6.S079 Lecture 15

Sampling / Sketching

Last Time

- Storage layouts & the importance of locality

- Arranging data that is accessed together nearby on disk or memory can deliver order-of-magnitude performance improvements

- Several locality-increasing techniques:
  - Column-orientation
  - Partitioning (single and multi-dimensional)
  - Sorting
  - Compression

Handling New Data

- In most data science applications, we don’t update existing data
- Do need to deal with new data that is arriving
- If we have a complex data layout, e.g., sorted, partitioned, columns, inserting data will be slow, because we’ll have to rewrite all data
- Idea: just create a new partition for new data, and write your program to merge results from all partitions

Problem: Lots of Partitions

- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?
- Log structured merge tree: arrange so partitions merge a logarithmic number of times

Problem: Lots of Partitions

- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?
- Log structured merge tree: arrange so partitions merge a logarithmic number of times

Problem: Lots of Partitions

- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- Log structured merge tree: arrange so partitions merge a logarithmic number of times

Problem: Lots of Partitions

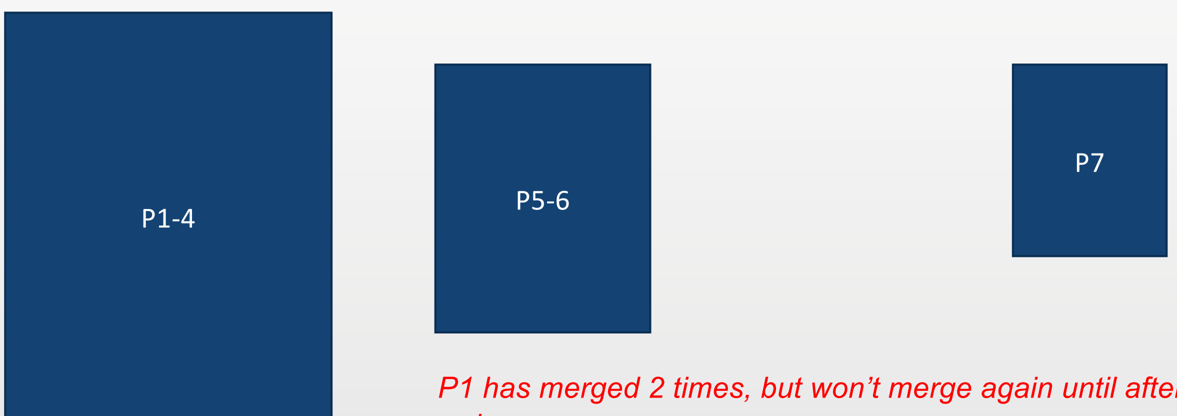
- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- Log structured merge tree: arrange so partitions merge a logarithmic number of times

Problem: Lots of Partitions

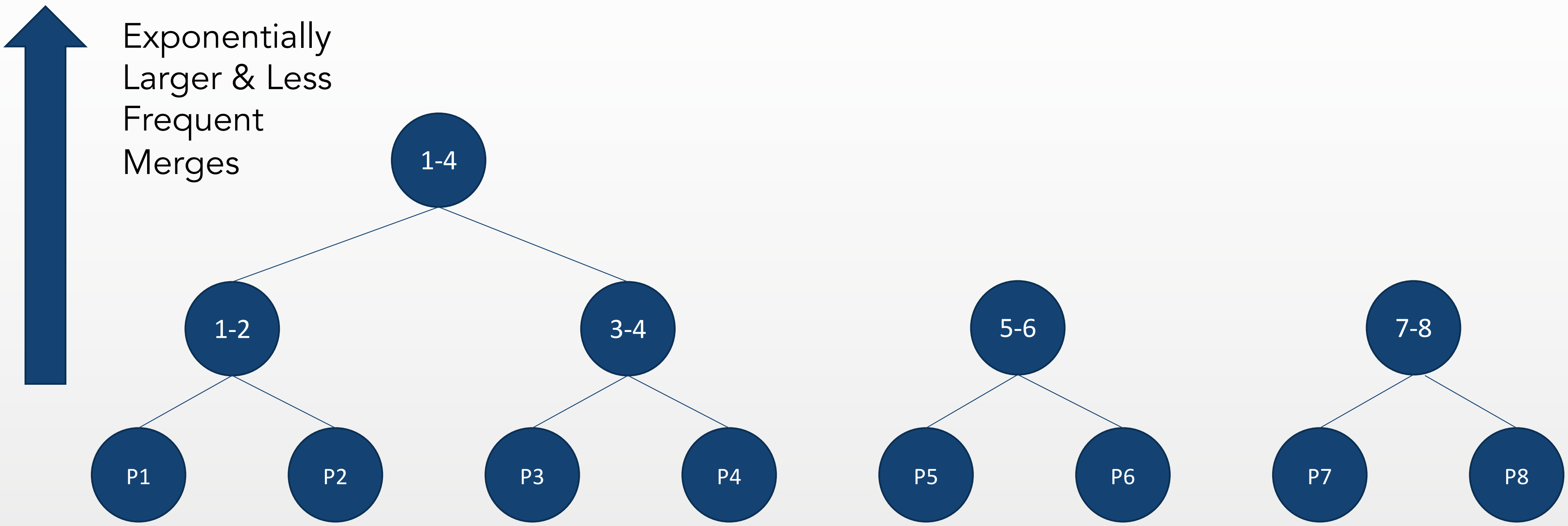
- Performance will degrade as you get many partitions
- Idea: merge some partitions together, but how?

- Log structured merge tree: arrange so partitions merge a logarithmic number of times



P1 has merged 2 times, but won't merge again until after 8 more partitions arrive

Log Structured Merge Tree



Do We Always Need to Process All the Data?

- For many data analytics applications, it may not be necessary to look at every record.

- E.g., suppose we want to see how revenue changed over the past 12 months

  - Could scan all data

<u>or</u>

  - Could **randomly sample** data and compute estimate / error bars

Sampling not because we do not have access to all the data, but because it can be more efficient to not look at all the data

Error Bars: Central Limit Theorem

- Given a population with a finite mean μ and a finite non-zero variance σ^2, the sampling distribution of the mean approaches a normal distribution with a mean of μ and a variance of σ^2/N as N, the sample size, increases.

- Here, the *sampling distribution of the mean* is the distribution of the means of samples of the dataset

- Allows us to estimate the mean, and estimate the error in the mean
  - μ = mean(sample)
  - $\boldsymbol{\sigma} = \frac{stddev(sample)}{\sqrt{N}}$, $stddev(sample) = \sqrt{\frac{\sum_{i\ in\ sample}(i - \mu)^2}{N}}$

Similar closed form solutions for sum, count, and other simple statistics

What if CLT Doesn't Apply

- E.g., suppose you want error bars on the median, or on percentiles in a histogram

- Or some complex predictive function, e.g., some ML algorithm

- The Nonparametric Bootstrap is a generic technique for this
  - Idea: repeatedly resample a sample

Bootstrap Method

Given a function F and a sample S of size N, with parameter K (the number of bootstraps)

Goal is +/- p confidence interval

For i in 1 .. K

- S_new = sample of size N of S *with* replacement
- Results[i] = F(S_new)

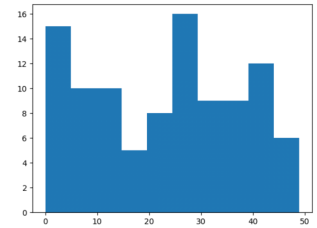
Sort results, return p, 1-p percentile of results

Example

Mean = 22.91

Data:

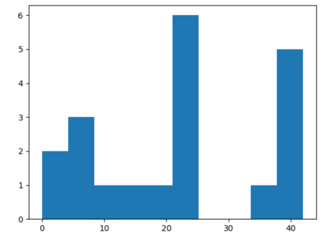
[36,23,7,25,27,31,27,10,11,8,21,4,41,0,20,5,0,36,40,10,12,31,24,2,28,8,9,25,48,43,40,2,26,0,25,32,9,0,10,33,1,23,7,39,18,32,16,40,4,42,28,28,26,42,0,45,25,10,13,31,3,11,28,25,23,16,31,22,6,34,19,48,27,48,39,40,6,3,28,26,19,34,38,42,1,47,22,7,36,38,35,35,42,49,41,40,11,10,1,1]



Sample:

[25,10,35,25,23,0,20,24,23,25,6,42,40,38,40,4,8,16,38,8]

Mean = 22.5



Resample 1: [42, 40, 8, 25, 0, 42, 24, 0, 16, 42, 23, 25, 25, 10, 40] Mean = 24.1

Resample 2: [23, 25, 10, 42, 23, 0, 0, 24, 23, 23, 38, 25, 16, 35, 25] Mean = 22.1

Resample 3: [6, 38, 40, 23, 23, 40, 23, 4, 8, 25, 4, 8, 25, 20, 0] Mean = 19.13

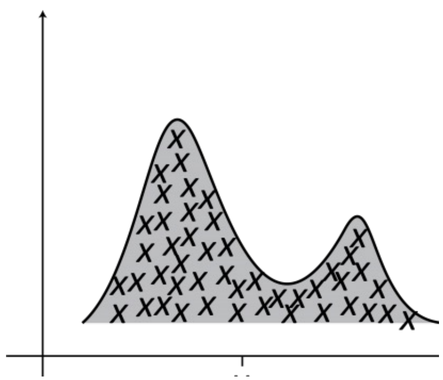
Resulting Means after 100 runs

```
14.93, 15.27, 15.33, 15.47, 16.60, 17.40, 17.53, 17.60, 17.80, 18.20,
18.27, 18.47, 18.47, 18.93, 18.93, 19.07, 19.07, 19.07, 19.13, 19.13,
19.53, 19.80, 19.80, 19.93, 20.00, 20.00, 20.13, 20.27, 20.40, 20.47,
20.60, 20.73, 20.80, 20.80, 21.07, 21.13, 21.13, 21.13, 21.20, 21.27,
21.33, 21.40, 21.47, 21.47, 21.87, 21.87, 22.13, 22.20, 22.27, 22.33,
22.33, 22.40, 22.73, 22.73, 22.80, 22.87, 22.93, 22.93, 23.00, 23.07,
23.13, 23.20, 23.20, 23.47, 23.53, 23.67, 23.67, 23.73, 23.73, 23.80,
23.93, 23.93, 23.93, 23.93, 24.00, 24.20, 24.20, 24.27, 24.47, 24.67,
24.80, 24.87, 24.87, 25.00, 25.13, 25.47, 25.47, 25.53, 26.07, 26.07,
26.07, 27.13, 27.33, 28.20, 28.47, 28.87, 28.87, 30.00, 30.53, 32.40,
```

Confidence interval of mean 16.6 … 28.87

Why Does This Work

- A random sample is an approximation of the distribution of the data
  - If it's big enough, it's a good approximation



Samples approximate the true distribution well

- Resampling the sample is *close* to resampling from the original data
  - Variation in those samples captures variation in the original data
  - Of course, it will miss outliers, extrema, etc.
  - But it will work well for a variety of descriptive statistics, including quantiles, regression errors, precision/recall estimates, etc.

When Doesn’t This Work

- Your sample needs to be big enough (N > 20 is a rule of thumb, but it will vary a lot depending on data)
- It won’t work for extrema (e.g., min / max)
- It won’t work well for highly structured data (i.e., you can’t randomly sample a graph, compute the average connectivity, and expect to get something meaningful)
- It won’t work if your sample is not truly random

Bootstrap Demo

BlinkDB

Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. ***In ACM EuroSys 2013***

BlinkDB Goal

- **Observation:** Many applications can tolerate quick, approximate answers over data
- **Trade-off**: few percent error for up orders of magnitude in efficiency
- Acceptable in decision support, recommendation system, diagnosis, root cause analysis

Overview

Problem

Users are overwhelmed by data volumes AND increasingly want to compute sophisticated statistics over their data. Existing database systems do not satisfy their needs.

Goal

Provide interactive ad-hoc analytical (SQL) queries over very large data sets.

Basic Approach

Run queries over stored/precomputed samples, providing answers with bounded errors for arbitrary functions.

Challenges/Solutions

Generality: Accurate error estimates for complex SQL statements and user-defined functions

- Use bootstrap to providing error estimates for arbitrary user-defined (differentiable) functions

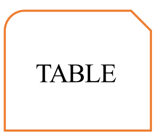
Flexibility/Reliability: Accurate estimations of response times for ad hoc queries (including over small domains)

- Use stratified sampling rather than random sampling

Parallelism/Scalability: Sub-second latencies for parallel queries running on hundreds of machines

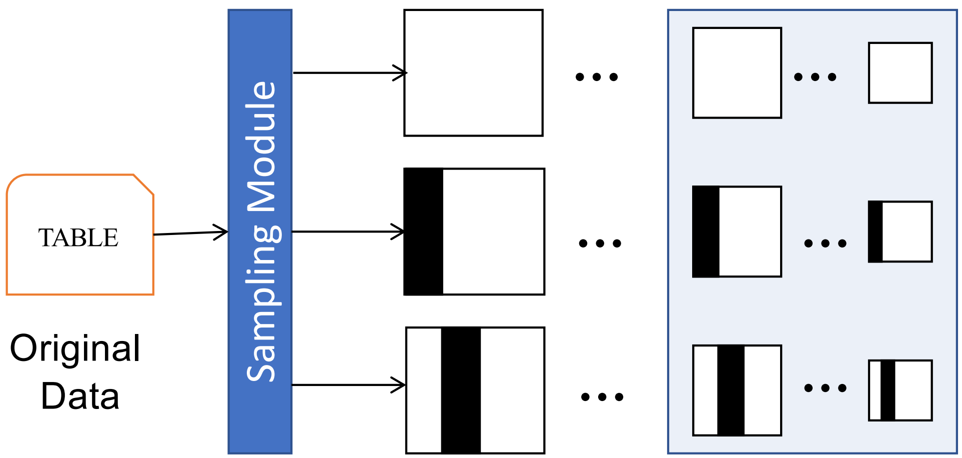
- Not doing online aggregation, but pre-computing samples
- Optimization problem!

System Architecture



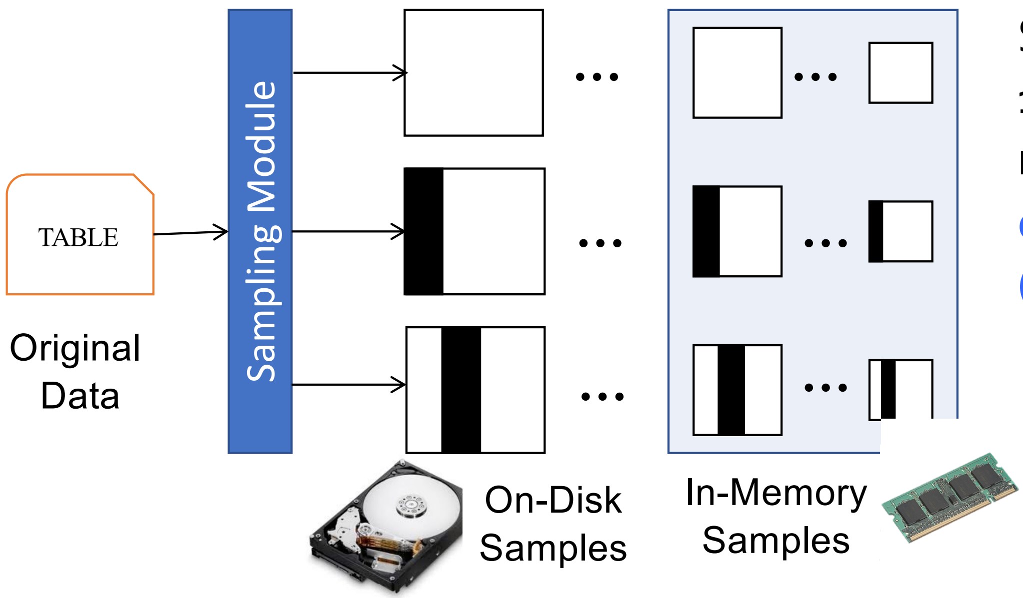
Original
Data

System Architecture



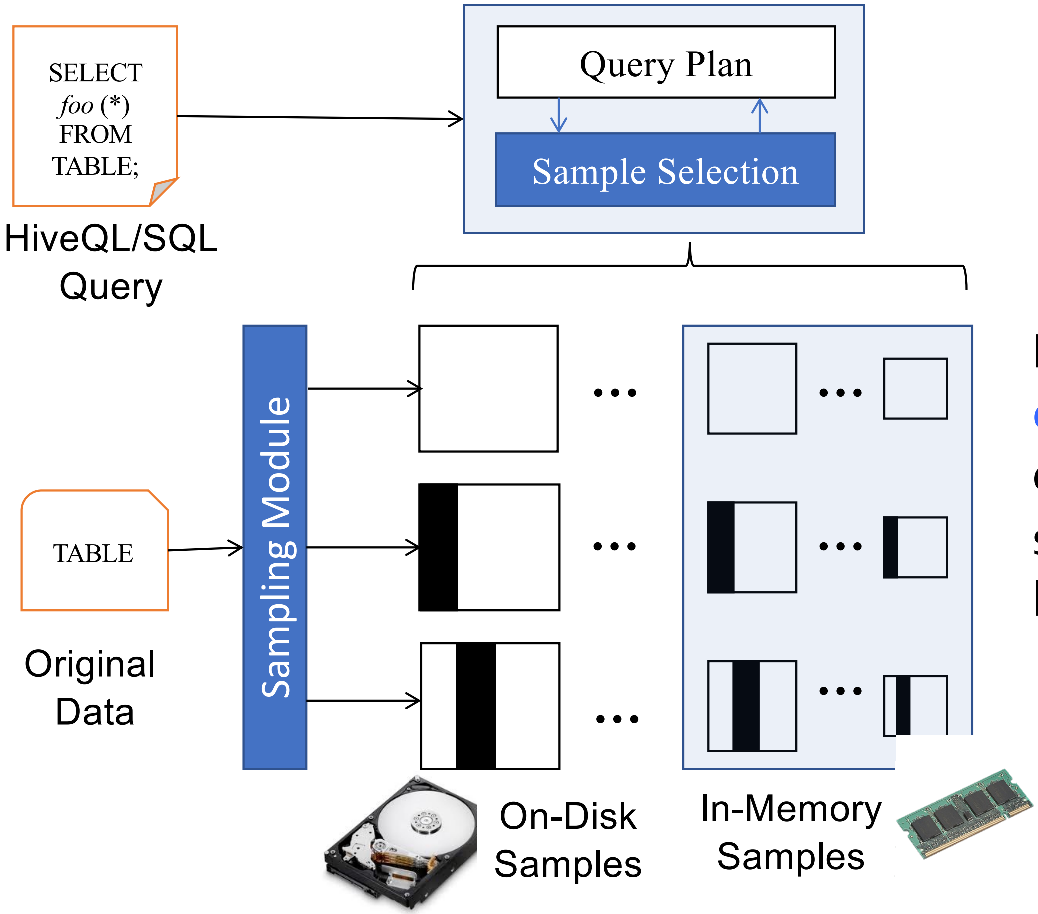
Offline-sampling: multiple data samples at various granularities and across different dimensions (columns)

Initial Prototype



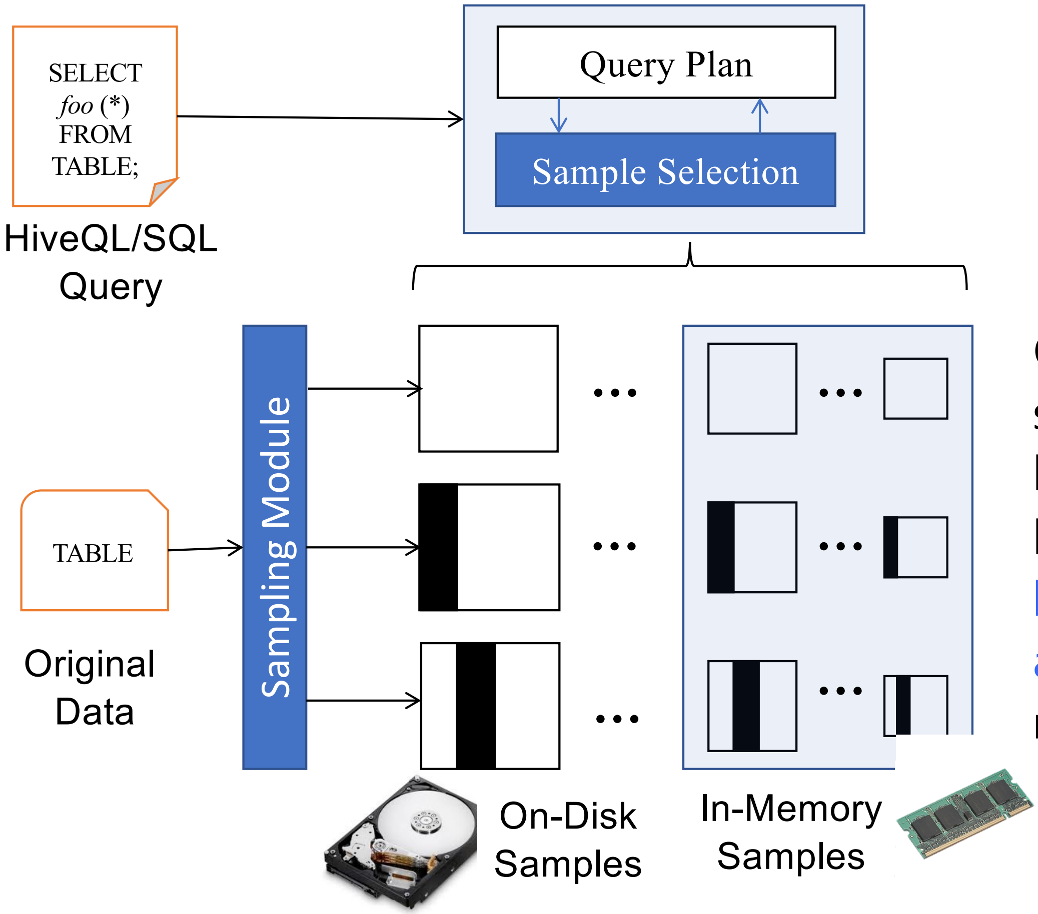
Samples striped over 100s or 1,000s of machines both on disks and in-memory (i.e., RDDs)

System Architecture



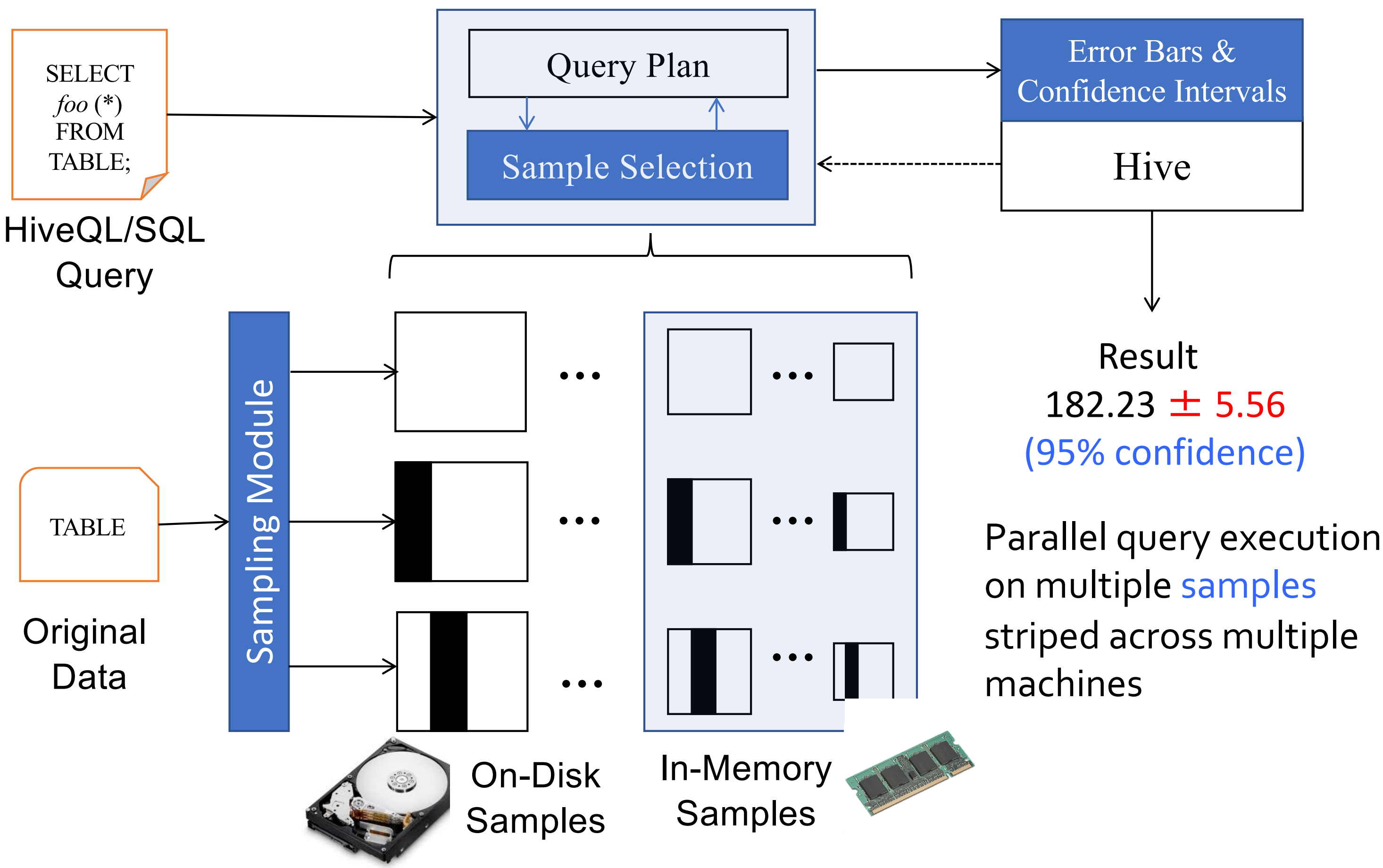
Predict cost and error for ad-hoc queries using smaller samples and historical context

System Architecture

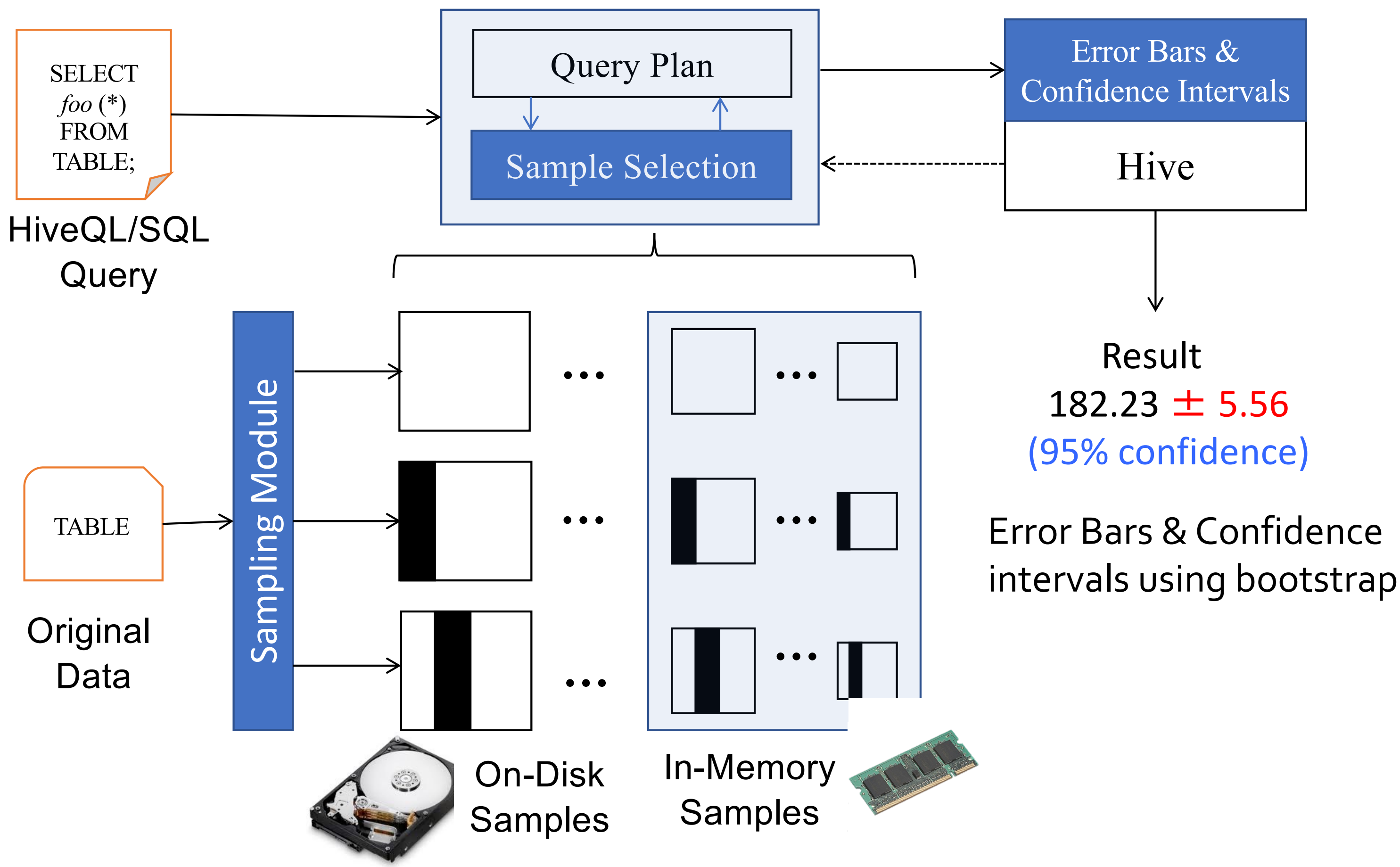


Online sample selection to pick best sample(s) based on query latency and accuracy requirements

System Architecture



System Architecture



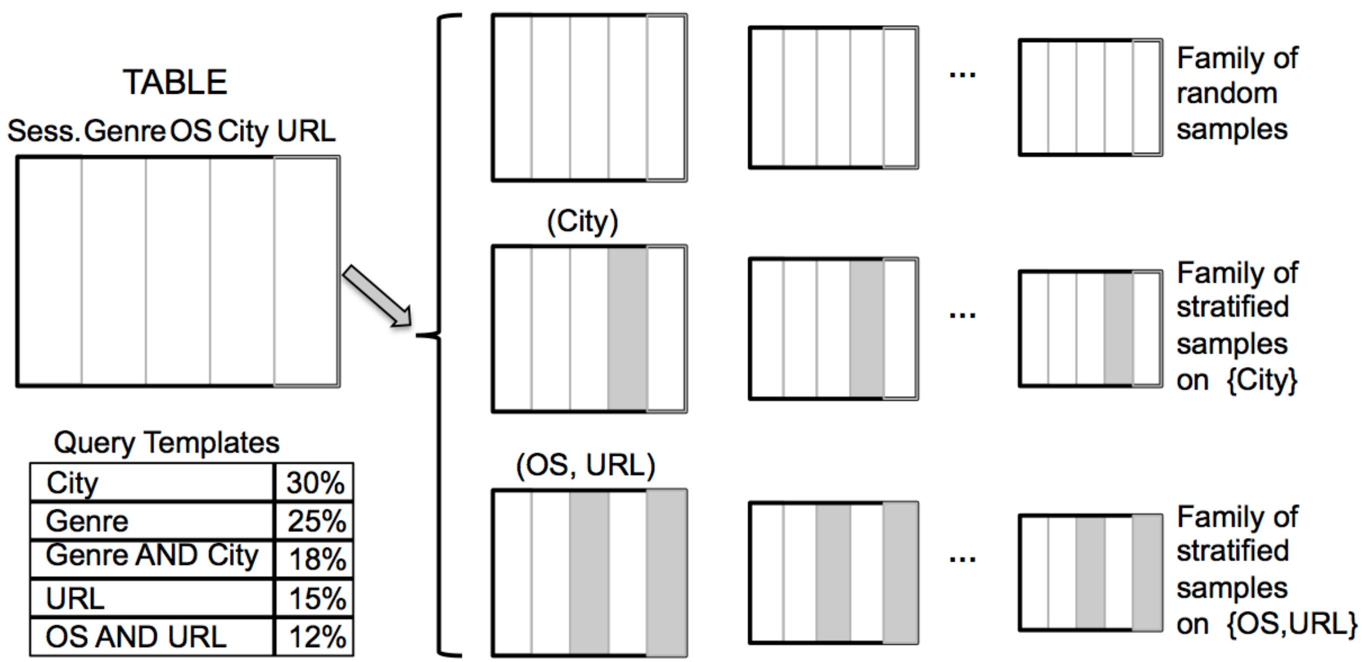
Handling Rare Values

- Some values in tables *much* less popular

```
Q1: SELECT avg(Salary) FROM employees WHERE  city='New York'
Q2: SELECT avg(Salary) FROM employees WHERE city='Cambridge'
```

Solution: Stratified sampling – only sample values that appear more than K times; preserve other values

Example



Query Templates

City	30%
Genre	25%
Genre AND City	18%
URL	15%
OS AND URL	12%

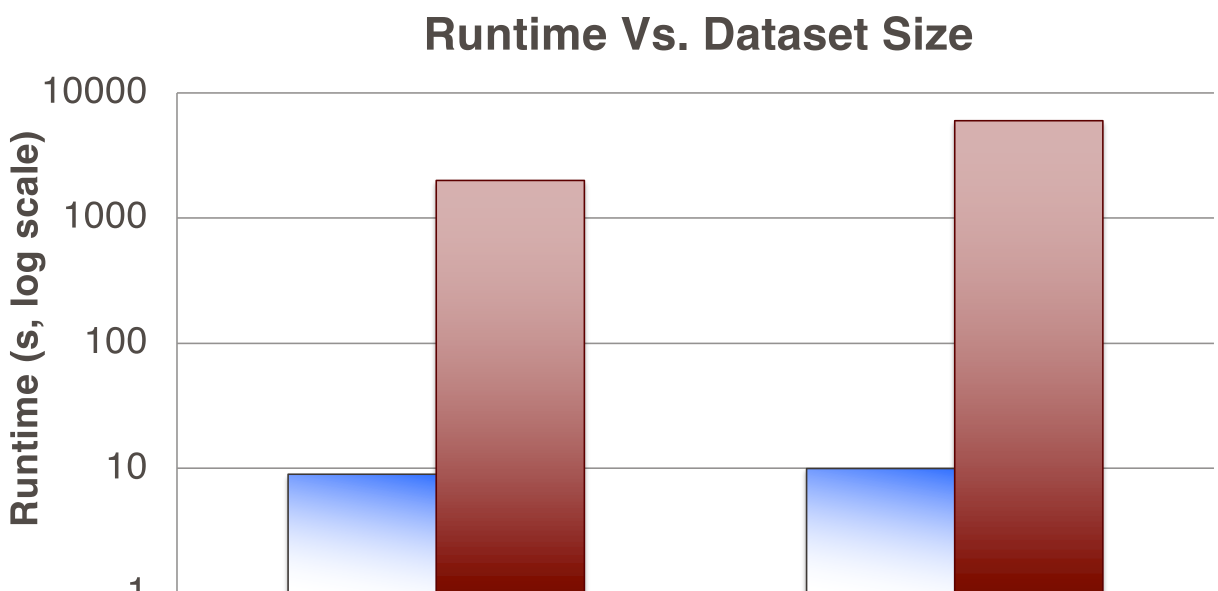
What Samples to Create?

1. Always maintain a uniform sample
2. For stratified samples, start from past "query templates"
3. Choose the combinations of columns that are "best" for those templates
   - Favor Non-uniform columns
4. Avoid "over-fitting" the past workload
   - Favor sample families useful for answering queries not captured by exiting templates

Experimental Setup

- 30-day log of media accesses by users from a video analytics company. Raw data 17 TB, partitioned this data across 100 nodes.
- Log of 20,000 queries (a sample of 200 queries had 42 templates).

Results



	2.5 TB	7.5 TB
BlinkDB (1% error)	9	10
Hive	2000	6000

BlinkDB – Summary

- A massively parallel DB that supports ad-hoc queries with error and response-time bounds.
- An optimal strategy for building & maintaining multi-dimensional, multi-granularity samples
- Dynamic Query Cost Estimation and Sample Selection

Break

Here is a photo-realistic image depicting a diverse group of students resting comfortably on a college campus. This scene captures the essence of a pleasant spring day, with students engaging in various activities such as chatting, reading, and napping under the shade of large trees.

Extreme Statistics

- What about cases where you need to estimate the max, min, # of distinct values etc?
- Sampling won't work
- **No free lunch**: Need to look at all of the values
- For min/max, can keep a running value
- But what about distinct values, top-N, etc?

Sketching Algorithms

Approximate (probabilistic) algorithms for estimating these types of statistics over (large) data sets

Count distinct: hyperloglog
Heavy hitters (top K): countmin
Quantiles (median): quantile sketch
…

Today : hyperloglog , countmin

How many samples on average until there are k trailing zeros?

25	0b110010
10	0b101000
35	0b100011
25	0b110010
23	0b101110
0	0b000000
20	0b101000
24	0b110000
23	0b101110
25	0b110010
6	0b110000
42	0b101010
40	0b101000
38	0b100110
40	0b101000
4	0b100000
8	0b100000
16	0b100000
38	0b100110
8	0b100000

Clicker:

a. k
b. 1
c. 2^k
d. k^2

https://clicker.mit.edu/6.S079/

How many samples on average until there are k trailing zeros?

25	0b110010
10	0b101000
35	0b100011
25	0b110010
23	0b101110
0	0b000000
20	0b101000
24	0b110000
23	0b101110
25	0b110010
6	0b110000
42	0b101010
40	0b101000
38	0b100110
40	0b101000
4	0b100000
8	0b100000
16	0b100000
38	0b100110
8	0b100000

Clicker:

a. k
b. 1
c. 2^k
d. k^2

https://clicker.mit.edu/6.S079/

Hyperloglog Algorithm – Approach 0

Given a vector of values, V, compute H(v) for all v in V

H is a hash function that goes from v to a large random integer

```
MaxZeros = 0
For each h in H(v) ∀ v in V:
   Zeros = count the number of leading zeros in h
   MaxZeros = max(Zeros, MaxZeros)
```

Distinct vals ~= 2^{MaxZeros}

HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm

Philippe Flajolet[1] and Éric Fusy[1] and Olivier Gandouet[2] and Frédéric Meunier[1]

Discussion

- This is an accurate estimator, but it is noisy
- We can do better by averaging a bunch of estimators
- Could repeat the previous algorithm N times, but requires computing N hashes per data item, which is expensive
- This is the problem hyperloglog tries to solve

Hyperloglog Algorithm – Approach 1

Idea: split hash value into m "bucket" bits and 128 – m "value" bits; store 2^m max's

Creates 2^m hashes out of a single hash

Given a vector of values, V, compute H(v) for all v in V

H is a hash function that goes from v to a large random integer

```
MaxZeros = [0, 0, ..] // length 2^m
For each h in H(v) ∀ v in V :
      bucket = bits 0 … m-1 of h
      value = bits m … 128 of h
      zeros = count the number of leading zeros in value
      MaxZeros[bucket] = max(zeros, MaxZeros[bucket])
```

Distinct vals = avg($2^{MaxZeros[0]}$, …, $2^{MaxZeros[2^m]}$)

Algorithm 1 Discussion

- Paper shows that taking the harmonic mean of the estimates, instead of the average, results in a better estimate. H(1,3,4) =

$$\left(\frac{1^{-1}+4^{-1}+4^{-1}}{3}\right)^{-1}=\frac{3}{\frac{1}{1}+\frac{1}{4}+\frac{1}{4}}=\frac{3}{1.5}=2\,.$$

- Error is 1.04/sqrt(m), where m is the number of maximums we maintain
- Discarding outlier buckets also helps
- Also can be updated – i.e., merged with another set of counters to get a new estimate of the cardinality

HyperLogLog Demo

CountMin

- Suppose we have an infinite stream of data (e.g., users arriving at a website) and we want to estimate some property over them, i.e.:
  - Most frequent visitors
  - Most popular OS version
  - …
- Could maintain running counts, but this may require unbounded state (i.e., if number of users is unbounded)
- CountMin provides a way to estimate such counts

Count-Min Sketch

Graham Cormode
AT&T Labs–Research, graham@research.att.com

Simple Idea #1

- Keep a table T with N elements, initialized to 0
- Suppose we have items with types (i.e., userids, OSes)
- For every item,
  - compute x=hash(item.type) mod N
  - increment T[x]

- To estimate the frequency of a type t, return T[hash(t)]
- Will be correct as long as no collisions in the hash function
- With collisions, can overestimate
  - If N < number of types, will be (some) collisions

Better Idea

- Keep M tables, each with N elements
- Each table uses a different hash function, H_1, H_2, …

N Elements

M Tables

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Better Idea

- Keep M tables, each with N elements
- Each table uses a different hash function, H_1, H_2, …

N Elements

M Tables

0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Compute H_1(item.type),

Value between 0 and N

Better Idea

- Keep M tables, each with N elements
- Each table uses a different hash function, H_1, H_2, …

N Elements

M Tables

0	1	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0		0	0	0	0	0
0	0	0	0	0	0	0	0

Compute H_1(item.type), H_2(item.type),

Better Idea

- Keep M tables, each with N elements
- Each table uses a different hash function, H_1, H_2, …

N Elements

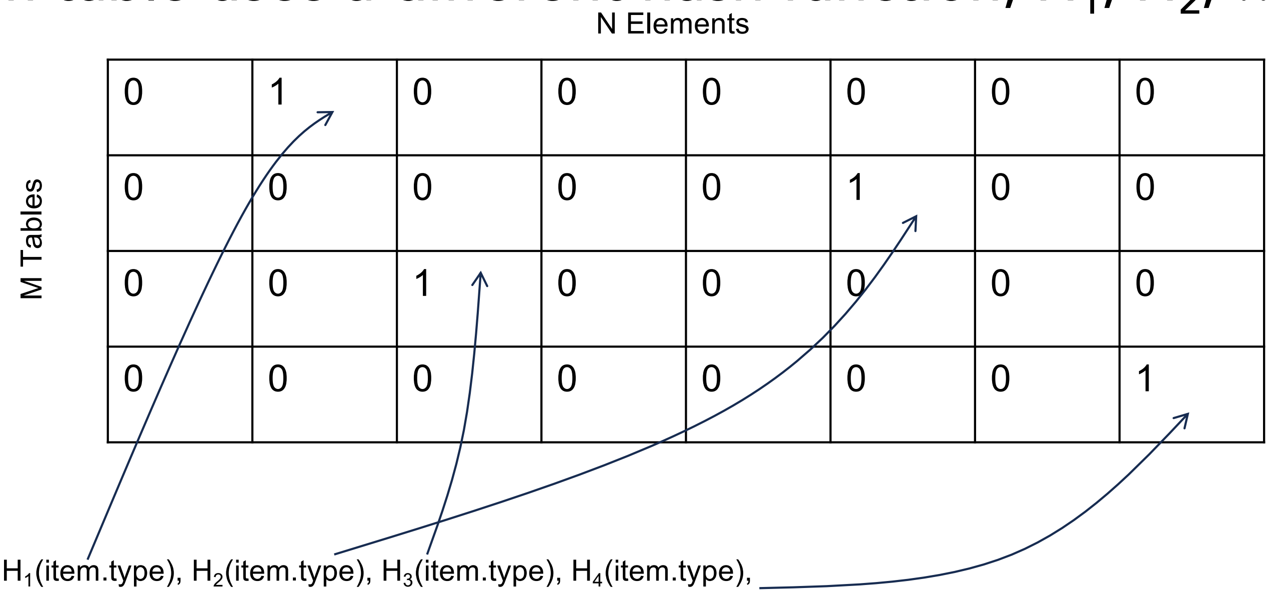
M Tables

0	1	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0

Compute H_1(item.type), H_2(item.type), H_3(item.type),

Better Idea

- Keep M tables, each with N elements
- Each table uses a different hash function, H_1, H_2, …



0	1	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1

Better Idea (lookup)

- Suppose we want to compute the frequency of type t
- Compute $H_1(t), \ldots, H_M(t)$
- Lookup in each of the M tables, i.e.:
  - $T_1(H_1(t)), \ldots, T_M(H_M(t))$
- Then compute $\min(T_1(H_1(t)), \ldots, T_M(H_M(t)))$ as estimate of number of occurrences of t

- This will only over-estimate if *all* of the hash functions have collided

Lookup Example

- Suppose we want to estimate frequency of type i

N Elements

M Tables

3	7	7	9	11	14	17	18
5	6	3	0	1	4	8	22
99	4	6	7	8	2	33	6
4	7	2	8	9	2	2	12

H_1(item.type), H_2(item.type), H_3(item.type), H_4(item.type),

7 4 6 12

Min = 4

CountMin Demo

Summary

- Sampling can be an effective way to dramatically reduce computation over large data sets
- Accurate for a variety of statistics, e.g., mean, sum, etc
- Bootstrap enables use of sampling over a larger set of statistics, e.g., quantiles, etc.

- For extreme value statistics, heavy hitters, etc – sketching algorithms provide a way to compute these in sublinear storage (but still require looking at every value)