

Introduction to Streaming Analytics

Objectives

- Streaming Optimizations
 - Recap from last lecture
- Logistics
 - Homework 2
 - Seminar
- Analytics
 - Data Preprocessing I

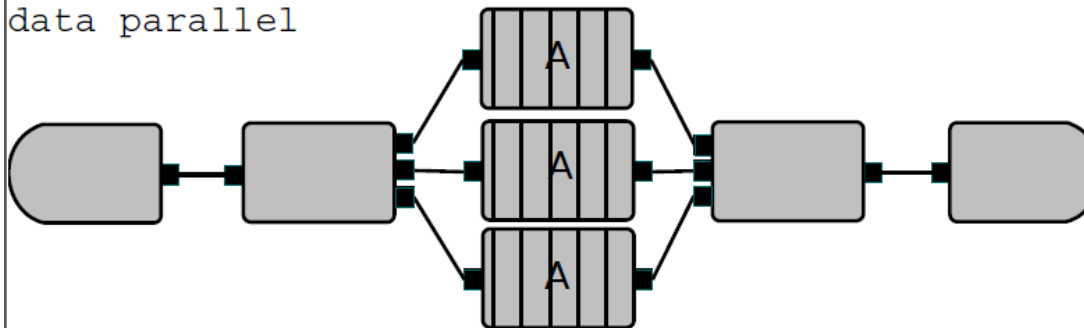
Recap: Types of Parallelism

pipelined parallel



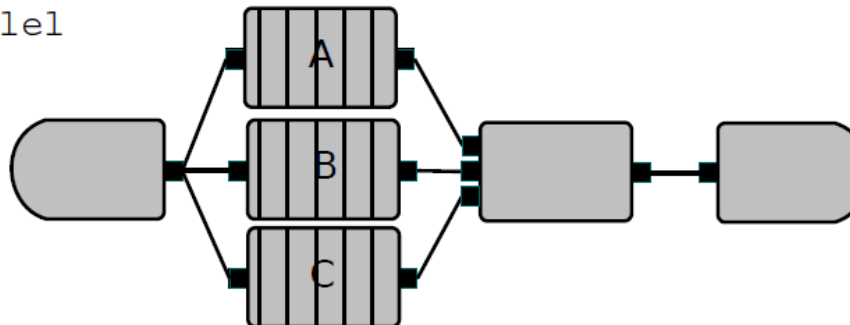
tuple 1	A	B	C					
tuple 2		A	B	C				
tuple 3			A	B	C			
tuple 4				A	B	C		
tuple 5					A	B	C	
tuple 6						A	B	C

data parallel



		time
tuple 1	A	
tuple 2	A	
tuple 3	A	
tuple 4		A
tuple 5		A
tuple 6		A

task parallel



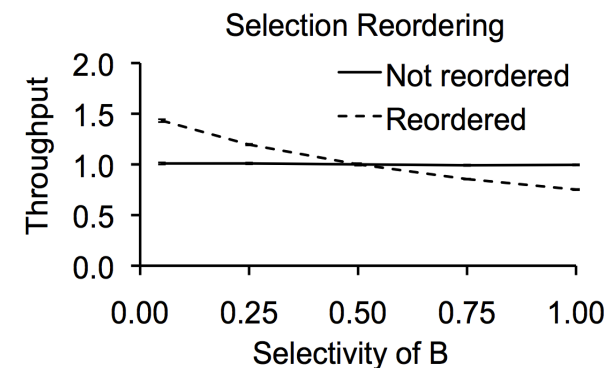
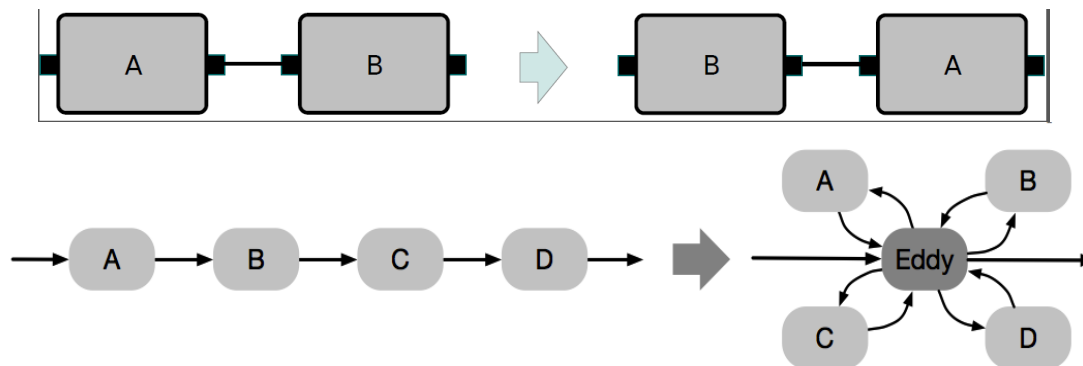
		time
tuple 1 ₁	A	
tuple 1 ₂	B	
tuple 1 ₃	C	
tuple 2 ₁		A
tuple 2 ₂		B
tuple 2 ₃		C

Recap: Kinds of Optimization

- Three dimensions
 - Does it change the graph?
 - Optimizations that involve graph transformations change the graph. E.g.: Fission
 - Some optimizations do not require transformation. E.g. Load balancing.
 - Does it impact the application semantics?
 - Ideally, an optimization should not impact the application semantics.
 - There are a few exceptions: load shedding, algorithm selection
 - Is it dynamic?
 - If the optimization can be applied at runtime, potentially based on current resource and workload availability, then it is dynamic
 - Static optimizations are often done at compile-time

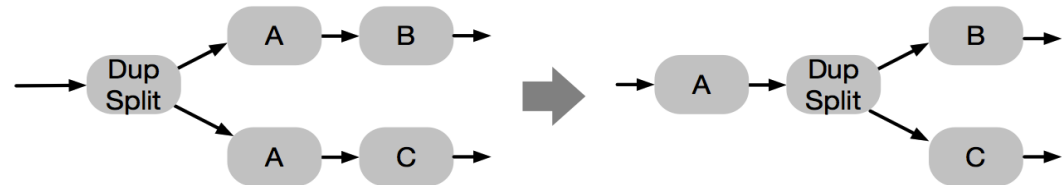
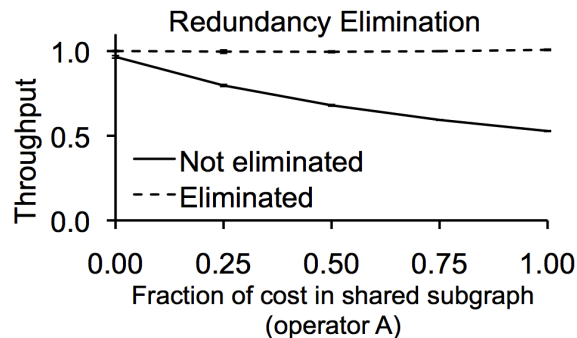
Operator Reordering

- Profitable
 - When selectivity value of second operator smaller than first
- Safety
 - Ensure commutativity



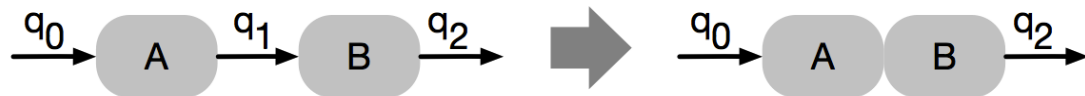
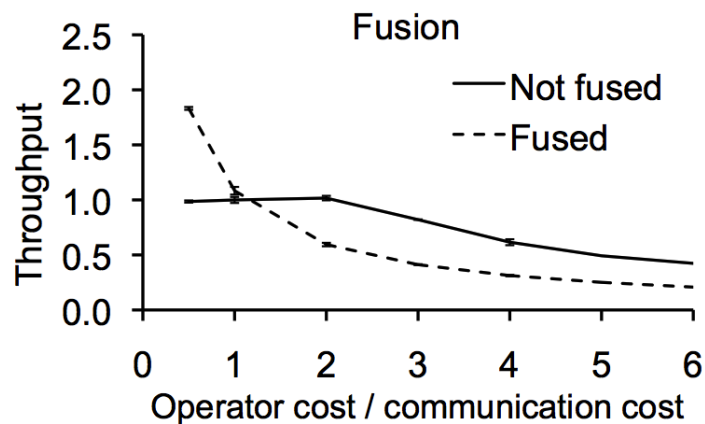
Redundancy Elimination

- Profitable
 - When cost of replicated operator is major fraction of total cost
- Safety
 - Make sure state is handled correctly, and that the operators are actually the same



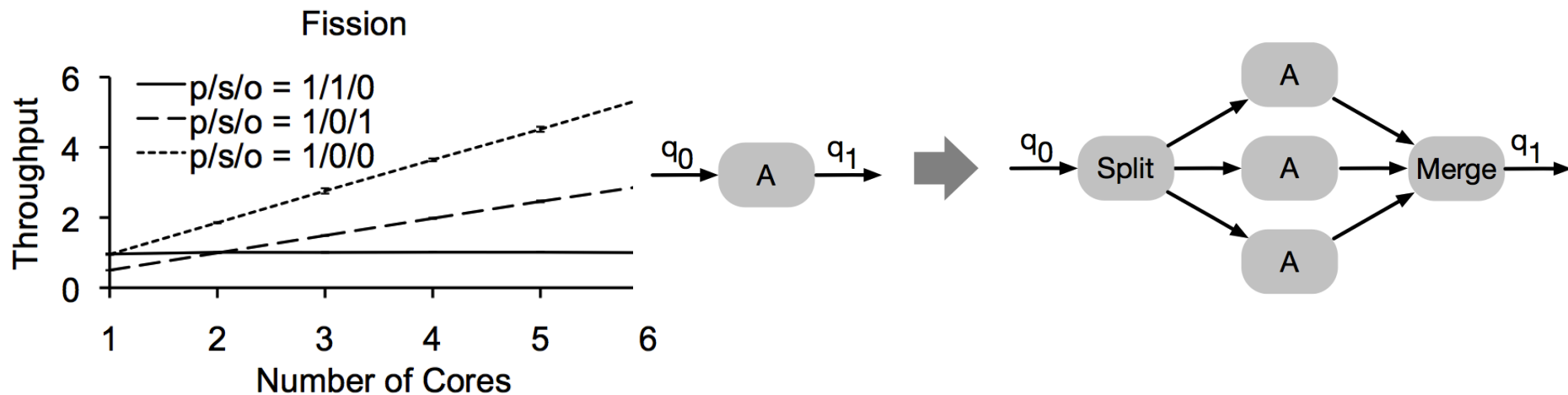
Fusion

- Profitable
 - When communication overhead is larger than processing
- Safety
 - Ensure resources are available, and have capacity



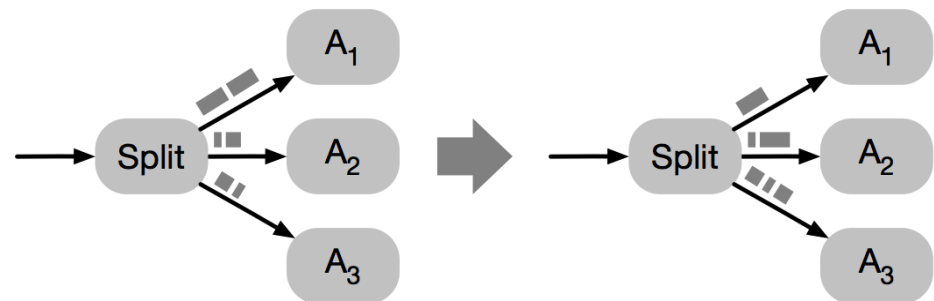
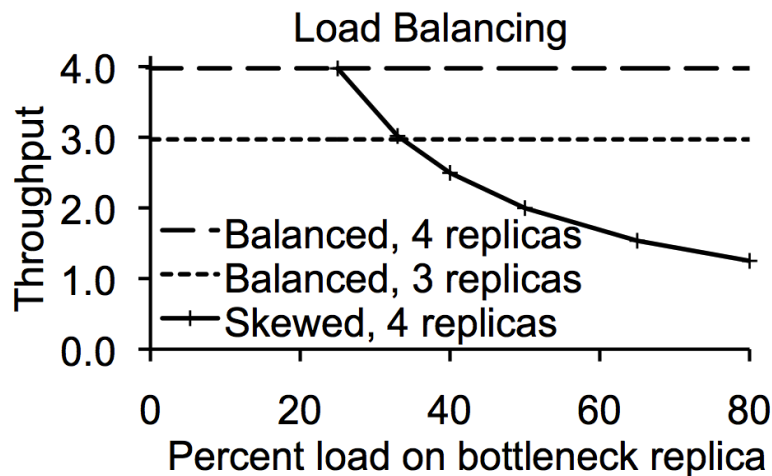
Fission: Data Parallelism

- Profitable
 - Trades split/merge overheads against resource utilization
- Safety
 - Ensure necessary resources available, stateful partitioning



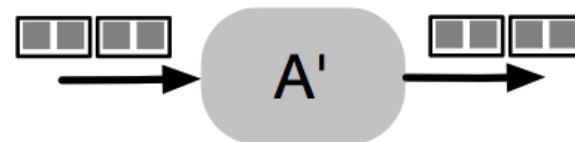
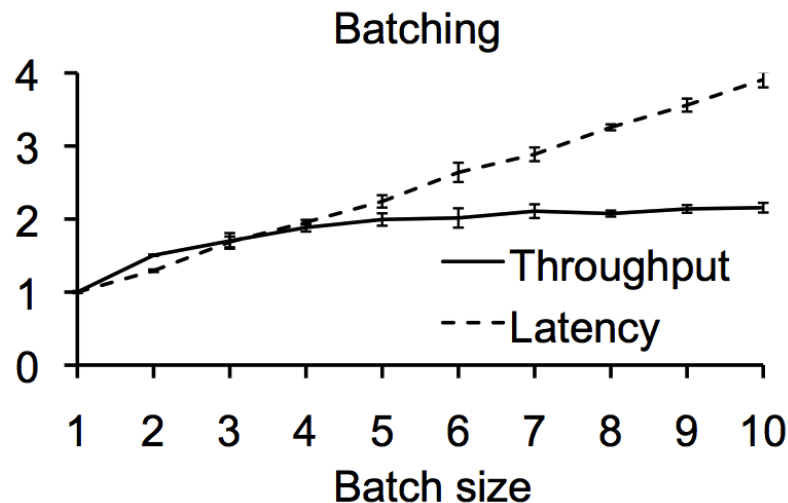
Load-balancing for Data Parallel Splits

- Profitable
 - Need to compensate for workload skew
- Safety
 - Make sure to avoid starvation. Manage state appropriately



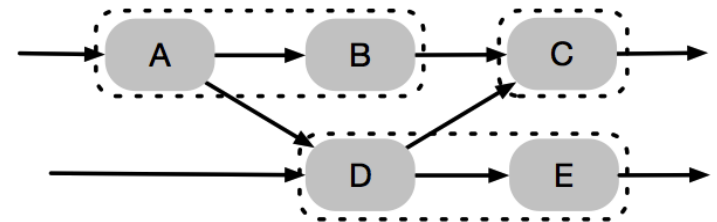
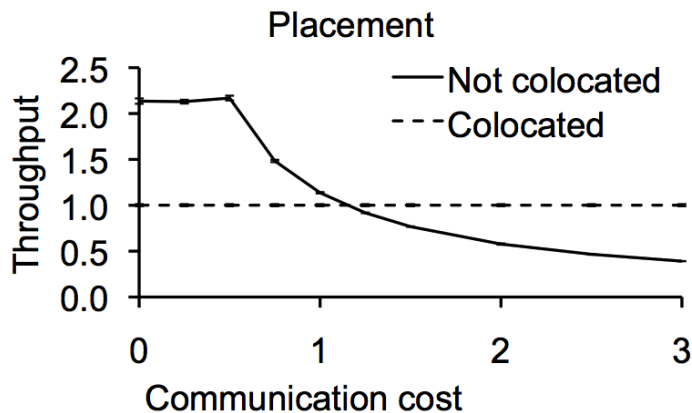
Batching

- Profitable
 - trades throughput for latency, amortizes overheads
- Safety
 - Avoid deadlocks, and satisfy real-time constraints



Placement

- Profitable
 - Trades communication cost against resource utilization
- Safety
 - Ensure resource kinds and availability



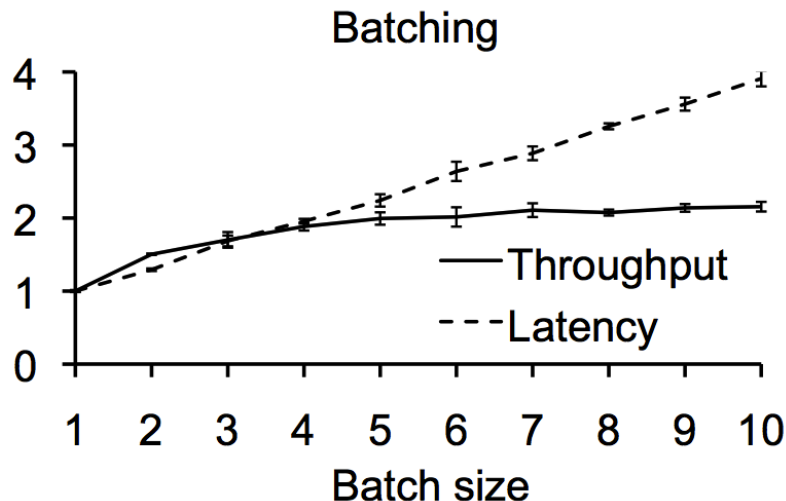
Batching

- Also known as
 - Train scheduling, execution scheduling
- *Process multiple data items in a single batch*
 - *Like RDDs*



Profitability of batching

- Batching trades throughput for latency
- Can reduce operator firing and communication costs
- The amortizable costs can include
 - Scheduling cost, context switch cost, synchronization cost, instruction cache costs, data cache costs



Assume $c(A) = c(Tuple) + c(overhead)$

With batching n tuples

$$c(A') = n \times c(Tuple) + c(overhead)$$

Amortize overhead costs

Latency: delay between when tuple arrives to when it is transmitted. *Different for different tuples in the batch.* Maximum delay scales linearly with batch size.

Safety of Batching

- Avoid deadlock
 - In the presence of feedback loops
 - In the presence of shared locks
- Satisfy deadlines
 - Real-time constraints on the per-tuple delay
 - QoS (quality of service) constraints, where delay impacts the QoS value
 - E.g. maintaining frame rates to avoid jitter in video processing

Dynamism

- The batch size can be set
 - Statically
 - StreamIt uses instruction cache vs data cache tradeoff to set the batch sizes
 - Dynamically
 - Aurora uses train scheduling to minimize context switching costs
 - Staged Event-Driven Architecture (SEDA) uses the tradeoff between latency and adaptivity to set the buffer sizes

Algorithm Selection

- Also known as
 - translation to physical query plan
- *Use a faster algorithm for implementing an operator.*

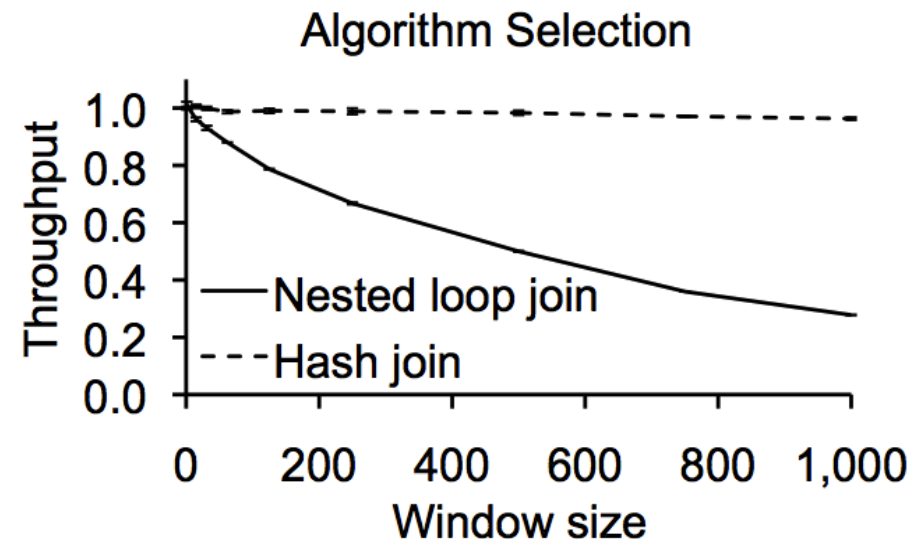
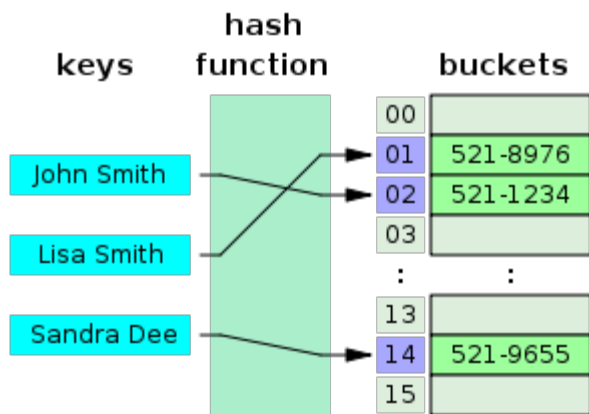


Profitability of algorithm selection

- It is profitable if a costly operator is replaced with a cheap one
- One algorithm may not be superior in all cases
 - E.g. An algorithm that works faster for small tuples vs. an algorithm that works faster for larger tuples
 - E.g. An algorithm that works faster but uses more memory.
- Example: Join
 - A hash-based implementation is almost always faster (unless the window is very small) than loop join (linear scan)
 - A hash-based implementation is suitable only for equi-joins

Profitability of algorithm selection

- Loop join: For each tuple iterate through the tuples in window of other stream to find match
- Hash join: Create a hash table for the stream window. Apply hash function to new tuple on other stream to find match



Aside on Hash Functions

```
function Hash(key)
    return key mod PrimeNumber
end
```

Additive Hash

```
ub4 additive(char *key, ub4 len, ub4 prime)
{
    ub4 hash, i;
    for (hash=len, i=0; i<len; ++i)
        hash += key[i];
    return (hash % prime);
}
```

Rotating Hash

```
ub4 rotating(char *key, ub4 len, ub4 prime)
{
    ub4 hash, i;
    for (hash=len, i=0; i<len; ++i)
        hash = (hash<<4)^(hash>>28)^key[i];
    return (hash % prime);
}
```

Bernstein's hash

```
ub4 bernstein(ub1 *key, ub4 len, ub4 level)
{
    ub4 hash = level;
    ub4 i;
    for (i=0; i<len; ++i) hash = 33*hash + key[i];
    return hash;
}
```

If your keys are lowercase English words, this will fit 6 characters into a 32-bit hash with no collisions

Several cryptographic hash functions, e.g. SHA2

Safety of algorithm selection

- Safety
 - Ensure same behavior
 - Be aware of variations
- Variations
 - Physical query plans
 - Auto-tuners
 - Empirical optimization
 - Different semantics
 - Load shedding via cheaper but less accurate algorithms

Dynamics of algorithm selection

- Dynamism
 - Algorithm selection can be dynamic
- Both algorithms are provisioned and the right one is selected at runtime
 - Performed via dynamic routing
- Change algorithm operation via parameter choice
 - E.g. size of ensemble (IMARS example)

Load-shedding

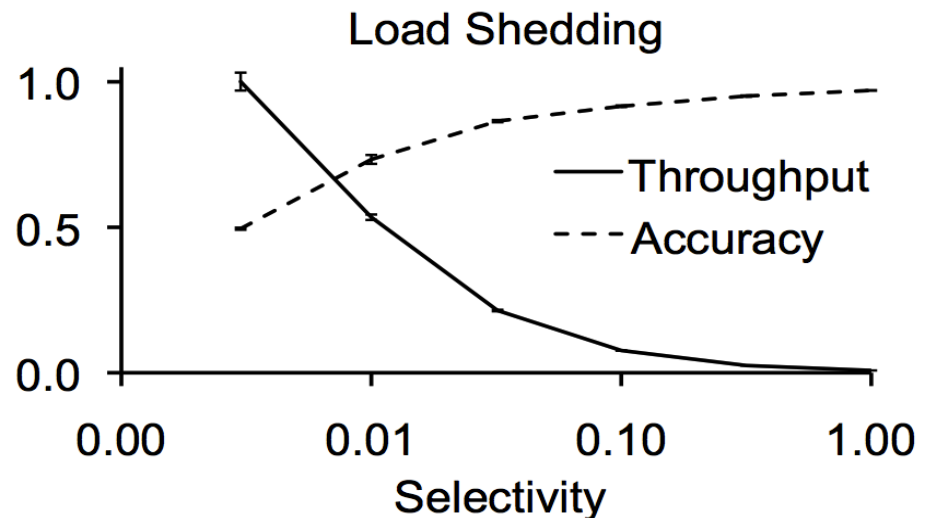
- Also known as
 - Admission control, graceful degradation
- *Degrade gracefully when overloaded.*



Profitability of load-shedding

- Load shedding improves throughput at the cost of accuracy
- Consider an aggregator-like algorithm that constructs a histogram
 - Sampling can be used effectively
 - Reducing the rate to one tenth may have a negligible impact on the histogram accuracy

Euclidean distance between histogram at full accuracy, and histogram at reduced accuracy



Safety and Dynamism of load-shedding

- Load shedding, by definition, is not safe
 - Ideally, the reduction in the quality should be acceptable
 - Major area of research
- Load shedding is always dynamic
 - Change the selectivity based on current load

Recap Optimization Catalog

Optimization	Graph	Semantics	Dynamics
Operator re-ordering	changed	unchanged	(depends)
Redundancy elimination	changed	unchanged	(depends)
Operator separation	changed	unchanged	static
Fusion	changed	unchanged	(depends)
Fission	changed	(depends)	(depends)
Placement	unchanged	unchanged	(depends)
Load balancing	unchanged	unchanged	(depends)
State sharing	unchanged	unchanged	static
Batching	unchanged	unchanged	(depends)
Algorithm selection	unchanged	(depends)	(depends)
Load shedding	unchanged	changed	dynamic

Discussion on Optimization

- Metrics for evaluation
 - many ways to measure whether a streaming optimization was profitable
 - throughput, latency, quality of service (QoS), accuracy, power, and system utilization
- Need standard benchmarks for streaming workloads
 - Stanford stream query repository including Linear Road [Arasu et al. 2006],
 - BiCEP benchmarks [Mendes et al. 2009]
 - StreamIt benchmarks [Thies and Amarasinghe 2010]
 - *Project idea?* 😊

Arasu, A., Babu, S., and Widom, J. 2006. The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15, 2 (June), 121–142.

Mendes, M. R. N., Bizarro, P., and Marques, P. 2009. A performance study of event processing systems. In TPC Technology Conference on Performance Evaluation & Benchmarking (TPC TC). 221–236.

Thies, W. and Amarasinghe, S. 2010. An empirical characterization of stream programs and its implications for language and compiler design. In Parallel Architectures and Compilation Techniques (PACT). 365–376.

Wrap-up on Optimization

- Non-trivial to optimize performance
 - Compiler and programming language support important
 - Often requires manual tuning and experimentation
 - Lots of dynamics (compute resources, operator performance, data characteristics and workload)
 - Several different competing objectives
- Guiding principle
 - Keep operators small and lightweight and explore many knobs
- Impact on fault tolerance
 - Many optimizations are orthogonal to whether or not the system is fault tolerant.
- Centralized versus distributed optimization
 - Assumptions on shared memory, or other resources may not be valid

Stream Mining and Analytics

- Streaming analytics
 - Techniques and algorithms from fields such as data mining, machine learning, statistics, signal processing, and artificial intelligence that are adapted to work in a streaming setting
- Important features of the streaming setting
 - Ideally Single scan: Data is seen only once, in order of arrival
 - Unlike stored data, that can be scanned multiple times, potentially indexed and accessed in other ways
 - Limited memory: The amount of memory that can be used is small compared to the size of the data stream
 - Linear space algorithms may be too expensive
 - Logarithmic or constant is good
 - Distributed approaches for mining and analysis can be exploited

Outline of Data Mining Process

- Data acquisition
 - Collect data from external sources
- Data pre-processing
 - Prepare data for further analysis: data cleaning, data interpolation (missing data), data normalization (heterogeneous sources), temporal alignment and data formatting, data reduction
- Data transformation
 - Select an appropriate representation for the data
 - Select or compute the relevant features (*feature extraction/selection*)
- Modeling (data mining)
 - Identify interesting patterns, similarity and groupings, partition into classes, fit functions, find dependencies and correlations, identifying abnormal data
- Evaluation
 - Use of the mining model and evaluation of the results

Data Preprocessing and Transformation

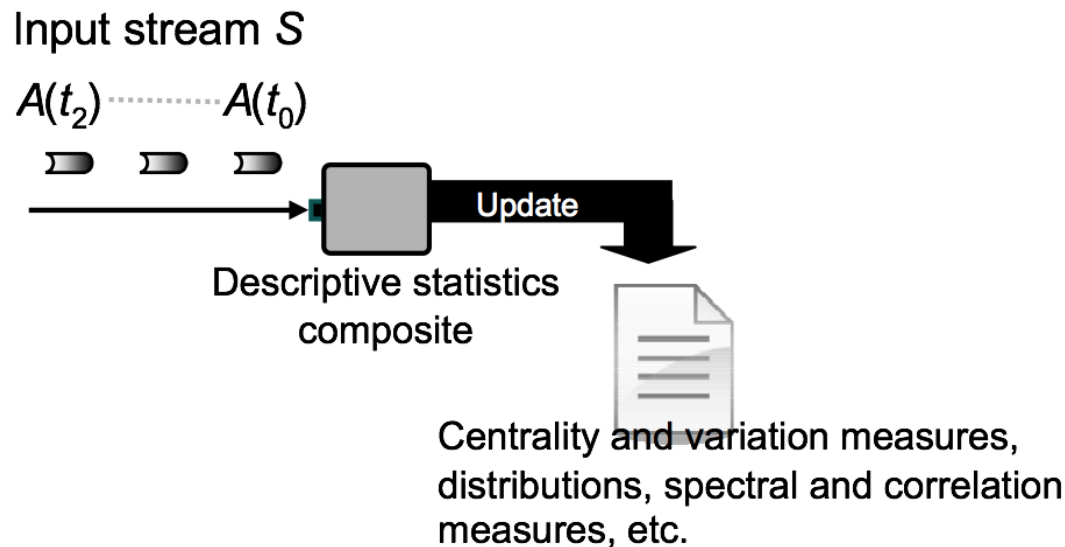
- Descriptive statistics
 - Extracting simple quantitative statistics of the distribution
- Sampling
 - Reducing the volume of the input data by retaining only an appropriate subset for analysis
- Sketches
 - Compact data structures that contain synopses of the streaming data, for approximate query answering
- Quantization
 - Reduce the fidelity of individual data samples to lower compute and memory costs
- Dimensionality reduction
 - Reduce the number of attributes within each tuple to decrease data volume and improve accuracy of the models
- Transforms
 - Convert data items or tuples and their attributes from one domain to another, such that data is better suited for further analysis

Quick Aside: Notation

- Stream: Sequence of tuples
 - $S = A(t_0), A(t_1), A(t_2), \dots$
- Tuple: Set of N attributes
 - $A(t) = \{a_0(t), a_1(t), \dots, a_{N-1}(t)\}$
- Special class of tuples: Numeric tuples
 - Common in many applications
 - Methods to convert categorical values to numeric values (e.g. bag of words)
 - Treated as vectors

$$A(t) = \mathbf{x}(t) = \begin{bmatrix} x_0(t) \\ \vdots \\ x_{N-1}(t) \end{bmatrix}$$

Descriptive Statistics



- Extract information from the tuples in a stream such that
 - statistical properties of the stream can be characterized or summarized
 - the quality of the data it transports assessed

Descriptive Statistics: Measures of Centrality

- Sample moments

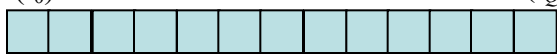
- i -th moment m_i

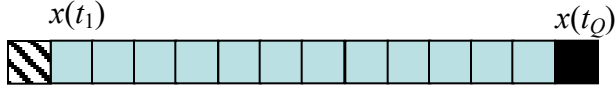
$$m_i = \frac{1}{Q} \sum_{j=0}^{Q-1} (x(t_j))^i$$

- First moment: Sample mean
- Second moment sample variance $\sigma^2 = m_2 - m_1^2$
- Third moment: sample skew
- Streaming friendly
 - All these can be computed using a single scan
 - They can also be computed over a sliding window

Maintaining Moments across Sliding Windows

- Moments can be computed incrementally without storing original value
 - Maintain sum (raised to the appropriate power) and number of values
- What happens with tumbling windows?
 - Reset sum and counter to 0 every tumbling window boundary
- What happens with sliding windows?
 - Consider the example of the sample mean, and a slide of 1

Window W_1  Sample mean: $m_1(W_1) = \frac{1}{Q} \sum_{j=0}^{Q-1} x(t_j)$

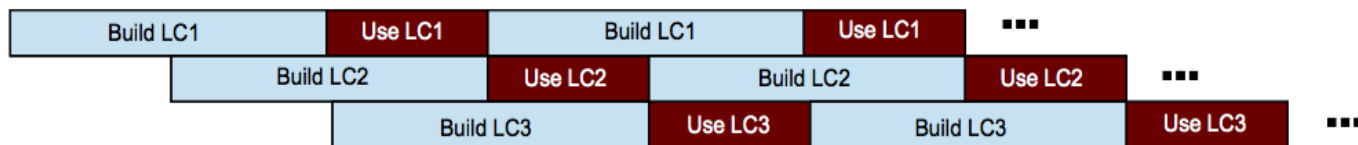
Window W_2  $m_1(W_2) = \frac{1}{Q} \sum_{j=1}^Q x(t_j)$ or $m_1(W_2) = m_1(W_1) - \frac{x(t_0)}{Q} + \frac{x(t_Q)}{Q}$

Naïve Incremental

Incremental requires $O(2)$ compute, as opposed to $O(Q)$ compute, and both approaches require storing $O(Q)$ values. Note that if slide = k , we need to typically store $O(Q/k)$ values.

Maintaining Statistics Across Sliding Windows

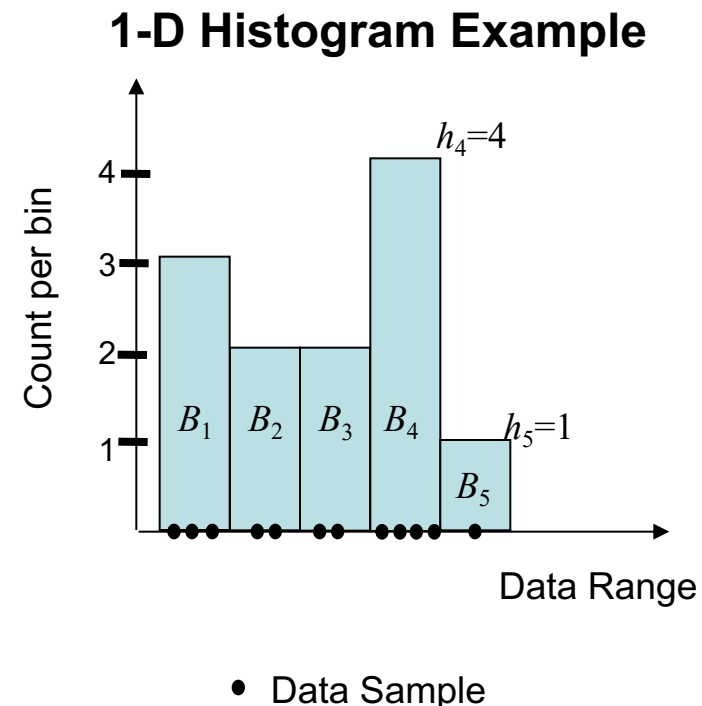
- Say you have some statistic for which you have an efficient algorithm (logarithmic space, one-scan, etc.) to compute it
- In this course, we will see several such algorithms
 - Often these algorithms assume a single scan, where data items are being added
 - However, you may need to compute sliding window versions of these to capture changing statistics
 - You will find algorithms in the literature that are specialized for this, but often the sliding window versions are complex
- A quick and dirty alternative is to use overlapping tumbling windows



- If we use n tumbling windows each of size Q
 - At any time we are using a summary computed from
 - last Q to $Q + Q/(n-1)$ items

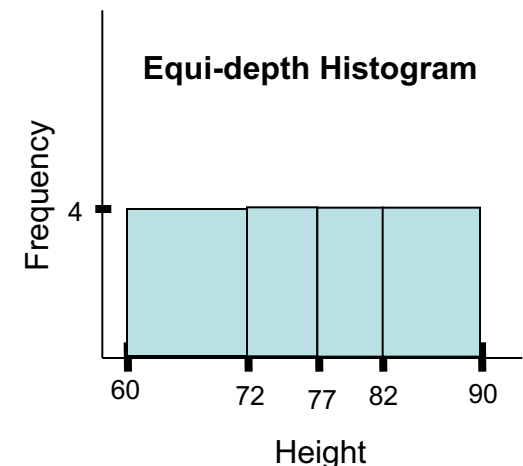
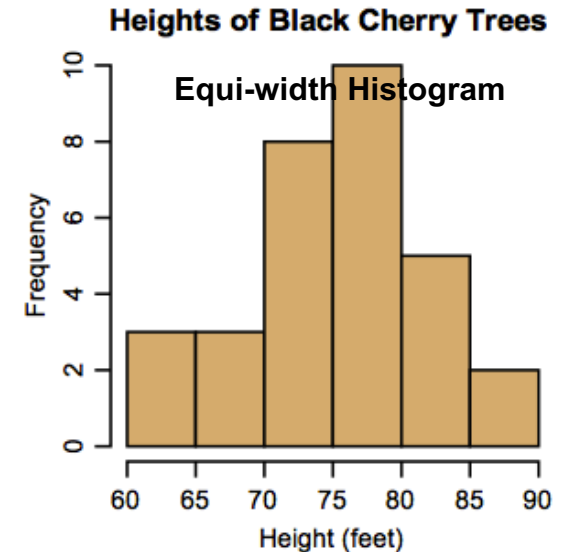
Counts and Histogram

- Given N data items
 - data items $\mathbf{x}(t_0) \dots \mathbf{x}(t_{N-1})$
- The domain of a variable is divided into q non-overlapping bins
 - B_1 through B_q
- The number of items in each bin is counted
 - h_1, \dots, h_q
 - $h_i = |\{\mathbf{x}(t_j) \in B_i ; 0 \leq j < N\}|$
- Histogram H is a collection of these counts
 - May be viewed as a vector



Counts and Histograms: 1-D

- An *equi-width histogram* has equally sized bins
 - Easy to compute using a single scan
 - Compute bucket and increment
- An *equi-depth histogram* has \sim equal count bins
 - Variable-width bins to get similar counts
 - Can be used to compute *quantiles*
 - k th q -quantile is the largest value v such that the probability of the variable being less than v is at most k/q .
 - Not easy to compute in a streaming fashion
 - Approximate techniques exist in the literature



Streaming Descriptive Statistics Algorithms

- Non sliding window methods
 - Auto Regressive Integrated Moving Average (ARIMA)
 - Kalman Filters
 - Holts-Winters Models (additive and multiplicative)
 - Kernel smoothing methods
- Sliding window methods
 - Variance, median and quantile estimation
 - Histogram Estimation

BasicCounting Algorithm

- Question: Assume you have a stream that contains tuples that are either 0s or 1s, how can you maintain the number of 1s in the last W tuples
 - Sliding window of size W , slide 1
- Assume W is large
 - So $O(W)$ is too large. We want to store logarithmic space
- The *BasicCounting* algorithm
 - Uses space of the order $O\left(\frac{1}{\varepsilon} \log^2 W\right)$
 - With error tolerance ε , i.e. the counts are within $1 \pm \varepsilon$ of its actual value. ε is specified by application objective
- Study: Data structure, update, query

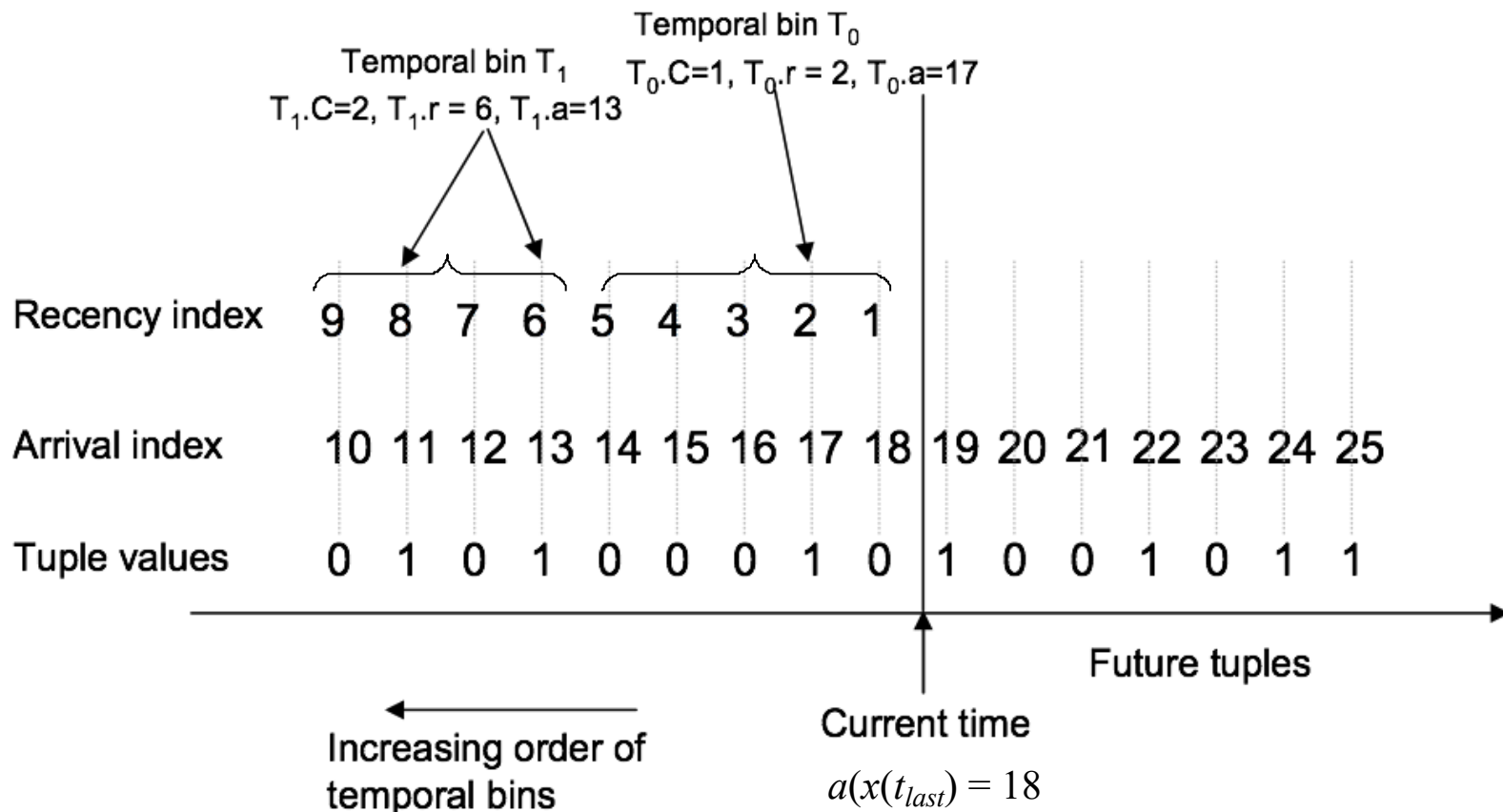
Data Structure for BasicCounting

- Assign each scalar tuple $x(t)$ an arrival index a
 - Sequence number incremented by one per tuple
- For each tuple $x(t)$ define recency index r
 - $r(x(t)) = a(x(t_{last})) - a(x(t)) + 1$
 - Distance of arrival index of tuple from last received tuple, i.e. how old or recent the tuple is
 - Changes every time a new tuple arrives!
- Maintain m temporal bins T_0, \dots, T_{m-1}
 - Each bin is a stream sub-sequence summary

Data Structure for BasicCounting

- For each bin T_i store
 - $T_i.r$: Smallest recency index of tuple that had value 1
 - $T_i.a$: Arrival index of the most recent tuple that had value 1
 - $T_i.C$: The number of tuples with value 1 in the bin
- Keep arrival index of the last received tuple as well
- Note: We do not need to explicitly store recency index for the bin
 - Can be computed from bin arrival index and arrival index of last tuple

Data Structure for BasicCounting



Structure Maintenance Algorithm

- Set $k = \left\lceil \frac{1}{\varepsilon} \right\rceil$
- When a tuple arrives, update arrival index of last tuple
 - Assume most recent bin is T_0 and oldest bin is T_{m-1}
 - If $T_{m-1}.r > W$, discard this bin
 - Does not contribute to the total count of 1s in the window
- If this tuple $x(t_{last})=1$
 - Create a new bin
 - If there are more than $k+1$ bins with count (C) 1, merge the oldest 2 bins with $C=1$ into a new bin with $C=2$
 - If there are more than $k/2+1$ bins with $C=2$, merge the oldest 2 bins with $C=2$ into a new bin with $C=4$
 - If there are more than $k/2+1$ bins with $C=y$, merge the oldest 2 bins of with $C=y$ into a new bin with $C=2*y$
- If tuple $x(t_{last})=0$, do nothing
- Creates bins with counts powers of 2 (what happens to the size?). Also ensures $T_j.C \geq T_{j-1}.C$ for all j

Structure Maintenance Example

i	$x(t_i)$	m	$T_0.C$	$T_1.C$	$T_2.C$	$T_3.C$	$T_4.C$	$T_5.C$	Operation
1	1	1	1	-	-	-	-	-	Insert
2	1	2	1	1	-	-	-	-	Insert
3	1	3	1	1	1	-	-	-	Insert
4	1	4	1	1	1	1	-	-	Insert
4	1	3	1	1	2	-	-	-	Merge
5	1	4	1	1	1	2	-	-	Insert
6	1	5	1	1	1	1	2	-	Insert
6	1	4	1	1	2	2	-	-	Merge
7	1	5	1	1	1	2	2	-	Insert
8	1	6	1	1	1	1	2	2	Insert
8	1	5	1	1	2	2	2	-	Merge
8	1	4	1	1	2	4	-	-	Merge

$$k = 2 \ (\epsilon = 0.5)$$

Each bin has at least one 1

Query Result and Error Analysis

- When asked for number of 1s in last W tuples, return

$$\sum_{j=0}^{m-2} T_j \cdot C + \frac{T_{m-1} \cdot C}{2}$$

- Error Analysis

- No bin summarizes only tuples older than W
 - We first delete such bins
- The only bin that summarizes tuples, some of which are older than W is T_{m-1} . Error in the count is due to these older tuples.
- Expected error in count is therefore $T_{m-1} \cdot C/2$

- Largest number of 1s possible is $\sum_{j=0}^{m-1} T_j \cdot C$
- Smallest number of 1s possible is $\sum_{j=0}^{m-2} T_j \cdot C + 1$

Error Analysis

- Maximum expected relative error occurs with the smallest number of possible 1s

- Hence relative $err \leq \frac{T_{m-1}.C}{2\left(1 + \sum_{j=0}^{m-2} T_j.C\right)}$

- Recall how we split bins (based on k).
 - We have bin sizes that are increasing, and powers of 2
 - There are at least k and at most $k+1$ bins of size 1
 - There are at least $k/2$ and at most $(k/2+1)$ bins of all other sizes

$$err \leq \frac{T_{m-1}.C}{2\left(1 + \sum_{j=0}^{m-2} T_j.C\right)} \leq \frac{1}{k}$$

Error Analysis

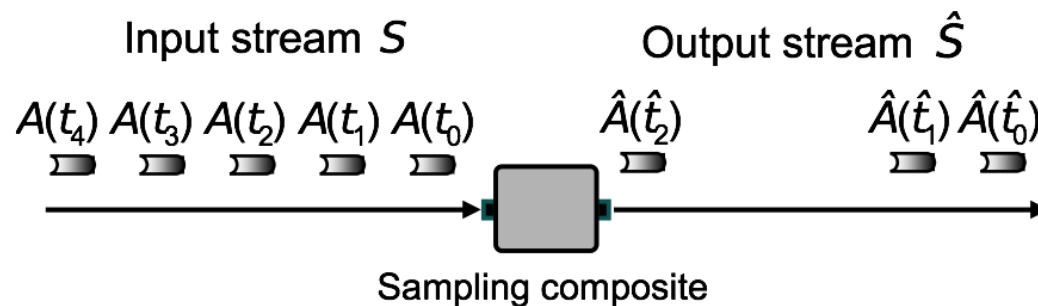
- Finally, since $k = \left\lceil \frac{1}{\varepsilon} \right\rceil$
- We have $err \leq \frac{T_{m-1}.C}{2 \left(1 + \sum_{j=0}^{m-2} T_j.C \right)} \leq \frac{1}{k} \leq \varepsilon$
- Relative error is bounded by ε
- Finally, we can show we only have $O\left(\frac{1}{\varepsilon} \log^2 W\right)$ bins
- BasicCounting algorithm extensions
 - From binary to produce counts of non-zero values if they lie in range $[0, R]$
 - Moving towards histogram estimation
- *Can you implement this algorithm?*

Basic Counting: Summary

- Need to tradeoff space and compute cost versus result accuracy in streaming scenario
- Design of a streaming algorithm non-trivial
 - Especially with bounds on error/space tradeoffs

Sampling

- Sampling is used to reduce the data stream by selecting some tuples based on one or more criteria



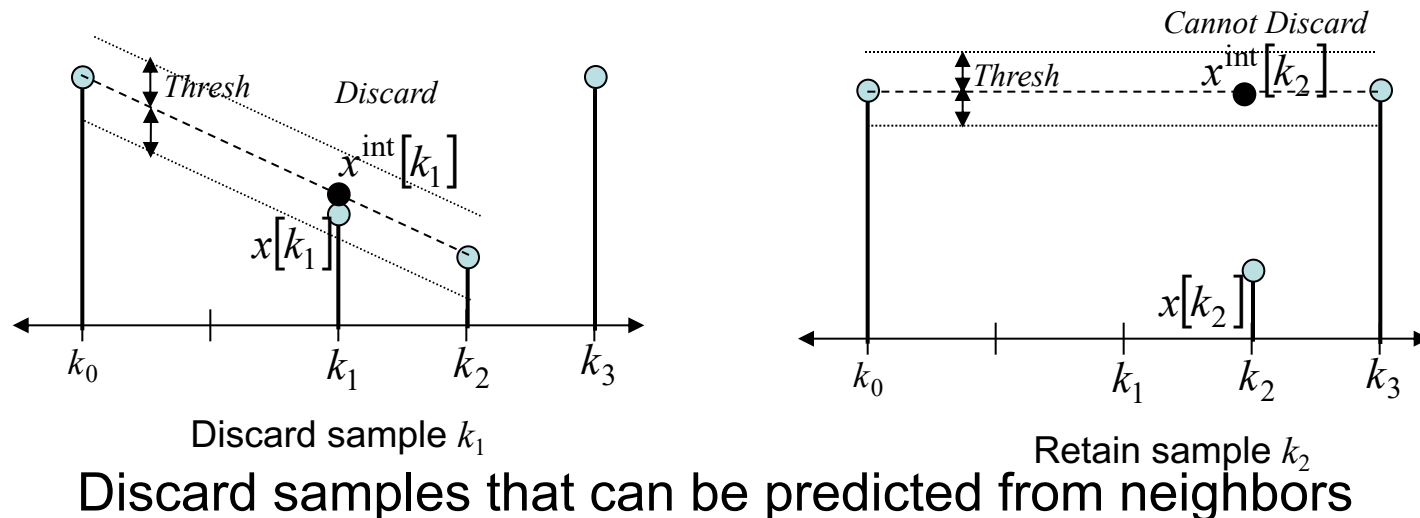
- Three types of sampling
 - Systematic or Uniform
 - Data Driven
 - Random

Systematic or Uniform Sampling

- Uniform Sampling
 - Sample every k -th tuple
 - Can be lossless
 - Recall Nyquist Sampling theorem and Aliasing
 - Simple streaming implementation
- Uniform sampling with random offset
 - Randomly pick a seed $[0, k-1]$
 - Starting from that sample every k -th sample
- Probability of sampling a tuple is $1/k$
- Number of retained samples grows with stream length
 - If we want to store samples – does this mean we have infinite samples when the data stream is infinite in length?

Data Driven Sampling

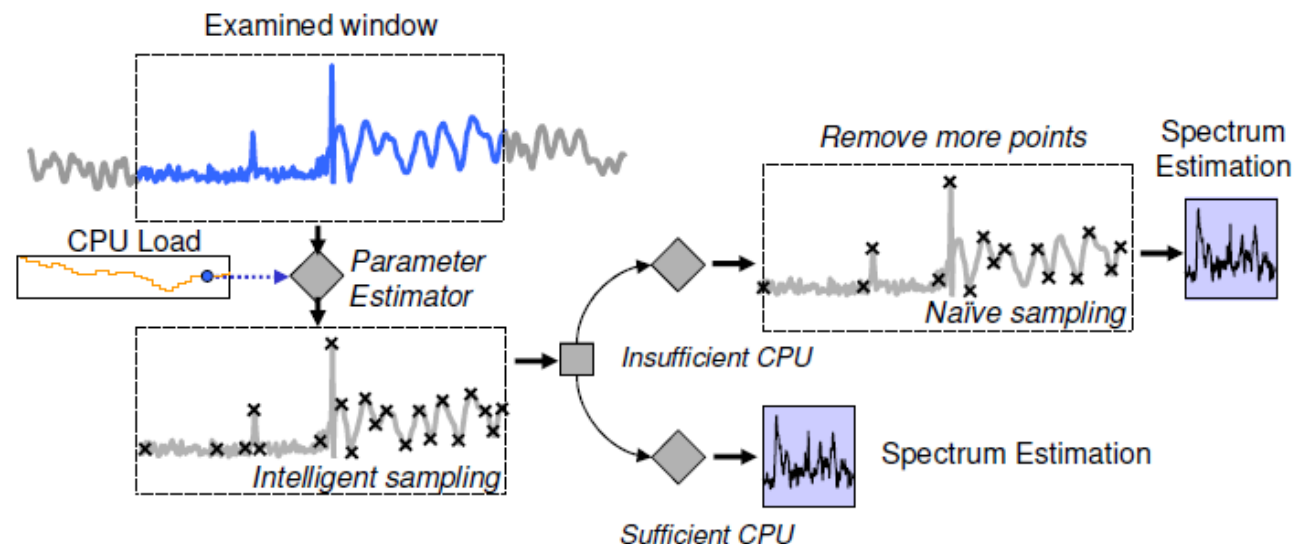
- Use data values to determine whether to sample or not
 - Does not result in uniformly spaced tuples even if original stream was uniformly spaced in time
 - Example: Interpolation driven sampling (SLIDE)
 - Based on prior values, or even future values



Data Driven Sampling

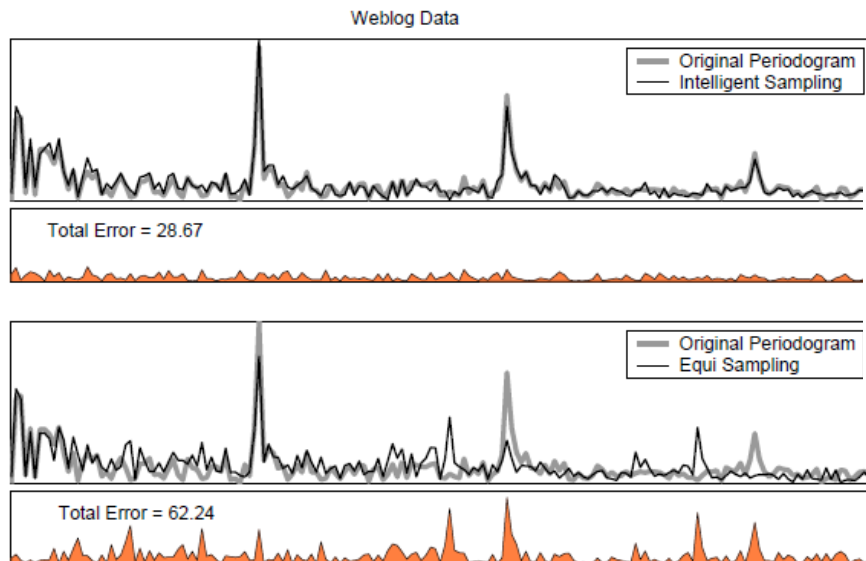
- Can result in much higher compression than uniform sampling (in terms of error recovery)
- What is the probability of sampling a tuple?
 - Dependent on tuple value. Unclear apriori
 - Depends on parameters, e.g. threshold. How do we set threshold?
 - Learn from previous window to set threshold for next window (can be done incrementally, requires storing of some features)

Adaptive Selection of Threshold Parameter



Data Driven Sampling

- Adds delay with look ahead
- Not trivial to estimate spectral properties of unevenly sampled signal
 - SLIDE allows some closed form estimates



Dataset	Threshold	Window Compression (%)	Error Equi-Sampling	Error Intelligent
ECG	20	80.96	1627.55	450.79
	60	91.40	2434.59	1326.23
	100	95.79	2934.84	2171.04
EEG	20	6.73	79.79	2.76
	60	18.45	202.03	33.10
	100	32.81	221.16	105.99
RTT	20	35.90	147.76	26.68
	60	60.69	174.24	81.21
	100	75.55	210.69	123.98
WebTrace	20	13.97	22.08	4.04
	60	37.26	46.29	18.98
	100	61.36	52.31	47.52

Random Sampling

- Select tuples randomly using some probability distribution
 - Used in statistics to get unbiased estimates
- Disadvantage of uniform and data-driven sampling (e.g. SLIDE)
 - Even if you sample, you still need infinite space, since the stream is endless
- Reservoir sampling
 - How do you maintain a reservoir of finite number of samples, such that every tuple has the same probability of being included
 - Independent of length of stream

Reservoir Sampling Algorithm

- Let q be the reservoir size
 - Indices in reservoir run from 0 to $q-1$
- Let i be the index of the tuple being processed right now
- If $i < q$
 - Append the tuple to the reservoir
- Otherwise
 - Select p as a random integer in range $[0, i]$
 - If $0 \leq p < q$
 - Replace tuple at index p in reservoir with current tuple

Analysis of Reservoir Sampling

- Consider that we have seen i tuples so far
- Consider a reservoir with size q
- Consider tuple k
 - We determine probability of this tuple being in the reservoir
- Case 1: $k < q$
 - This tuple was added to reservoir when it arrived
 - For this tuple to be in reservoir, no other tuple after q should have replaced it

$$p(k) = \prod_{j=q}^i \left(1 - \frac{1}{j+1} \right) = \frac{q}{i+1}$$

Analysis of Reservoir Sampling

- Case 2: $k \geq q$
 - This tuple can be added to the reservoir with probability

$$\frac{1}{k+1} / \frac{1}{q} = \frac{q}{k+1}$$

- If this was added, then probability it is not removed by subsequent tuples is

$$\prod_{j=k+1}^i \left(1 - \frac{1}{j+1}\right) = \frac{k+1}{i+1}$$

- Hence probability it stays in reservoir is

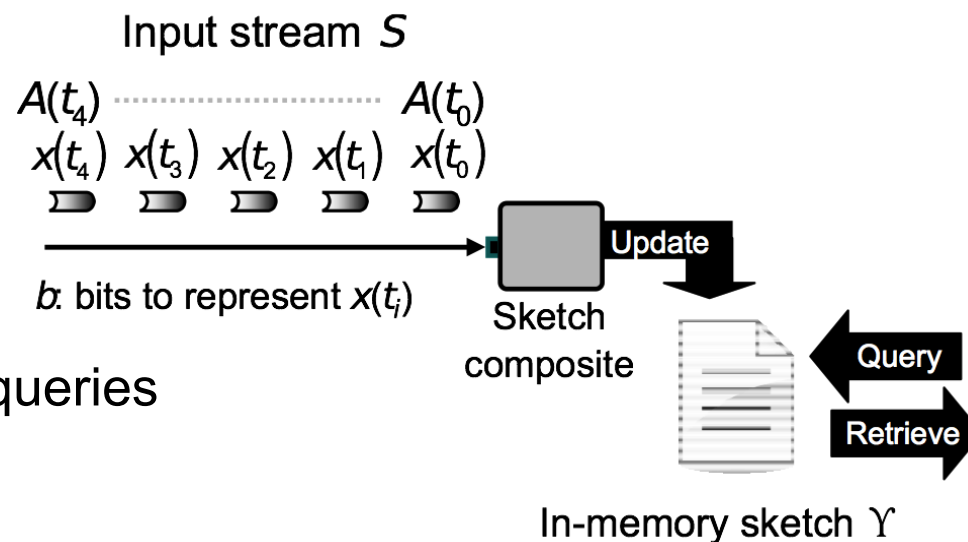
$$p(k) = \frac{q}{k+1} \frac{k+1}{i+1} = \frac{q}{i+1}$$

Analysis of Reservoir Sampling

- Probability of retaining any tuple
 - Only depends on reservoir size, and how many total tuples seen so far
 - Identical for all tuples: uniform sampling probability
- Originally proposed for tape deck compression
 - Pretty useful in streaming context
 - Recall: used during profiling
 - Easy to implement
- Reservoir sampling extended to sliding window case

Sketches

- In-memory data structures that contain compact, often lossy, synopses of streaming data
- They capture the key properties of the stream



- In order to answer specific queries
 - Error bounds and
 - probabilistic guarantees
- Used for approximate query processing

Popular Sketches

- Sketches to answer *frequency based queries*
 - Queries about the frequency distribution of a dataset
 - frequency of specific values
 - heavy hitters (frequently appearing values)
 - quantiles and equi-depth histograms
 - Examples: Count-Min sketch, AMS sketch
- Sketches to answer *distinct value based queries*
 - Queries about the cardinality of the stream
 - number of distinct values
 - Useful for cardinality estimation of relational queries
 - Examples: Flajolet-Martin, Gibbons-Tirthapura, BJKST

Aside on Hash Functions

```
function Hash(key)
    return key mod PrimeNumber
end
```

Additive Hash

```
ub4 additive(char *key, ub4 len, ub4 prime)
{
    ub4 hash, i;
    for (hash=len, i=0; i<len; ++i)
        hash += key[i];
    return (hash % prime);
}
```

Rotating Hash

```
ub4 rotating(char *key, ub4 len, ub4 prime)
{
    ub4 hash, i;
    for (hash=len, i=0; i<len; ++i)
        hash = (hash<<4)^(hash>>28)^key[i];
    return (hash % prime);
}
```

Bernstein's hash

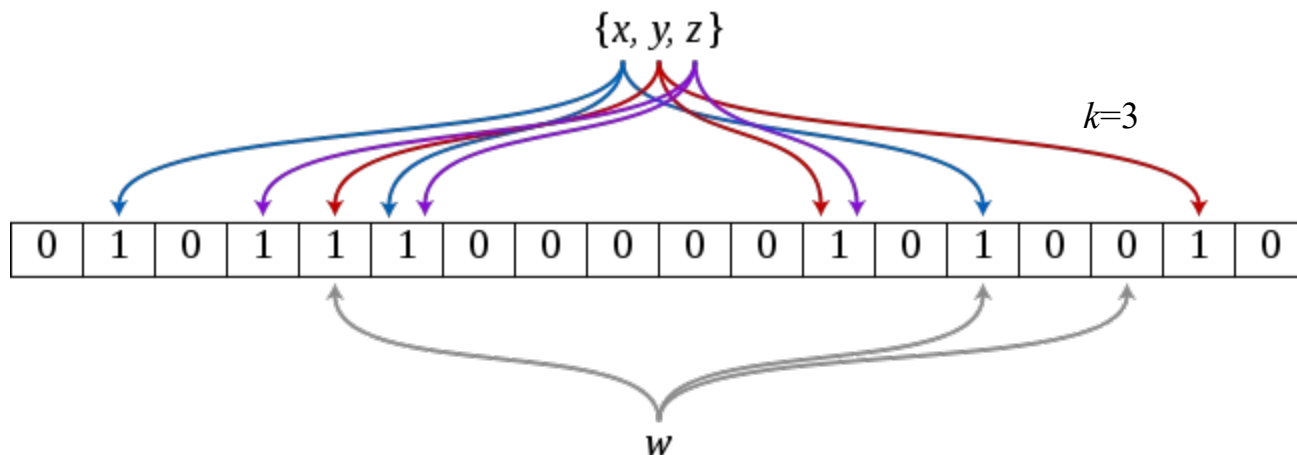
```
ub4 bernstein(ub1 *key, ub4 len, ub4 level)
{
    ub4 hash = level;
    ub4 i;
    for (i=0; i<len; ++i) hash = 33*hash + key[i];
    return hash;
}
```

If your keys are lowercase English words, this will fit 6 characters into a 32-bit hash with no collisions

Several cryptographic hash functions, e.g. SHA2

Bloom Filters

- Let us say we want to answer containment queries
- Whether a given item has been seen so far or not
- Bloom filter
 - Keep a bit array of size m
 - Use k pairwise independent hash functions (more later)
 - Update: Hash item to k locations and set the bits to one
 - Query: Hash item to k locations
 - Return true if all of the k locations are set



Bloom Filters

- False negatives are not possible
 - If we say an item has not been seen, then it is not seen
- False positives are possible
 - If we say an item has been seen, we may be wrong
 - We want to set m, k to reach a desired possibility
- How to create pairwise independent hash functions?
 - In general difficult (we will discuss an example later)
 - Take a hash function with large range and slice into k parts
 - If a hash function takes a seed, give it k different seeds

Bloom Filter Analysis

- Probability of a bit not being set to 1 by a hash function (assuming hash function is uniform)

$$1 - \frac{1}{m}$$

- Probability of a bit not being set by k independent hash functions

$$\left(1 - \frac{1}{m}\right)^k$$

- Probability of a bit not being set after observing n values

$$\left(1 - \frac{1}{m}\right)^{kn}$$

- Hence probability of a bit being 1 after n values

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Bloom Filter Analysis

- Now, when we test for new value probability that all k bits are set to 1 is

$$\left[1 - \left(1 - \frac{1}{m}\right)^{kn}\right]^k$$

- This false positive probability may be approximated as

$$\left(1 - e^{-kn/m}\right)^k$$

- Optimal number of hash functions for a given m and n is

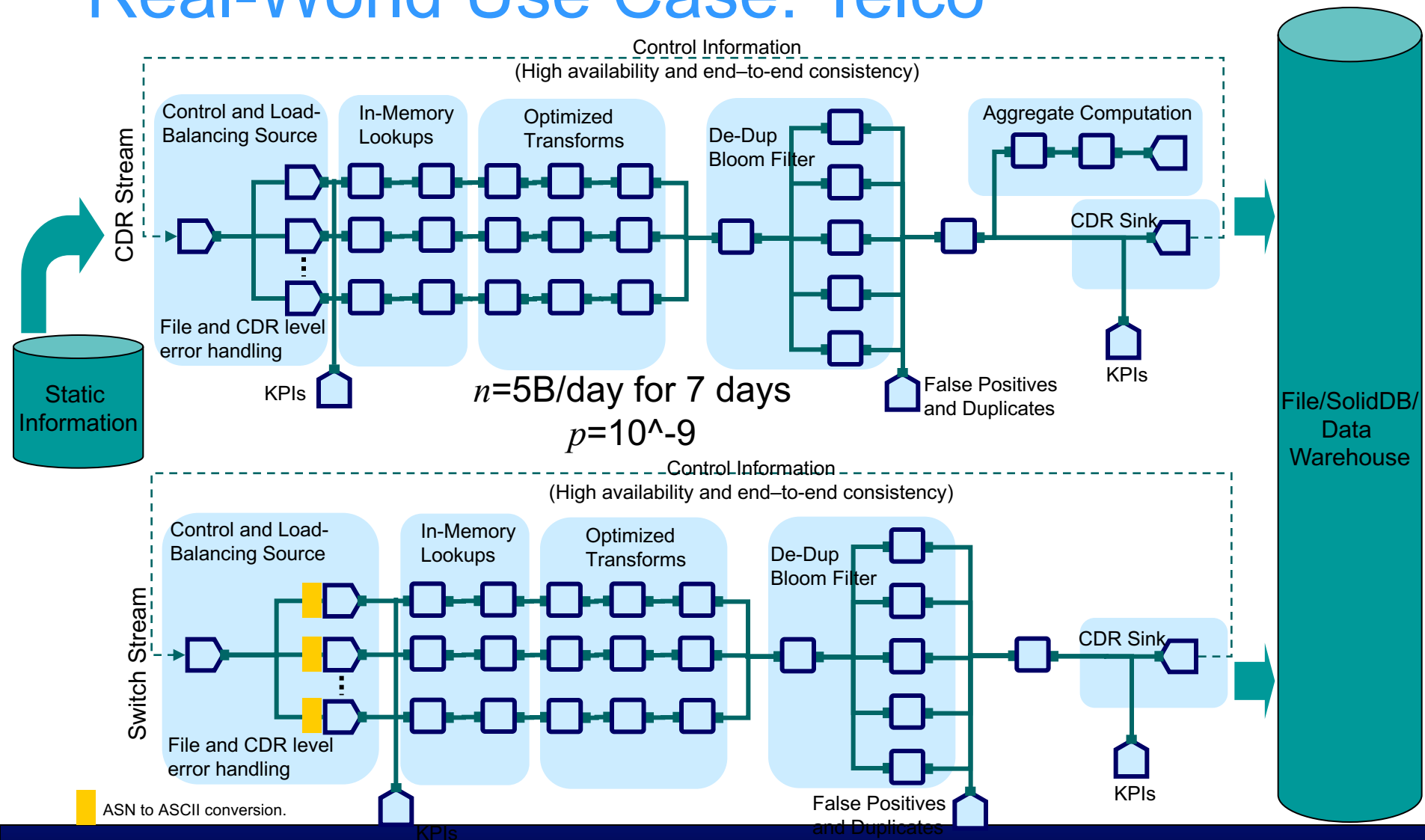
$$k = \frac{m}{n} \log 2$$

- Hence, given a n a desired false alarm rate p , we have

$$m = \frac{-n \ln p}{(\ln 2)^2}$$

Bloom filter calculator: <http://hur.st/bloomfilter?n=4&p=1.0E-20>

Real-World Use Case: Telco



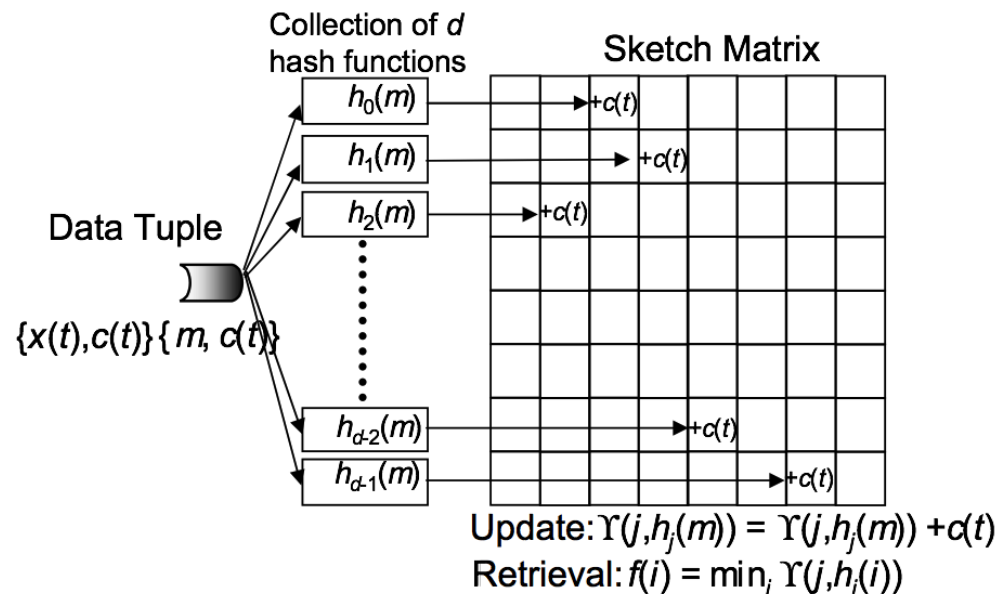
Count-Min Sketch

- Assume we have a numeric attribute with a very large domain
 - E.g. the domain of ip v6 addresses
- Assume we want to find, given a value, its frequency
- Let us use M to denote the domain size
- The Count-Min sketch keeps d rows and w columns
- Let H be a set of d pair-wise independent hash functions with range $[0..w)$

$$Pr_{h \in H} [h(x_1) = y_1 \wedge h_k(x_2) = y_2] = \frac{1}{w^2}$$

Count-Min Sketch: Construction and Retrieval

- Construction: Apply the d hash functions
 - Increment the cell in each row whose index is the hash value
- Retrieval: Apply the d hash functions
 - Pick the smallest count value



Count-Min Sketch Properties

- Set $w = \lceil 2/\epsilon \rceil$ and $d = \lceil \log(1/\delta) \rceil$
- With probability $1 - \delta$
 - The error in the estimate ($|\text{original-estimate}|$) is less than ϵ times the sum of all frequencies
- Say δ is 0.01 and ϵ is 10^{-6}
- So with probability 0.99, our estimate is going to be within $\pm 10^{-6}$ of the original count
- This will require 20MB state
- 5 hash functions (very quick update and query)

Generating Independent Hash Functions

- Pick a prime p in the range $w < p < 2w$
 - such a prime exists due to Bertrand-Chebyshev theorem
- For each j in $[0..d)$
 - Pick two random numbers $a_j > 0$ and b_j in range $[0..p)$
 - Set

$$h_j : x \rightarrow ((a_j \cdot x + b_j) \bmod p) \bmod w$$

Quantization

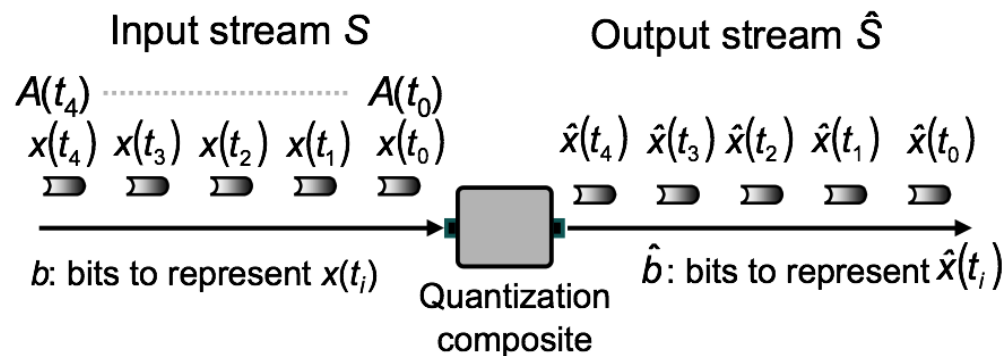
- Simplest lossy data reduction mechanism
 - Perfect reconstruction of original signal not possible
- Maps Input Sample \rightarrow Reconstruction Value from predefined codebook

- Example

- Rounding Function
- Floor Function
- Truncation

- Original sample

- Numeric Tuples with Continuous or Discrete Space



Quantization

- We go from b bits to b' bits, where $b' < b$
- The *quantization error*: A function on the difference between the original and the quantized value
- Advantages:
 - Reduces memory requirement
 - Reduces computation requirements
- Two main techniques
 - Scalar quantization
 - apply on a single attribute
 - Vector quantization
 - apply on a vector of attributes

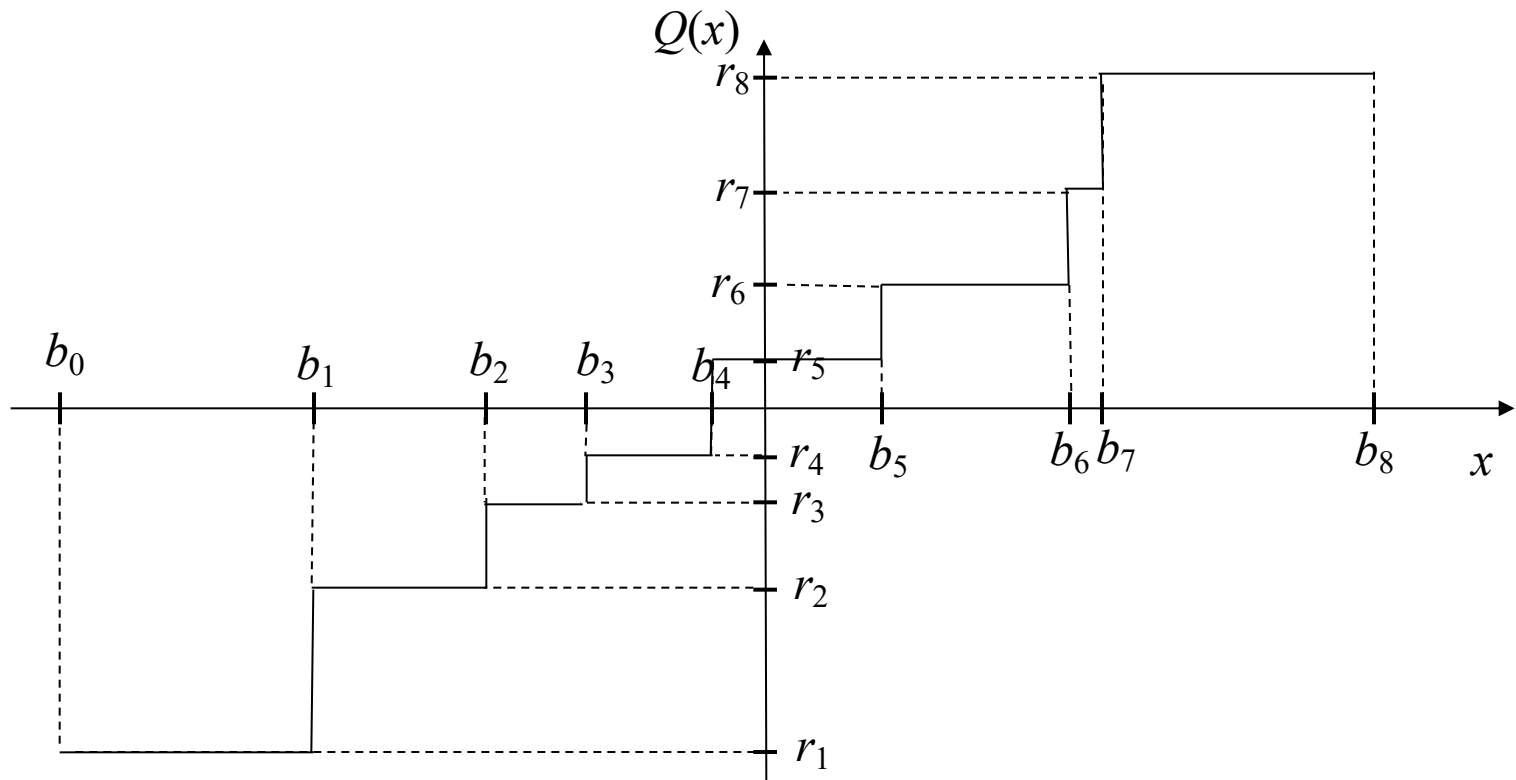
Scalar Quantization

- Quantizer Q described by
 - L : the number of reconstruction levels
 - Set $L = \{1, 2, \dots, L\}$ of reconstruction level indices
 - Requires at most $(\log_2 L)$ bits for representation
 - b_l : the set of boundary values
 - Boundary region $B_l = [b_{l-1}, b_l)$
 - Support space $B = \{B_l\}$
 - r_l : the set of reconstruction levels

$$Q(x) = r_l \text{ if } x \in B_l, l \in L$$

Partitions the support space of random variable x into L regions
Represents x with L discrete values

Scalar Quantization



Graphical Representation

Scalar Quantization

- Select boundaries and reconstruction levels to minimize reconstruction error
 - Mean absolute error
 - Mean squared error
- Multiple Greedy Optimization Techniques
 - Lloyd-Max algorithm (not a streaming algorithm)

Scalar Quantization

$$D_Q = E[d(X, Q(X))] = \int_{x \in} d(x, Q(x)) p_X(x) dx$$

Distance function Probability density

$$d(x, y) = |x - y|$$

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Gaussian

$$d(x, y) = |x - y|^2$$

Mean squared error (MSE)

$$p_X(x) = \begin{cases} \frac{1}{b-a} & a \leq x < b \\ 0 & \text{otherwise} \end{cases}$$

Uniform

$$p_X(x) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}$$

Laplacian

Scalar Quantization

$$D_Q = \int_{x \in} d(x, Q(x)) p_X(x) dx$$

$$D_Q = \sum_{l=1}^L \int_{b_{l-1}}^{b_l} d(x, r_l) p_X(x) dx$$

$$D_Q = \sum_{l=1}^L \int_{b_{l-1}}^{b_l} (x - r_l)^2 p_X(x) dx$$

MSE distance function

To derive the optimal or Minimum Mean Squared Error Quantizer, take derivatives w.r.t. r_l and b_l and set to zero

Scalar Quantization

$$\frac{\partial D_Q}{\partial r_l} = 0 \Rightarrow \int_{b_{l-1}}^{b_l} 2(x - r_l) p_X(x) dx = 0$$

$$r_l = \frac{\int_{b_{l-1}}^{b_l} x p_X(x) dx}{\int_{b_{l-1}}^{b_l} p_X(x) dx} = E[X | X \in B_l]$$

Optimal reconstruction values lie at the centroid of each boundary region

$$\frac{\partial D_Q}{\partial b_l} = 0 \Rightarrow (b_l - r_l)^2 p_X(b_l) - (b_l - r_{l+1})^2 p_X(b_l) = 0$$

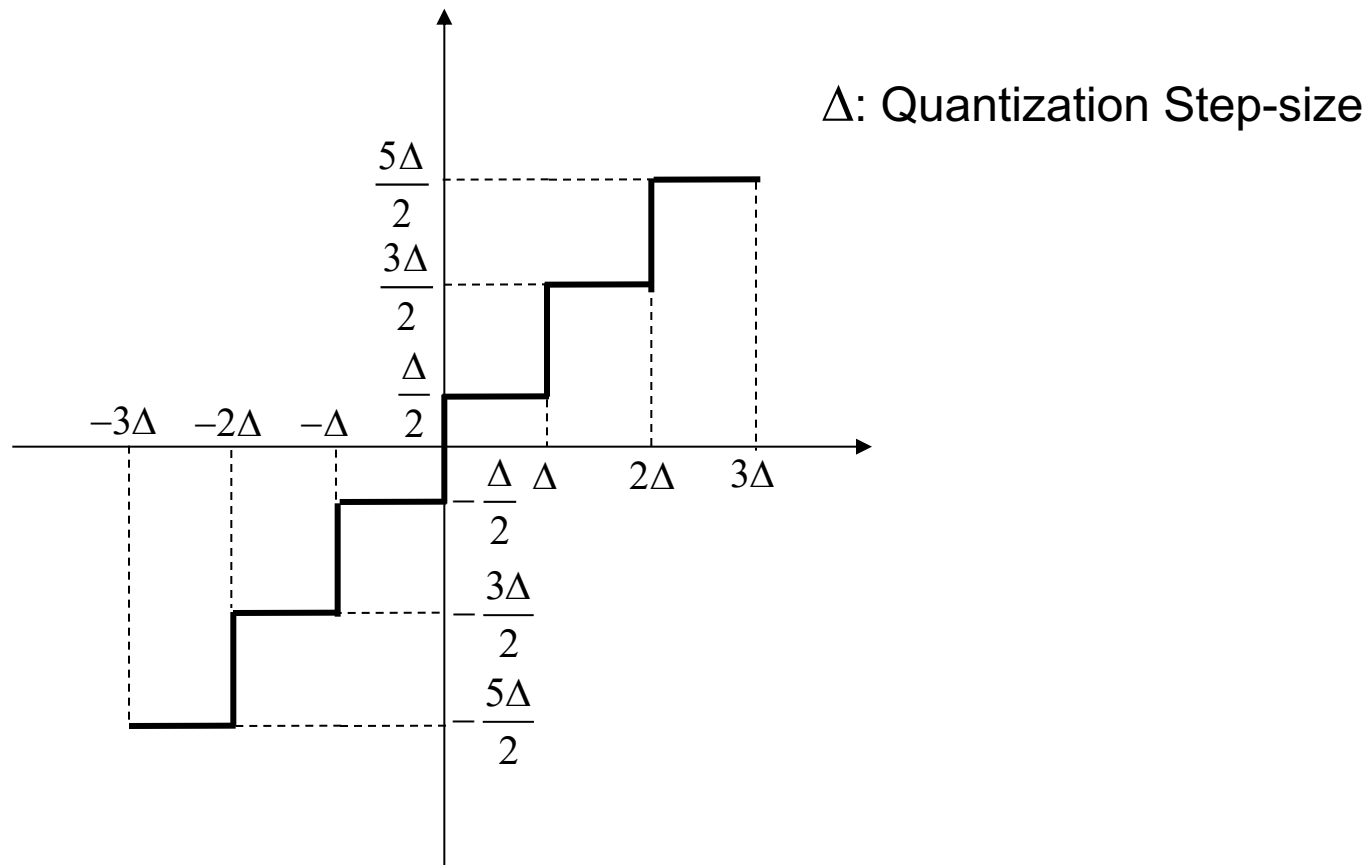
$$b_l = \frac{r_{l+1} + r_l}{2}$$

Lloyd-Max Algorithm

- Iterative Algorithm to determine r_l and b_l
 - Based on a set of training samples
- Step 1: Pick a set of reconstruction levels
- Step 2: Determine boundary partitions
 - Average of reconstruction levels (use nearest neighbor condition if discrete set of values)
- Step 3: Update reconstruction levels as centroids of each boundary region
- Step 4: Compute Quantization Distortion
- Iterate Steps 2 through 4 till Distortion converges

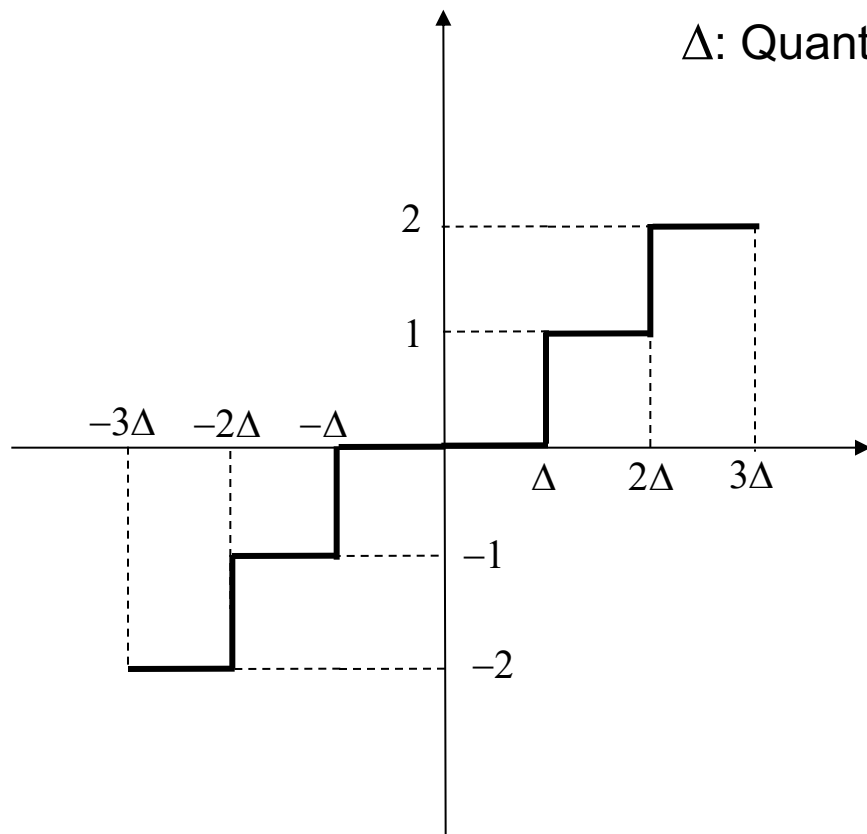
- Gradient Descent Based Algorithm
 - Convergence guaranteed to only Local Minimum (not necessarily Global Minimum)
- *Is this Streaming?*

Uniform Quantizer



Optimal MMSE Quantizer
for uniform distribution

Uniform Quantizer: Mid-tread



Uniform Mid-Tread Quantizer
Quantization with "Dead Zone"

"Dead Zone"

$$Q(x) = \text{sign}(x) \times \frac{|x + f(\Delta)|}{\Delta}$$

$$\hat{x} = \Delta Q(x)$$

Reconstruction of original

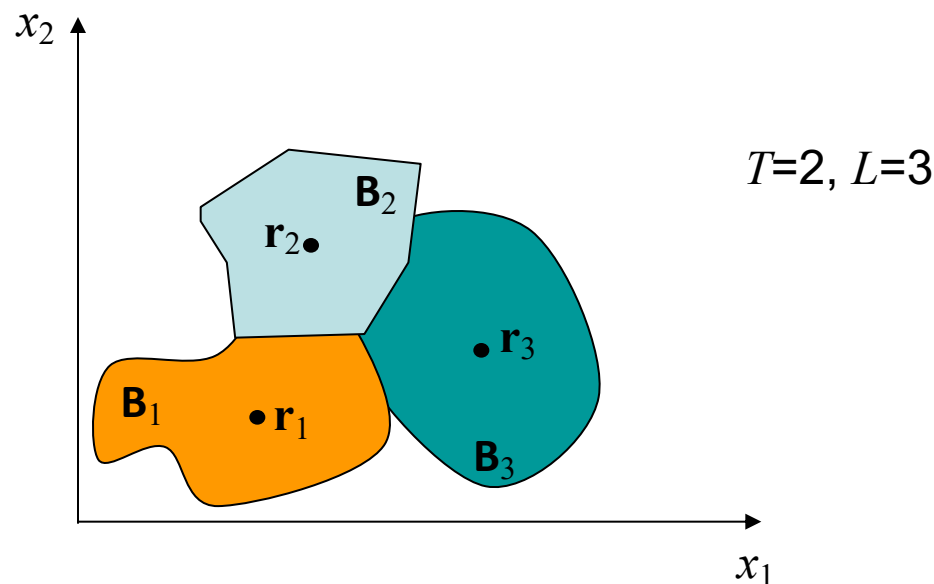
Vector Quantization

- Quantizer Q described by
 - T : the number of dimensions per vector
 - L : the number of reconstruction vectors
 - Set $L = \{1, 2, \dots, L\}$ of reconstruction vector indices
 - Boundary region B_l
 - Support space $\mathbf{B} = \{B_l\}$
 - \mathbf{r}_l : the set of reconstruction vectors (codewords)

$$Q(\mathbf{x}) = \mathbf{r}_l \quad \text{if } \mathbf{x} \in B_l, \quad l \in L$$

Partitions the support space of random vector \mathbf{x} into L regions
Represents \mathbf{x} with one of L discrete values

Illustration: 2D Vector Quantization



Represents vectors in this 2-D space with 3 reconstruction vectors

Bit-rate required after quantization $\sim \log_2 3$ per tuple $\sim 1/T(\log_2 3)$ per dimension

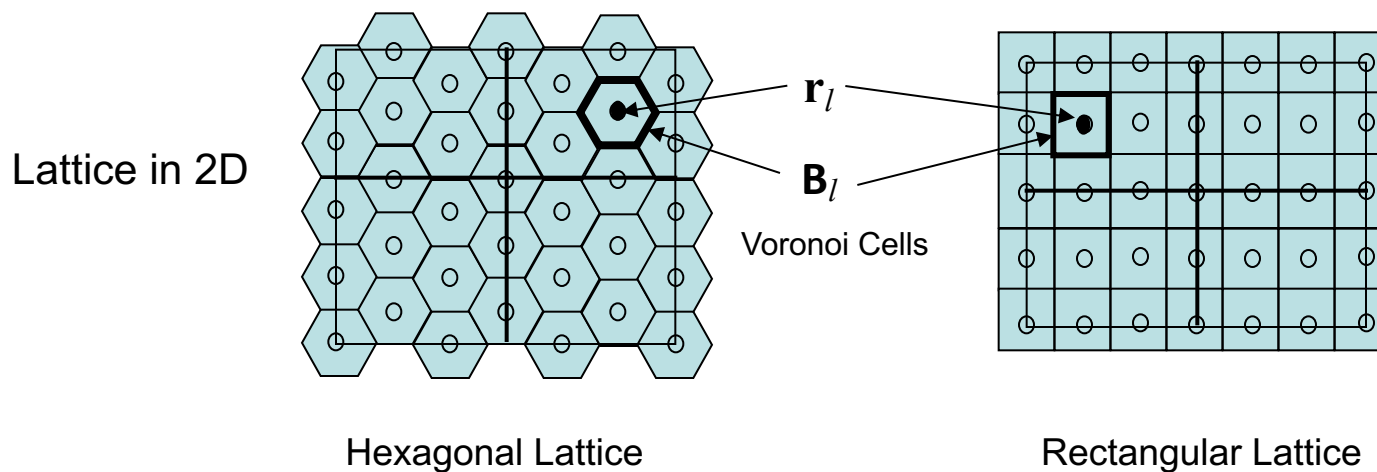
Quantization – analogous to clustering: Similar goals and algorithms

Vector Quantization Algorithms

- Linde Buzo and Gray (LBG) Algorithm
 - Analogous to the Lloyd-Max algorithm
- Choice of initial codebook
 - A representative subset of the training vectors
 - e.g., some face blocks, some shirt blocks, some background blocks
 - Scalar quantization in each dimension
 - Splitting...
- Nearest Neighbor (NN) algorithm [Equitz, 1984]
 - Start with the entire training set
 - Merge the two vectors that are closest into one vector equal to their mean
 - Repeat until the desired number of vectors is reached, or the distortion exceeds a certain threshold

Uniform Vector Quantizer

Uniform vector quantizer is a lattice quantizer



Recall: Analogy to sampling

Codebook contains all lattice point (reconstruction vectors)
Nearest neighbor property used for quantization