Georgia Tech

Caching Statistics to Accelerate Exploratory Data Analytics

Apaar Shanker

DATA ANALYTICS
USING DEEP LEARNING
GT CS 8803 // FALL 2018 //

CREATING THE NEXT®

TODAY'S AGENDA

- Problem Overview
- Key Ideas
- Proposed Solutions
- Validation and Evaluation Methods
- Benchmarks
- Future Work

Our Objective

- A typical data scientist’s workflow involves computing certain functions - such as mean, variance etc., on overlapping, or hierarchical ranges in the data-table.
- Currently, these functions need to be recomputed for the entire range each time they are invoked.
- This involves wasteful recomputation and increases lag. We want to do better by memoizing operations through use of statistical primitives and better indexing of data.

Statistics		Basic Aggregates				
Type	**Formula**	$\sum x$	$\sum x^2$	$\sum xy$	$\sum y^2$	$\sum y$
Mean (avg)	$\frac{\sum x_i}{n}$	■				
Root Mean Square (rms)	$\sqrt{\frac{1}{n} \cdot \sum x^2}$		■			
Variance (var)	$\frac{\sum x_i^2 - n \cdot \mathtt{avg}(x)^2}{n}$	■	■			
Standard Deviation (std)	$\sqrt{\frac{\sum x_i^2 - n \cdot \mathtt{avg}(x)^2}{n}}$	■	■			
Sample Kurtosis (kur)	$\frac{1}{n}\sum(\frac{x_i - \mathtt{avg}(x)}{\mathtt{std}(x)})^4 - 3$	■	■			
Sample Covariance (cov)	$\frac{\sum x_i \cdot y_i}{n} - \frac{\sum x_i \sum y_i}{n^2}$	■		■		■
Simple Linear Regression (slr)	$\frac{\mathtt{cov}(x,y)}{\mathtt{var}(x)}, \mathtt{avg}(x), \mathtt{avg}(y)$	■	■	■		■
Sample Correlation (corr)	$\frac{n \cdot \sum x_i \cdot y_i - \sum x_i \cdot \sum y_i}{\sqrt{n \cdot \sum x_i^2 - (\sum x_i)^2}\sqrt{n \cdot \sum y_i^2 - (\sum y_i)^2}}$	■	■	■	■	■

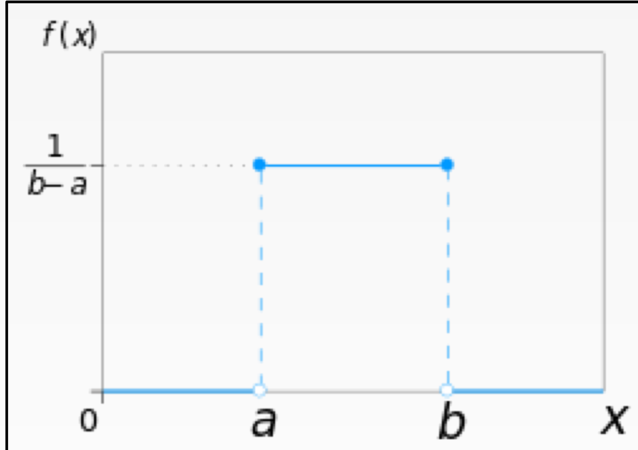
Table 1: Data Canopy synthesizes statistics from a library of basic aggregates.

Courtesy: Data Canopy Paper

Accelerating Statistical Query Process through “**aggregation of primitives**” and efficient “**adaptive inverse-indexing**” techniques.

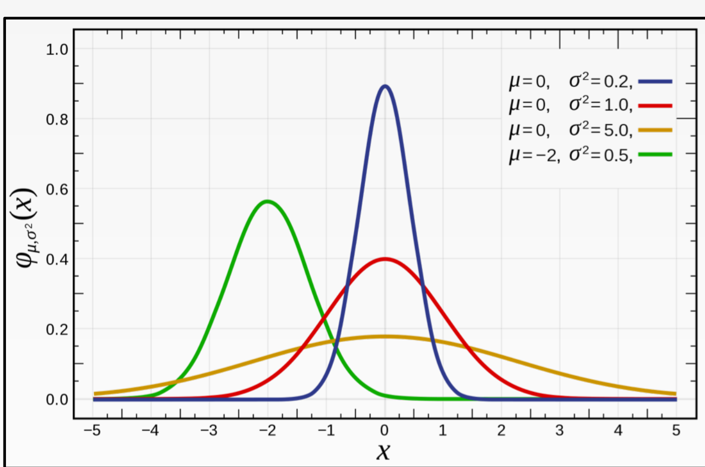
Different Types of Data

Uniform Distribution



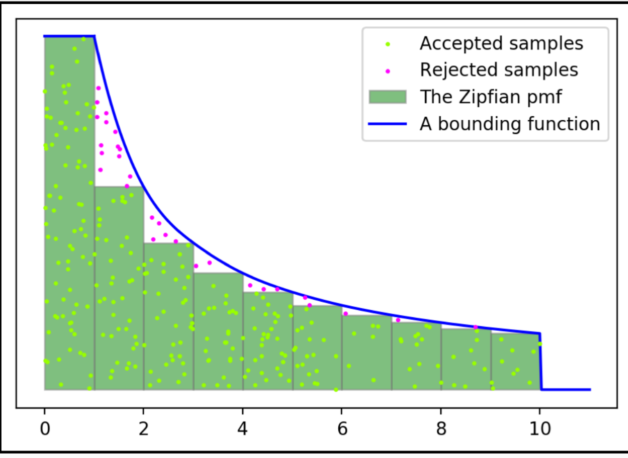
- Customers with birthdays on Monday
- Customers with names starting with “A”

Normal Distribution



- Age of Consumers
- Credit Rating of loan seekers

Zipfian Distribution



- Salary of Employees
- Price of Cars
- frequency of words in natural languages

Different Types of Queries

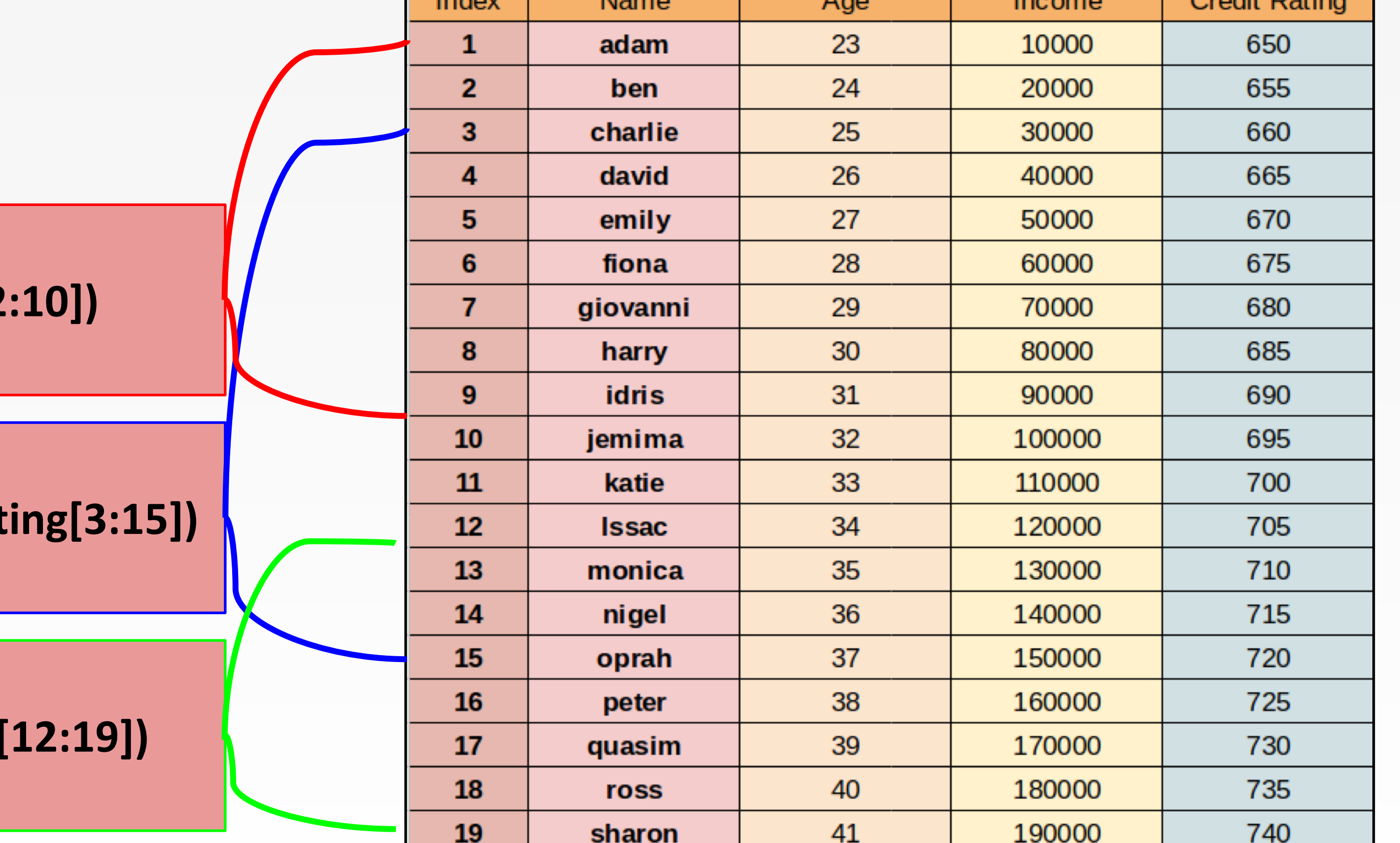
Interval Queries

Range Queries

SUM(Age[2:10])

Mean(Credit Rating[3:15])

Variance(Age[12:19])



Index	Name	Age	Income	Credit Rating
1	adam	23	10000	650
2	ben	24	20000	655
3	charlie	25	30000	660
4	david	26	40000	665
5	emily	27	50000	670
6	fiona	28	60000	675
7	giovanni	29	70000	680
8	harry	30	80000	685
9	idris	31	90000	690
10	jemima	32	100000	695
11	katie	33	110000	700
12	lssac	34	120000	705
13	monica	35	130000	710
14	nigel	36	140000	715
15	oprah	37	150000	720
16	peter	38	160000	725
17	quasim	39	170000	730
18	ross	40	180000	735
19	sharon	41	190000	740

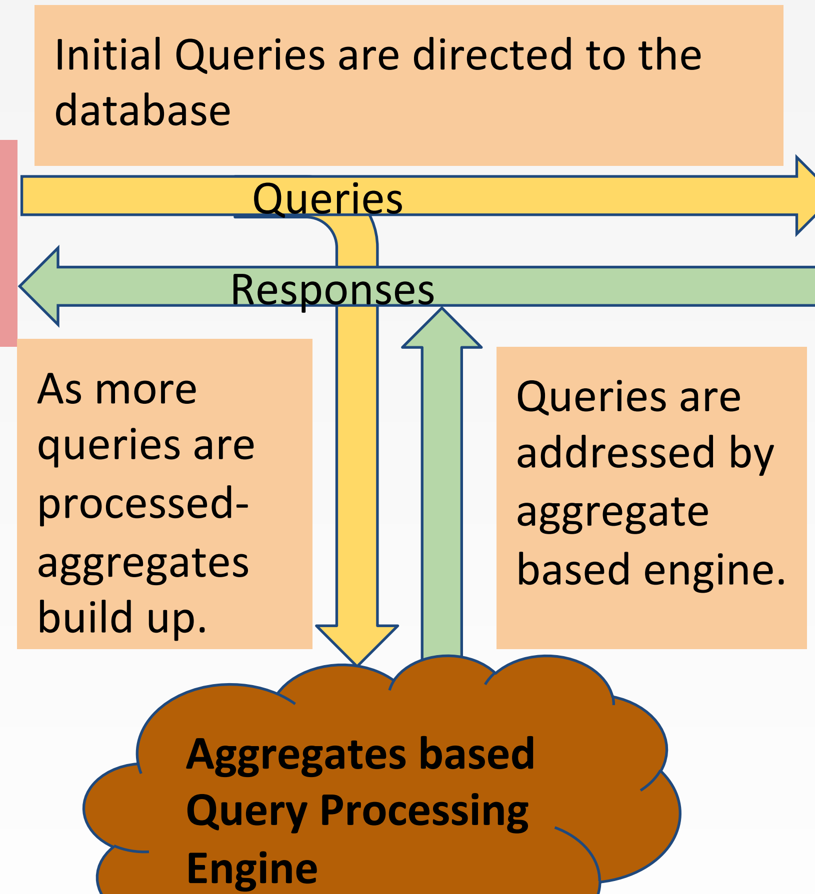
Where(Age>20 and Age < 45)

income = Table[‘John’]].Income()

Where(x>income-10 and Age < income+10)

Accelerating Queries Through Aggregation

- Where(Age>20 and Age < 45)
- SUM(Age[2:10])



Index	Name	Age	Income	Credit Rating
1	adam	23	10000	650
2	ben	24	20000	655
3	charlie	25	30000	660
4	david	26	40000	665
5	emily	27	50000	670
6	fiona	28	60000	675
7	giovanni	29	70000	680
8	harry	30	80000	685
9	idris	31	90000	690
10	jemima	32	100000	695
11	katie	33	110000	700
12	Issac	34	120000	705
13	monica	35	130000	710
14	nigel	36	140000	715
15	oprah	37	150000	720
16	peter	38	160000	725
17	quasim	39	170000	730
18	ross	40	180000	735
19	sharon	41	190000	740

For Interval Queries: Segment Trees

1 A[0:6]

2 A[0:3] 3 A[4:6]

4 A[0:1] 5 A[2:3] 6 A[4:5] 7 A[6:6]

8 A[0:0] 9 A[1:1] 10 A[2:2] 11 A[3:3] 12 A[4:4] 13 A[5:5]

Segment Tree

tree [1] = A[0:6]
tree [2] = A[0:3]
tree [3] = A[4:6]
tree [4] = A[0:1]
tree [5] = A[2:3]
tree [6] = A[4:5]
tree [7] = A[6:6]
tree [8] = A[0:0]
tree [9] = A[1:1]
tree [10] = A[2:2]
tree [11] = A[3:3]
tree [12] = A[4:4]
tree [13] = A[5:5]

Segment Tree represented as linear array

More Information about Segment Tree

Segment tree is a **static binary tree** used for storing information about intervals or segments

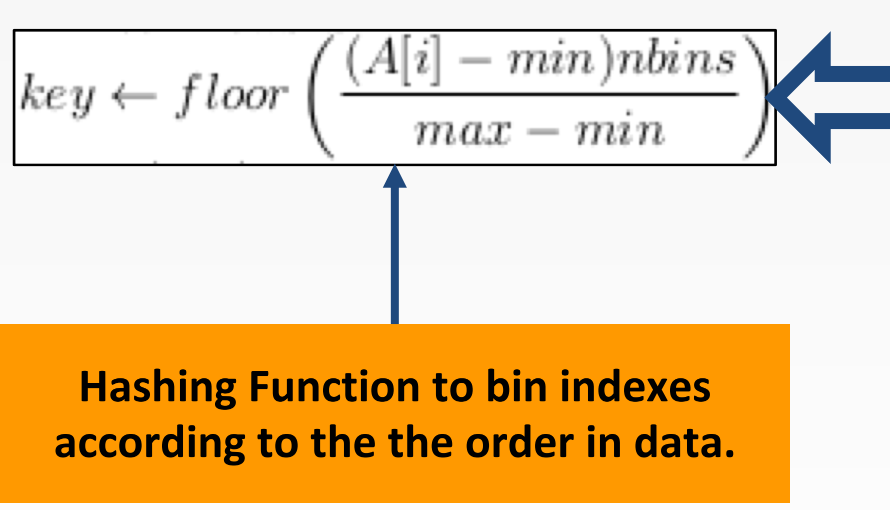
Storage: O(nlog(n))
Build: O(n(log(n)))

This data structure can used to cache information like **sum, sum of squares, min, max** etc. for a hierarchy of contiguous ranges in any given data structure.

For Range Queries: Hash-table based Inverted Index

Map : A key-Value Store

Keys	Indexes
1	31, 33, 78, 90
2	1, 20, 37
3	213, 23
.	.
.	.
.	.
.	.
.	.
.	.
.	.
.	.
.	.
nbins	61, 44, 28, 77, 89

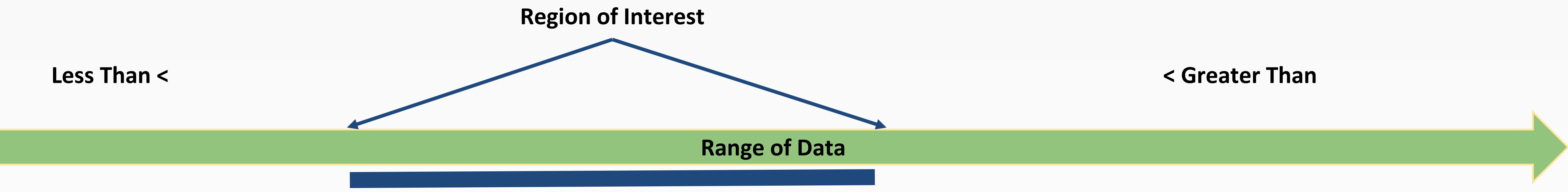


Index	Name	Age	Income	Credit Rating
1	adam	23	10000	650
2	ben	24	20000	655
3	charlie	25	30000	660
4	david	26	40000	665
5	emily	27	50000	670
6	fiona	28	60000	675
7	giovanni	29	70000	680
8	harry	30	80000	685
9	idris	31	90000	690
10	jemima	32	100000	695
11	katie	33	110000	700
12	Issac	34	120000	705
13	monica	35	130000	710
14	nigel	36	140000	715
15	oprah	37	150000	720
16	peter	38	160000	725
17	quasim	39	170000	730
18	ross	40	180000	735
19	sharon	41	190000	740

Assumes: data falls within a min/max range

Adaptive Inverted Indexes: Database Cracking

- Previous approach leads to an extraordinarily **high initial cost**.
- Also, the data structure is not adaptive and can get unbalanced depending on the distribution of the underlying data.
- A better strategy would be to employ the notion of database cracking.



Adaptive Inverted Indexes: Database Cracking

Region of Interest

k1 < Greater Than

K2 < Greater Than

Range of Data

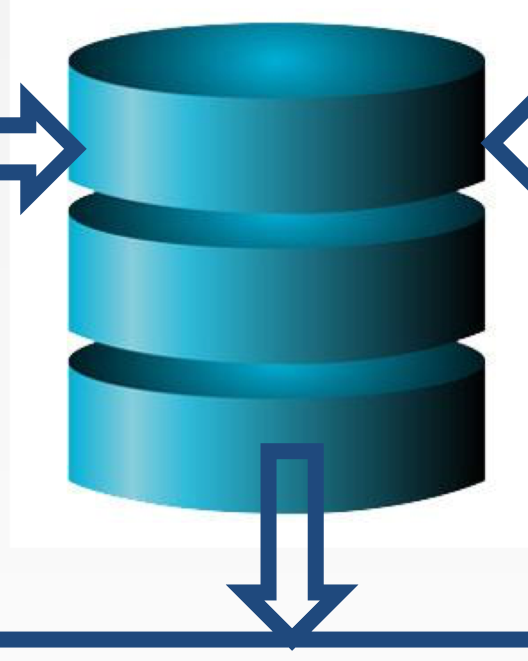
Only generate inverted indices for data in this user queried range.

key : Value Pairing

(k1, k2) : [list of indices found in this range]

High Level Description of Implementation

Database Generator
- A 2-D array of floating point values
- Populate the columns using uniform, normal or zipfian distribution



Workload Generator
- Interval Queries: **sum**(a[i1:i2])
- Range Query: **greater than, less than**, k1 < x < k2.

Primitive Aggregates
- Segment Tree

Inverted Indexes
- hashtable for indexes
- DB Cracking Solution

Hardware Description

Device Used

12 Intel Xeon CPU E5-2650 v4 @ 2.20 GHz

128 gigabytes of RAM,

Running Ubuntu 16.04 OS.

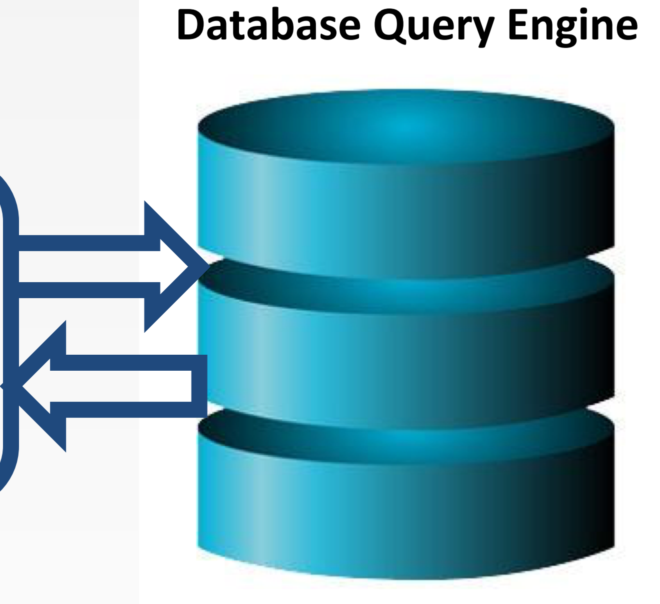
Evaluation and Validation

Naive Query Implementation for Validating Correctness

```
for i in range(N)
        if r1 < data[i] < r2:
                store.add(i)

print all i's in store
```

- Populate the columns using uniform, normal or zipfian distribution
- Raise Range Queries and record processing time.



Results

InvIndx and naive

InvIndx
naive

2
1.5
1
0.5
0

1.00E+07 5.00E+07 1.00E+08

Uniform , Normal and Zipfian

Uniform Normal Zipfian

0.03
0.02
0.01
0

DB Cracking: Work in Progress!

Objectives Achieved

- ➢ 80% : All database components and aggregate generation tool implemented
- ➢ 100% : if adaptivity/ database cracking is achieved
- ➢ 100-110%: Speeding up a practical workflow using Aggregate + Cracking

Future Work

- ➢ Implement persistence and study effect of storage latency
- ➢ Study impact of cracking on an online query environment
- ➢ Study different practical workloads

References

[1] Abdul Wasay et al. Data Canopy: Accelerating Exploratory Statistical Analysis. SIGMOD 2017.

[2] Abdul Wasay et al. Queriosity: Automated Data Exploration. 2016

[3] First use of segment tree and original reference

[4] First use of B+ tree

[5] Usage of hashing/B+ trees for orthogonal range queries.

Examples of Reusable Computation

- Query 1: The data scientist requests mean temperatures for each day

- Query 2: The data scientist requests mean temperatures for each week.

- Query 3: The data scientist requests variances in temperature for every two weeks.

B+ Tree/ Range Tree (Adaptivity Introduced)

An example of a 1-dimensional range tree. Each node which is not a leaf stores the maximum value in its left subtree.

Time: $O(\log^{d-1} n + k)$

Space: $O\left(n\left(\frac{\log n}{\log\log n}\right)^{d-1}\right)$