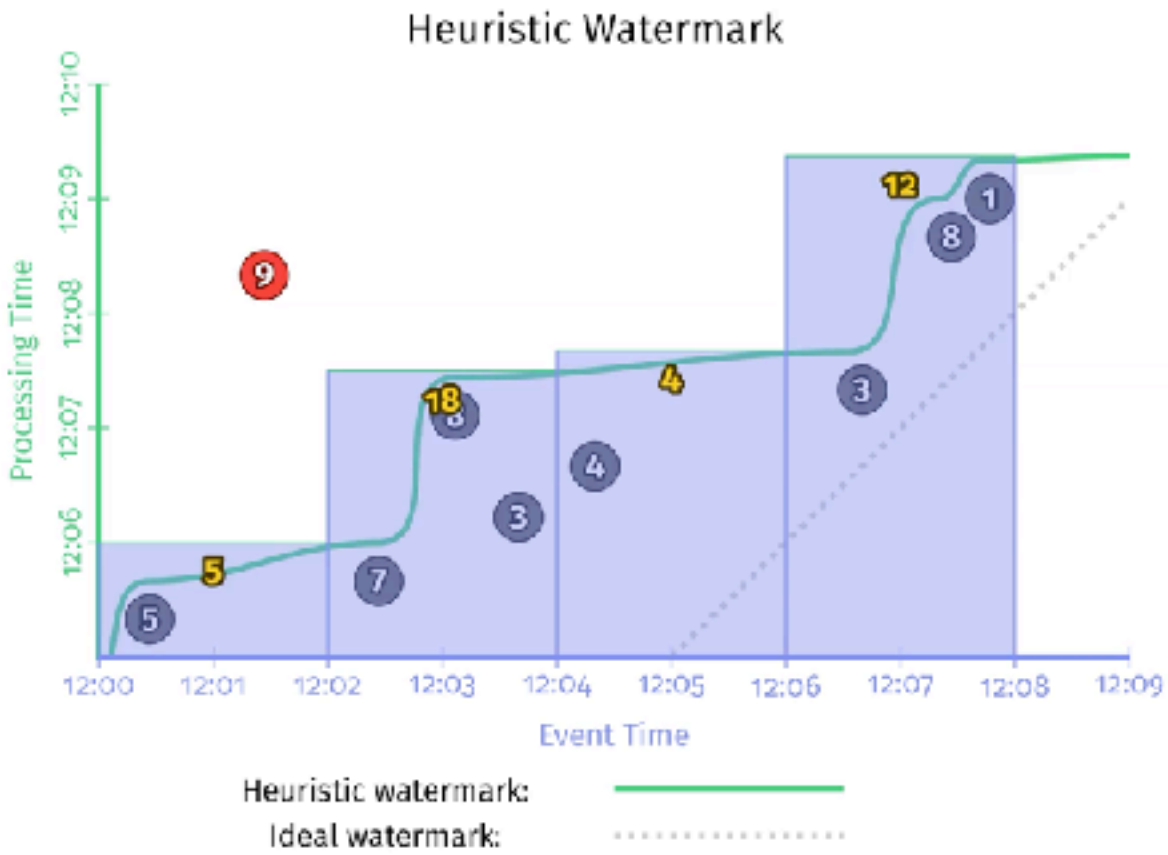
How do we use watermarks?

- For every window, if we have hit the watermark, we can use a completeness trigger to "publish it"
  - with the understanding that that window will not be changed in the future

- Downsides of watermarks:
  - Perfect watermarks may end up being too conservative
  - Heuristic watermarks may end up missing out on data or being too conservative

Example


Perfect Watermark
Processing Time
12:00
12:01
12:02
12:03
12:04
12:05
12:06
12:07
12:08
12:09
12:10
Event Time
Perfect watermark:
Ideal watermark:


Heuristic Watermark
Processing Time
12:00
12:01
12:02
12:03
12:04
12:05
12:06
12:07
12:08
12:09
12:10
Event Time
Heuristic watermark:
Ideal watermark:

How do we choose?

- To get the best of both worlds, we can use both periodic and completeness-oriented triggers
  - Publish results every k minutes, while also providing an indication when the results are likely to be "complete" for a given window

- What about storage? Options:
  - We maintain all statistics per window, providing up-to-date statistics per window every time it is published
    - Downside: a lot more storage since we don't know when to delete old window data
  - We maintain only statistics from the last time the results were published
    - This is a "delta" from the previous
    - e.g., we produce the sum of unpublished values and their count from the last time
    - Downside: the "client" needs to do more work

All this for a simple aggregate query!

- What happens for
  - Filters/projections?
    - Can simply drop tuples that don't match/non-relevant attributes before processing
  - Joins (on time)?
    - Can maintain tuples for one table in memory until we know that we're sure that matching tuples from the other table will no longer arrive (a watermark)
    - General joins are a LOT harder to do
  - Grouping (apart from time windows)?
    - Can maintain separate aggregates for each group

Streaming Systems

- Still very much an open playing field (most systems below are <5 years old)
- Some example systems:
  - Storm
  - Heron
  - Spark Streaming
  - Millwheel
  - Flink
  - Google's Cloud Dataflow
  - Beam (connects to many systems above)
- Research projects: STREAM, TelegraphCQ, Niagara, …
- The concepts I introduced are based on Beam

Some syntax (from Apache Beam)

- For the periodic update triggers:

```
return input
      .apply(Window.<KV<String, Integer>>into(FixedWindows.of(TWO_MINUTES))
      .triggering(Repeatedly.forever(AfterProcessingTime.pastFirstElementInPane().alignedTo(TWO_MINUTES, …))))
            .withAllowedLateness(Duration.standardDays(1000)))
      .apply(Sum.integersPerKey())
```

- For the combined completeness+periodic update triggers:

```
return input
      .apply(Window.<KV<String, Integer>>into(FixedWindows.of(TWO_MINUTES))
            .triggering(AfterWatermark.pastEndOfWindow()
                  .withEarlyFirings(AfterProcessingTime.pastFirstElementInPane().plusDelayOf(ONE_MINUTE))
                  .withLateFirings(AfterPane.elementCountAtLleast(1)))
            .withAllowedLateness(Duration.standardDays(1000)))
      .apply(Sum.integersPerKey())
```

Combining with Sampling

- Can drop tuples as they are streaming in based on some criteria, ensuring that there are enough tuples left "per strata"
- Challenge is ensuring that we have a uniformly random sample

- One example algorithm: Reservoir Sampling

- Say we have space for k items, and we want a random sample of n items where n is the number of items seen so far

How would we do this?

- OK, let's say we have a k element uniform sample from the first n elements.
- Now, we observe the n+1th element
- We want to update our k element uniform sample to be from the n+1 elements
- Q: How would we do this?

- A: we keep the (n+1)th element with probability k/(n+1)
- If we are keeping it, then with pick one of the k elements at random to evict, each with probability 1/k
- So, for the k elements, whose probability of being in the sample in the previous round was k/n, this round will have probability

$$\frac{k}{n} \cdot \left(\frac{k}{n+1} \cdot \frac{k-1}{k} + \frac{n+1-k}{n+1}\right) = \frac{k}{n+1}$$

Takeaways

- For sampling:
  - Important when data is finite but large
  - Powerful but sometimes deceptive
  - Need to ensure enough samples "per strata"
  - Ensuring uniform randomness is often at odds with the overall objective
    - Page-based sampling may be an OK compromise
- For streaming
  - Important when data is unbounded: look at each item precisely once, with limited ability to store
  - Windowing as a means to study data based on time
  - Triggers to update materialized aggregates on a periodic or watermark basis
  - Still very much an area in flux!