CS 686: Special Topics in Big Data

Summarizing and Sampling Streams

Lecture 25

Today’s Schedule

- Project 2 Updates
- An Intro to Sketching Big Data
- Running Statistics
- Running Samples

Today's Schedule

- **Project 2 Updates**
- An Intro to Sketching Big Data
- Running Statistics
- Running Samples

Project 2: Averaging

- A quick tip about reporting feature values: it's often better to report an average rather than a total
  - Both **can** make sense
  - But an average (mean) is often more flexible
- For example: you may not have data for all days of a month
  - And some months are shorter than others (not by much, but you get the idea!)

Project 2: Readme Files

- Before grading your project, I'll take a look at your readme to decide which parts we'll go over
- Provide lots of detail and explain the patterns/phenomena you're seeing
- The more images, the better!
  - You can take screenshots from geohash.org, weather websites, etc.

Today's Schedule

- Project 2 Updates
- **An Intro to Sketching Big Data**
- Running Statistics
- Running Samples

Streaming Big Data

- In many cases, data is produced much faster than we can analyze it
- Batch processing systems like MapReduce let us do analysis offline, after the fact
  - Good: studying long-term trends
  - Bad: reacting quickly…
    - Health monitoring, rerouting traffic, etc.
- We can use a stream processing system, but what happens when that can't handle the workload?

Sketching

- Rather than storing/processing everything, we can build **sketches** of the datasets
- Some information is thrown away…
- …but we can store a wider breadth of information.
- These approaches have memory and processing benefits
  - Also well-suited to IoT devices, low-powered cloud instances, etc

A Few Types of Sketches

- Dimensionality reduction
  - Perhaps a dimension (feature) in our dataset can be expressed as a function of another dimension
- Wavelets
  - Used in signal processing
  - Does a particular wavelet correlate with the signal we are examining?
- **Summarization**

Scenario

- Let's assume we have a data feed from NOAA that looks just like our NAM dataset, but in real time
- We want to provide some basic statistics about the weather
  - Highs, lows, averages, etc.
- If we store these in an array (or similar structure) then we can easily find the values we need
  - This will consume a lot of memory (or disk space)

Today's Schedule

- Project 2 Updates
- An Intro to Sketching Big Data
- **Running Statistics**
- Running Samples

Optimization

- Being the clever big data people we are, we realize that providing the highs/lows doesn't actually require us to store the entire data stream
- We can just check whether the new value we've seen is larger/smaller than what is recorded
- Great! But now we also want to know what the average temperature is…

Gathering More Statistics

- To improve the expressiveness of our weather reports, we also want to gather:
  - Total number of data points
  - Average, variance, and standard deviations
- These statistics provide a high-level overview of the data distributions
  - For instance, if we can assume a **normal** distribution then this tells us a lot about the data

Online Statistics Collection

- Since new records are constantly streaming into the system, recalculating statistics each time is inefficient
- We also operate in a distributed world: what if multiple nodes in our cluster are receiving data points at the same time?
- Solution: *Online* statistics collection via ***Welford's Method***

Welford's Method

- Allows statistics about a dataset to be updated incrementally
  - Computation is performed in a single pass (each data point is inspected **once**)
- Each new record incurs a small calculation cost, but avoids re-calculating statistics for the entire dataset
  - Takes about 1 microsecond (0.000001 second) on commodity CPUs

Welford Implementation

- We'll maintain:
  - the number of observations, *n*
  - the running mean, $\bar{x}$
  - the sum of squares of differences from the current mean, *Sn*
- As a recurrence relation:

$$\bar{x}_0 = 0, S_0 = 0$$

$$\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n}$$

$$S_n = S_{n-1} + (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n)$$

Or, in Code

```java
/** Add a new sample to the running statistics. */
public void put(double sample) {
    n++;
    double delta = sample - this.mean;
    this.mean = this.mean + delta / n;
    this.Sn = this.Sn +
       delta * (sample - this.mean);

    min = Math.min(this.min, sample);
    max = Math.max(this.max, sample);
}
```

Variance / Standard Deviation

$$\bar{x}_0 = 0, S_0 = 0$$

$$\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n}$$

$$S_n = S_{n-1} + (x_n - \bar{x}_{n-1})(x_n - \bar{x}_n)$$

$$\sigma^2 = \frac{S_n}{n} \qquad \sigma = \sqrt{S_n/n}$$

Additional Statistics

- We can use this information to perform t-tests, check the probability of values given a distribution, and more
- Another big benefit: these statistics can be **merged**
  - Collect data points on each machine in our cluster, merge them back together!
- Works well with streaming systems **and** MapReduce

Memory Impact

- We end up maintaining:
  - Min
  - Max
- And:
  - Count (n)
  - Mean
  - Sn
- In Java, we're looking at around 50 bytes or so

Pushing it Further

- This approach works well for inspecting a single feature such as temperature
- We can also maintain 2D online statistics:
  - put(temperature, humidity)
- Here we maintain the differences in the sum of squares across the two features
  - Keep a 1D instance of each feature plus this information (just ~8 more bytes)

2D Statistics

- Maintaining the 2D relationships between variables gives us:
  - Correlations
  - Slope and intercept for linear regression
  - Calculation of statistical significance
- These take milliseconds to compute and consume minimal memory

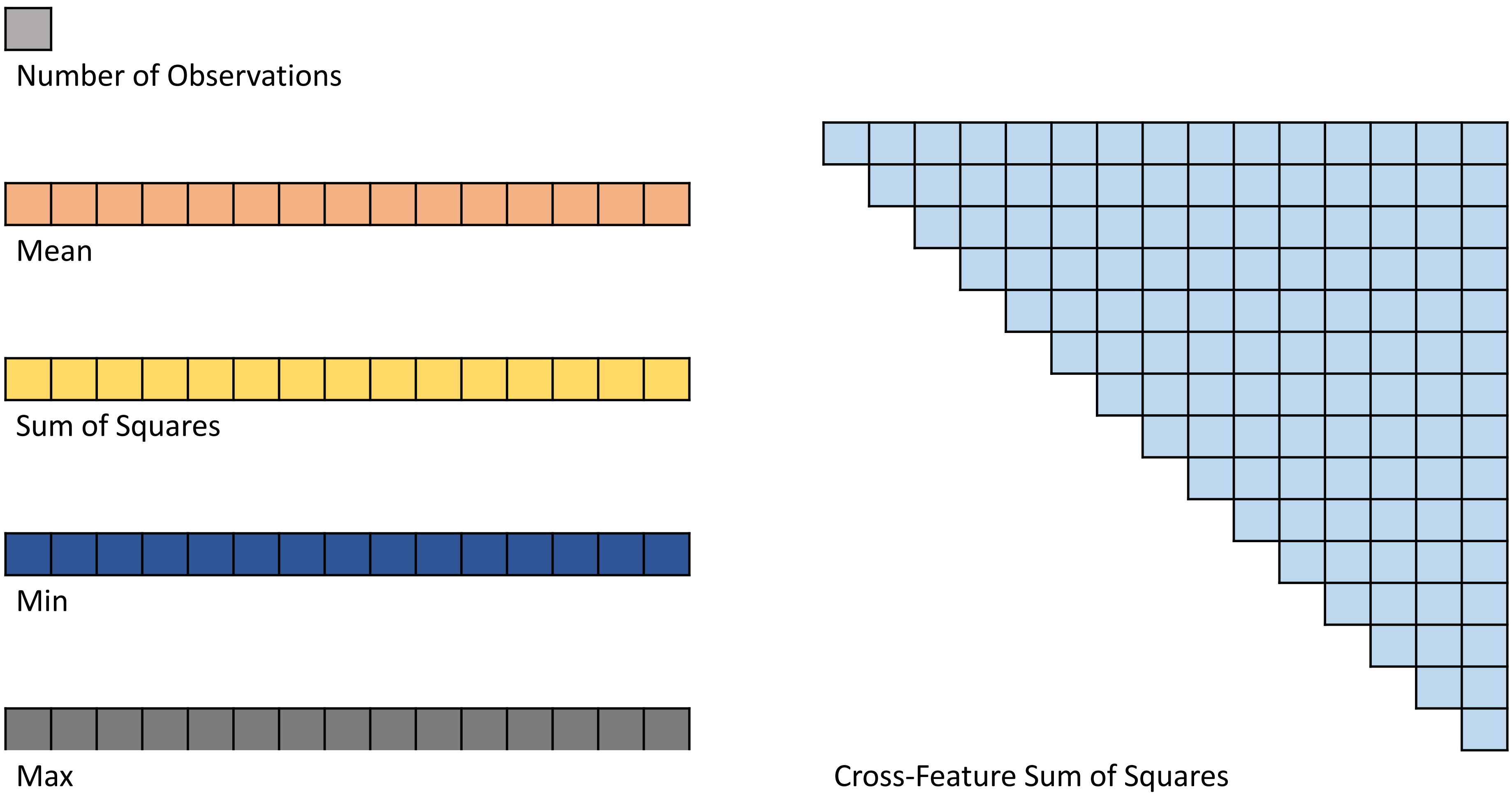
2D Summary Matrix

- After creating our 2D summaries, we can put them in a *summary matrix*
- Each feature combination ends up being represented twice:
  - Temperature → humidity
  - Humidity → temperature
- Additionally, each 1D instance contains duplicate information:
  - Number of samples seen
  - Mean value

Optimized 2D Summaries

- Instead of maintaining an entire statistics matrix, place summary instances in a triangular matrix
- Further, remove all duplicate data from the 1D instances
- This creates a new summary structure for any number of dimensions while reducing memory consumption by about 40%!

Data Structure (15 features)



Today's Schedule

- Project 2 Updates
- An Intro to Sketching Big Data
- Running Statistics
- **Running Samples**

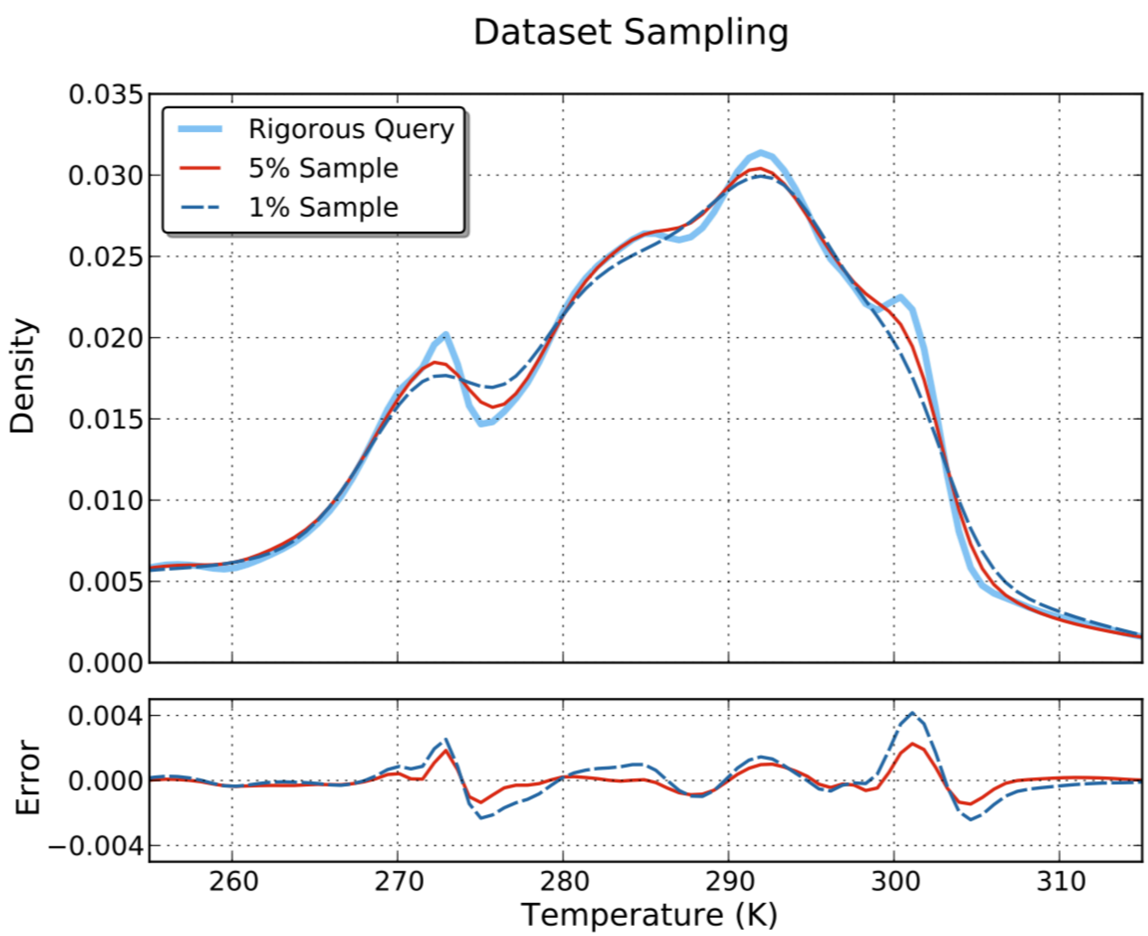
Sampling

- Welford's method is great if we want to throw away **everything** and just keep some stats
- In many cases, we'd like to actually look at raw data points as well
- Instead of storing everything, let's take a sample

Basic Random Sampling

- Take an array of N elements and a sample fraction F, for example 0.3 (30% sample)
- Randomly select N * F items from the array
- This can be done with or without **replacement**
  - Putting each selected element back into the array, allowing them to be drawn multiple times
- Great, except once again we need the whole array

Sampling our NOAA Data



Reservoir Sampling (1/2)

- When the size of the incoming stream is unknown or there are memory constraints, *reservoir sampling* allows creation of representative samples
- Set to a fixed size (array) on creation
  - Limits memory usage
- As data is streamed in, data points are placed randomly into the array
- Over time, the likelihood that incoming data will be stored in the array decreases
  - Ensures long-term representativeness

Reservoir Sampling (2/2)

- Online sampling technique that creates representative random samples when:
  - The number of incoming data points in unknown
  - The total dataset cannot fit in main memory
    - Fixed size (n)
- When data points arrive, they are assigned a random ***insertion key*** (k) in the range [0, 1]
  - If $k < n / C$, where C is the total number of observations, the data point replaces a random entry in the reservoir
  - The probability of replacement decreases over time

Reservoir Sampling Extensions

- Reservoir sampling can be augmented by allowing *sample weights* to increase the likelihood of certain data points being placed in the array
  - We may place a greater weight on samples from a particular sensor
- Additionally, storing the insertion key when placing data in the reservoir allows merging later
  - To determine which elements go in the merged arrays, just sort by insertion key

Representativeness

- While reservoir sampling provides a replacement for our standard random sampling procedure, it does have weaknesses
- The sample must fit into memory (generally acceptable)
- Outliers or uncommon values will be under-represented

Stratified Sampling

- Sometimes the outliers are actually more interesting than the common cases!
- Here, we can use **stratified sampling** to produce a sample that better represents all populations rather than just the majority
- Observe the distribution of data points, and then create sub-reservoirs across the distribution
  - Uncommon data points now have their own reservoir and won't be overpowered by the majority

Motivation: Sensor Data

- Advancements in low-power computing devices enable collection and processing of sensor data
  - Rather doing everything in a central location, ***fog nodes*** take on part of the storage and processing load
- Fog nodes can maintain a reservoir sample of the observations they record

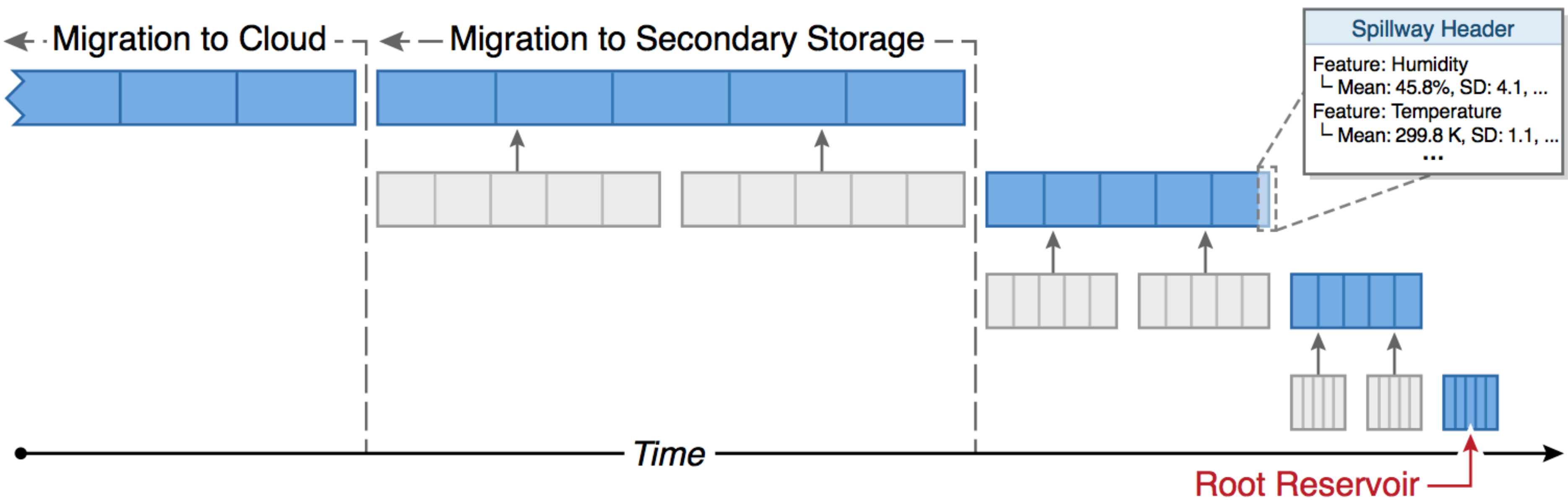
Data Insertion Workflow

1. New data points arrive and are sampled by the fog nodes
2. Data is retained in main memory and indexed to facilitate queries
3. Over time, the data precision is reduced to make space for new observations
4. Old data is migrated to secondary storage
5. Even older data is migrated to the cloud

Spillways

- Over time, the reservoirs will be updated less frequently
  - Ensures the samples are representative, but eventually the time range of data points gets very large
- ***Spillways*** are a hierarchical collection of reservoirs that vary in spatial scope
  - Each reservoir is the same size but is responsible for a different amount of data
  - Spillways ***merge*** reservoirs as they age

Spillway Structure



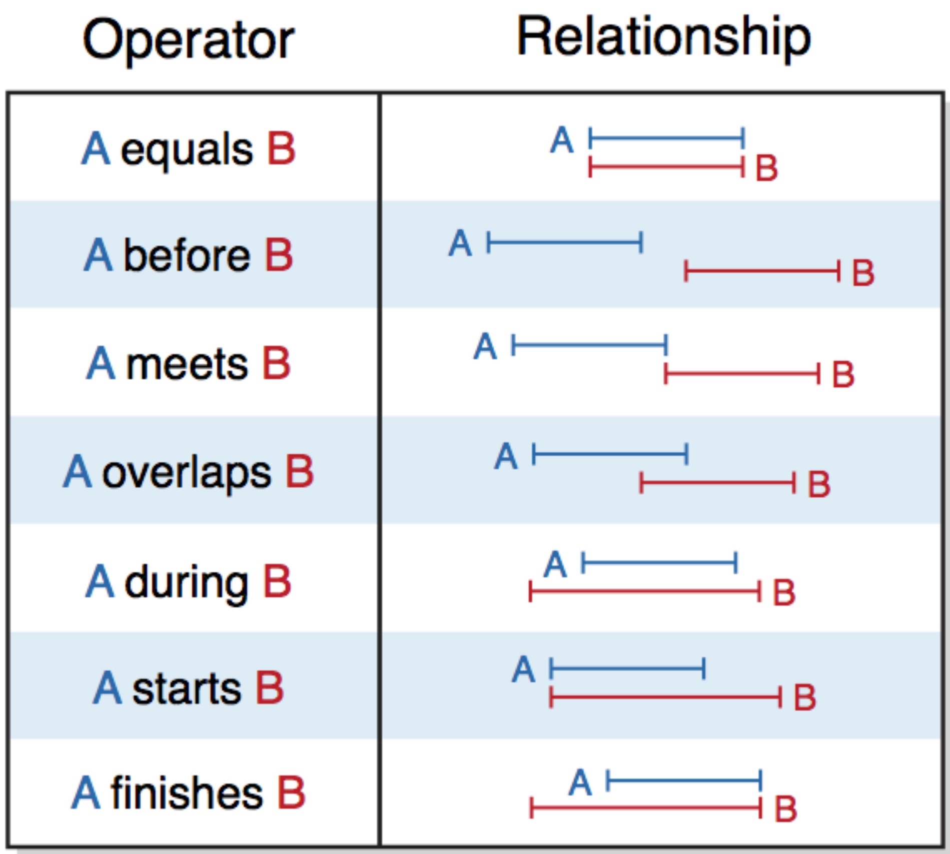
Configuration Options

- ***Temporal Curve***: how many reservoirs are at each step in the hierarchy
  - Previous example: $f(x) = 2^x$
- ***Merge Threshold***: how many reservoirs should be maintained at each level before a merge can occur
  - Default T = [2, 2, 2, 2] (four-level hierarchy)
  - Keep most recent samples: T = [5, 4, 2, 2]

Facilitating Queries

- Multidimensional data points that chosen to be placed in a reservoir are also added to a red-black tree
  - Better performance than a B-Tree for small datasets
- This allows temporal range queries and operators supported by *interval algebra*
- When a data point is replaced or its spillway is migrated, it is removed from the red-black tree

Interval Algebra Operators

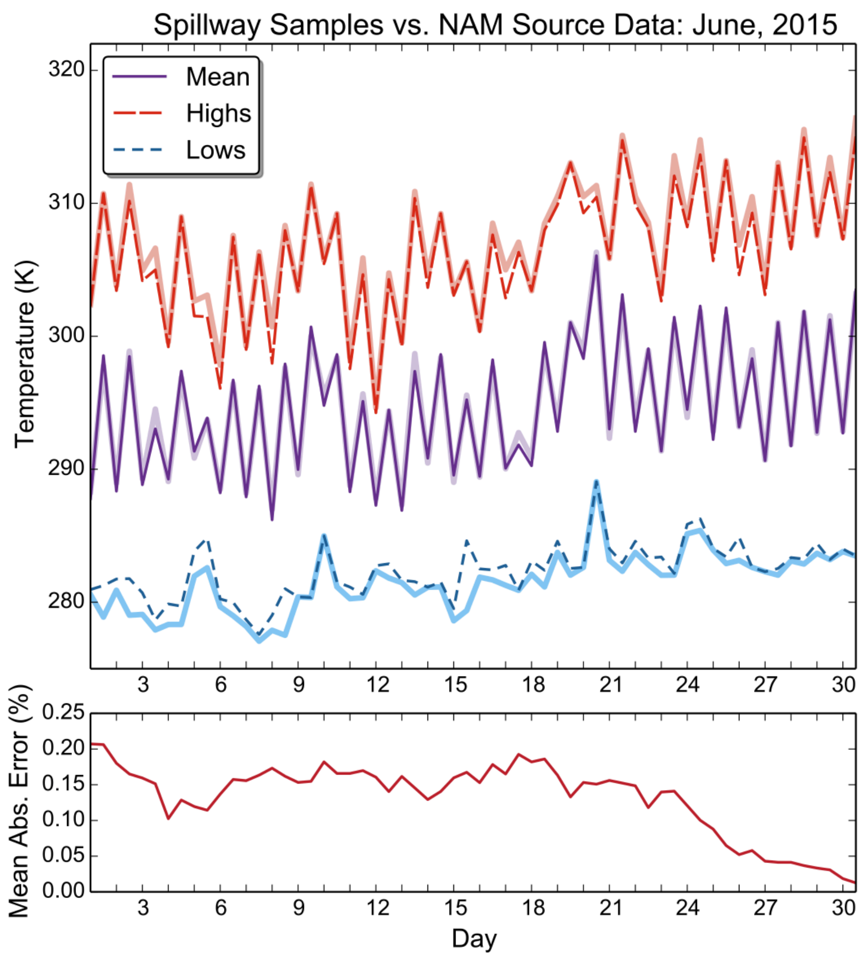


Operator	Relationship
A equals B	A B
A before B	A B
A meets B	A B
A overlaps B	A B
A during B	A B
A starts B	A B
A finishes B	A B

Evaluating the Spillway

- We can evaluate the Spillway data structure in three ways:
  1. Accuracy of the samples compared to actual data
  2. Effectiveness of coordination between the fog and cloud (what happens if relevant data is on both)
  3. Query throughput handled by the system
- Test setup: 48 fog nodes, 16 cloud nodes, and several EC2 clients

Accuracy Evaluation



Wrapping Up

- When dealing with big data, think about what you **really** need to store
  - If two features are highly correlated, it may be better to throw one away and just predict it instead
- Summarization, compression, and sketching are all good ways to make big data more manageable
- As a backup, you can always have a batch system storing full-resolution data

Wrapping Up

- Sampling can greatly improve:
  - Memory/disk consumption
  - Computation time
- Just make sure your sample is representative enough for your particular analysis!