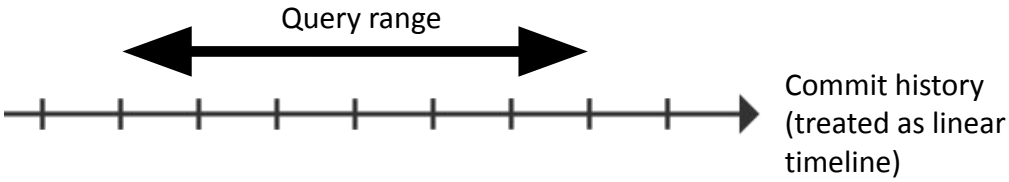
RepoStats

by Rui Hong

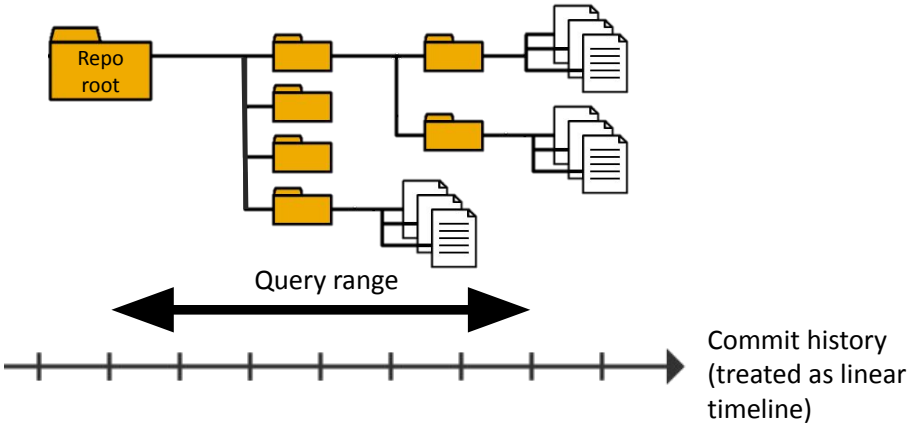
Answering Range Queries

The purpose of answering range queries on a git repo is to find commits/modules/files of interest within a large git repo.

What is a range query?

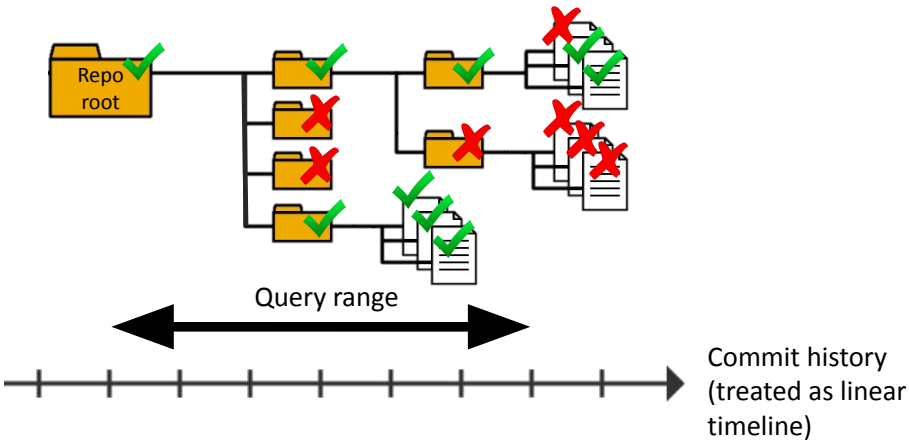


What is a range query?



Each commit is a version of the directory tree. And in each query we want to find out what are the subfolders to a certain depth that contains any file within its entire subtree that satisfies our query parameters

What is a range query?



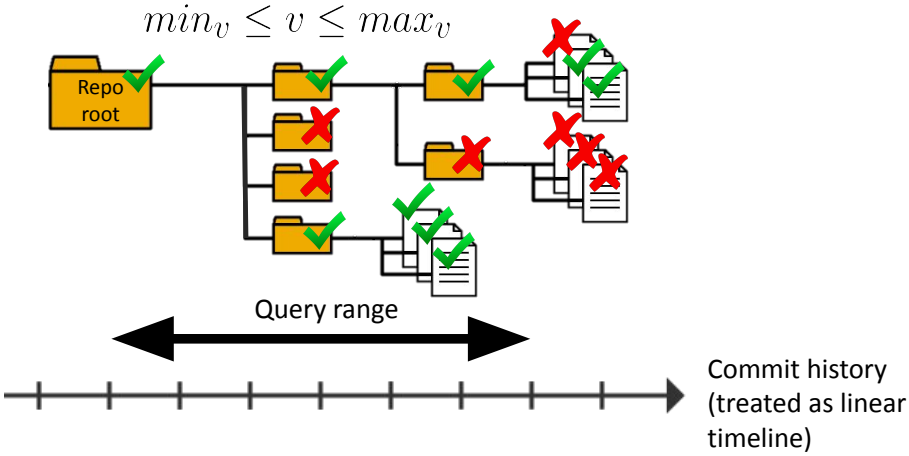
For example, this folder should appear in our results set because …

Conversely …

What is a range query?

$for\ each\ condition\ v$

$min_v \leq v \leq max_v$



Each file has multiple attributes, and most query parameters follow a range constraint, with a min and max value for each file attribute

Query Parameters

Parameter	Type	Description
Commit 1	Hash	First commit to be included within the range of the query
Commit 2	Hash	Final commit to be included within the range of the query
Subfolder	Path	Directory to be treated as query root (can be repo root)
Depth	Integer	Depth of results (search depth is infinite) relative to subfolder
Number of changes	Range	Number of commits a file has to be modified
Average % change	Range	Average % change of files considered to be modified
Average variance	Range	Variance of average % change
Variance	Range	Average of how evenly spread are the lines changed in each

Abridged Problem Statement

Given:

- K number of versions of tree T (T_0, T_1, T_2, … T_K)
- Each version T_x is given by a list of modified vertices from the previous
- An additional complete tree T_K
- Several attributes on each leaf vertex

Query:

- Find all nodes (to a depth of D relative to chosen node C) whose subtree between versions T_{first} and T_{final} ($0 \leq first \leq final \leq K$) contains any leaf vertices whose attributes satisfy all range query parameters

In abstract form, this is the problem we are trying to solve to design the algorithm and data structure for efficient construction and query. Any questions?

Constraints

- Versions (commits): $0 \leq K \leq 1\,000\,000$
- Vertices (files and folders): $0 \leq N \leq 1\,000\,000$

Naïve Solution (optimal creation)

Creation time:

- O(***KN***)

Query time:

- O(***KN***)

Space:

- O(***N***)

- Versions: $0 \leq K \leq 1\,000\,000$
- Vertices: $0 \leq N \leq 1\,000\,000$

We will divide each solution into two parts: a creation part where we construct a data structure that will be persisted and a query part that makes use of the data structure to answer queries efficiently. This is the complexity of a naïve solution that optimizes creation.

Naïve Solution (optimal creation)

Creation:

- Build complete tree T_{master} with all possible vertices (no other info)

Query:

- For each (partial) tree T_x within range ($T_{first} \leq T_x \leq T_{final}$), add all the nodes satisfying the query parameters to a set
- Traverse T_{master} bottom up and add all eligible non-leaf nodes to the set
- Traverse T_{master} top down from chosen node C to a depth of D and check whether each node is in the set

And this is a description of its algorithm

Naïve Solution (optimal query)

Creation time:

- $O(K^2N)$

Query time:

- $O(N)$

Space:

- $O(K^2N)$

- Versions: $0 \leq K \leq 1\,000\,000$
- Vertices: $0 \leq N \leq 1\,000\,000$

Naïve Solution (optimal query)

Creation:

- Build complete tree T_{master} with all possible vertices (no other info)
- Build all K^2 pairs of complete trees ($T_{0,0}$, … $T_{0,K}$, … $T_{K,0}$, … $T_{K,K}$) with all possible vertices and precompute all attributes with a sliding window

Query:

- Traverse T_{master} top down from chosen node C to a depth of D and check if the node satisfies the query parameters in the corresponding pair of tree versions T_{first} and T_{final}

My solution

Creation time:

- O(***KN***)

Query time:

- O(***NlogK***)

Space:

- O(***KN***)

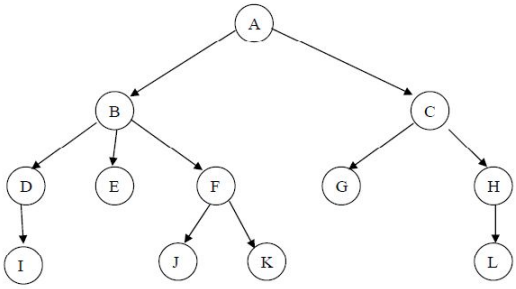
- Versions: $0 \leq K \leq 1\,000\,000$
- Vertices: $0 \leq N \leq 1\,000\,000$

Comparison

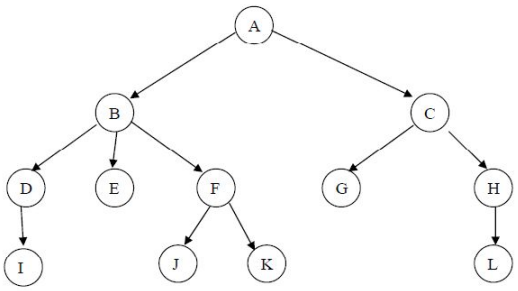
	Naïve solution (optimal creation)	Naïve solution (optimal query)	My solution
Creation time	O(***KN***)	O(***K²N***)	O(***KN***)
Query time	O(***KN***)	O(***N***)	O(***NlogK***)
Space	O(***N***)	O(***K²N***)	O(***KN***) sparse

- Versions: $0 \leq K \leq 1\,000\,000$
- Vertices: $0 \leq N \leq 1\,000\,000$

This is how the complexity of my algorithm compares to naïve solutions

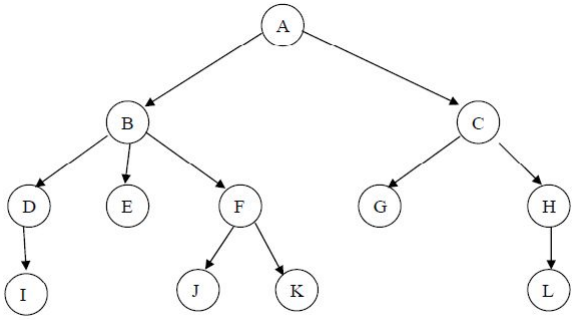


Let us consider a single version of a tree for now



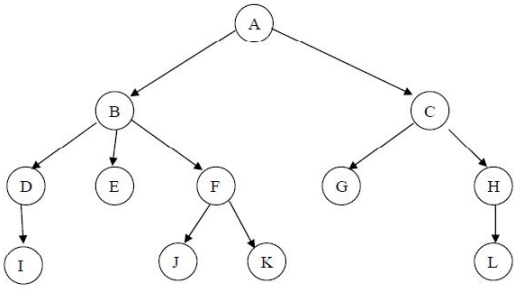
0	3	0	5	0	0
E	G	I	J	K	L

Leaf nodes have attributes each



A						
B						
C						
D						
F						
H						
	0	3	0	5	0	0
	E	G	I	J	K	L

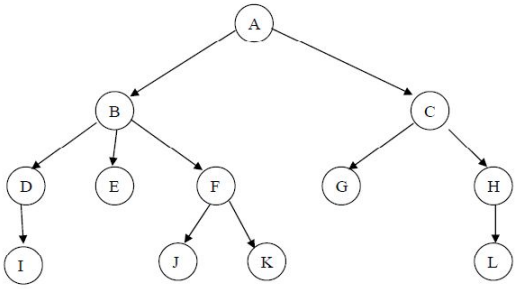
Euler Tour Technique



A						
B						
C						
D						
F						
H						
	0	3	0	5	0	0
	E	G	I	J	K	L
	0	1	2	3	4	5

Give each leaf node an index

Euler Tour Technique

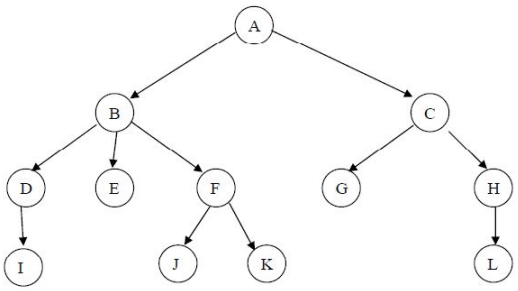


A						
B						
C						
D						
F						
H						
	0	3	0	5	0	0
	E	G	I	J	K	L
	0	1	2	3	4	5

	A	B	C	D	E	F	G	H	I	J	K	L
Start												
End												

Try to fill in this table on the start and end indexes of the range of each non-leaf node

Euler Tour Technique

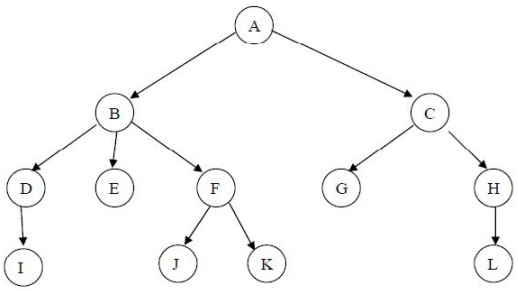


A												
B												
C												
D												
F												
H												
	0	3	0	5	0	0						
	E	G	I	J	K	L						
	0	1	2	3	4	5						

	A	B	C	D	E	F	G	H	I	J	K	L
Start	0			2	0	3	1	5	2	3	4	5
End	5			2	0	4	1	5	2	3	4	5

Cannot fill in for B and C

Euler Tour Technique

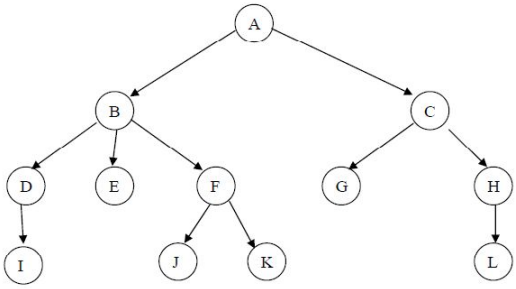


A						
B						
C						
D						
F						
H						
	0	3	0	5	0	0
	I	E	J	K	G	L
	0	1	2	3	4	5

	A	B	C	D	E	F	G	H	I	J	K	L
Start												
End												

The Euler Tour Technique is a variation of DFS. I will not go in depth about the Euler Tour Technique here due to time constraints.

Euler Tour Technique

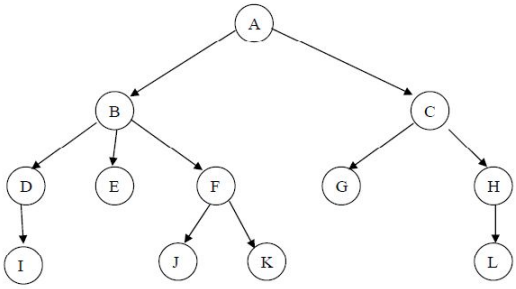


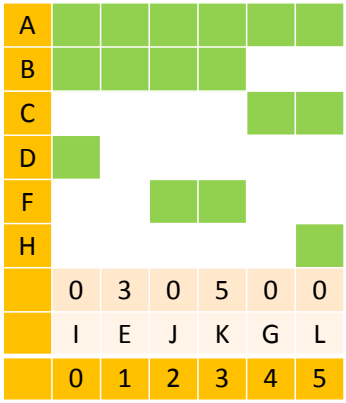
A												
B												
C												
D												
F												
H												
	0	3	0	5	0	0						
	I	E	J	K	G	L						
	0	1	2	3	4	5						

	A	B	C	D	E	F	G	H	I	J	K	L
Start												
End												

With our leaf nodes reordered, now we have contiguous ranges for our non-leaf nodes.

Euler Tour Technique

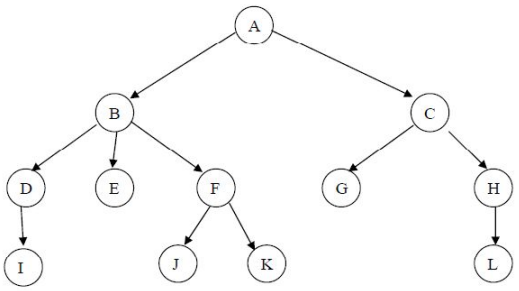




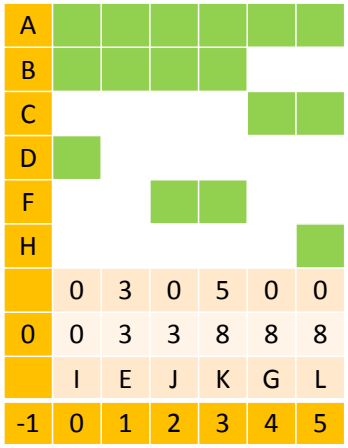
	A	B	C	D	E	F	G	H	I	J	K	L
Start	0	0	4	0	1	2	4	5	0	2	3	5
End	5	3	5	0	1	3	4	5	0	2	3	5

And we can fill in this table. The purpose of contiguous ranges…

Prefix Sum



	A	B	C	D	E	F	G	H	I	J	K	L
Start	0	0	4	0	1	2	4	5	0	2	3	5
End	5	3	5	0	1	3	4	5	0	2	3	5



is so that we can do a prefix sum! Each index in a prefix sum array, contains the sum of all of the elements up to that index in the original array.

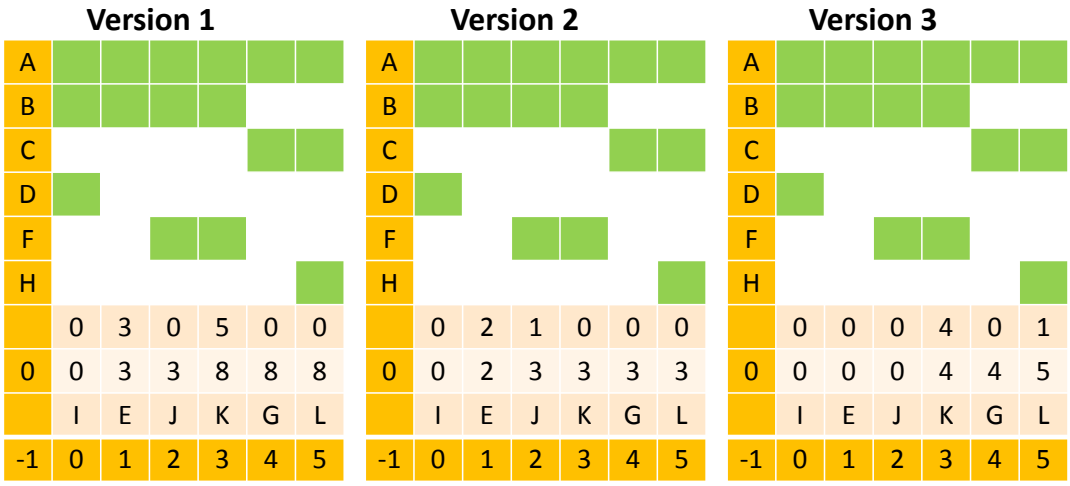
0 + 3 = 3. 0 + 3 + 0 + 5 = 8.

Now we can query for the sum of non-leaf nodes in constant time, by taking the element at their end index subtracted by the element 1 less than their start index.

(Example on F)

See the table on the left?

Versioning



Now we have multiple versions of it. So how do we deal with multiple versions?

Versioning

Version 1

0	0	3	3	8	8	8
	I	E	J	K	G	L
-1	0	1	2	3	4	5

Version 2

0	0	2	3	3	3	3
	I	E	J	K	G	L
-1	0	1	2	3	4	5

Version 3

0	0	0	0	4	4	5
	I	E	J	K	G	L
-1	0	1	2	3	4	5

Let’s take the bottom part of these tables

Versioning

Version 1	0	0	3	3	8	8	8
Version 2	0	0	2	3	3	3	3
Version 3	0	0	0	0	4	4	5
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

And stack them together

2D Prefix Sum

1D Prefix Sum

Version 1	0	0	3	3	8	8	8
Version 2	0	0	2	3	3	3	3
Version 3	0	0	0	0	4	4	5
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

2D Prefix Sum

Version 1	0	0	3	3	8	8	8
Version 2	0	0	5	6	11	11	11
Version 3	0	0	5	6	15	15	16
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

And now we can do a 2D prefix sum on this table

2D Prefix Sum

1D Prefix Sum

Version 1	0	0	3	3	8	8	8
Version 2	0	0	2	3	3	3	3
Version 3	0	0	0	0	4	4	5
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

2D Prefix Sum

Version 1	0	0	3	3	8	8	8
Version 2	0	0	5	6	11	11	11
Version 3	0	0	5	6	15	15	16
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

To query we can select a first and final version,
in this case versions 2 and 3
And we select rows, 2-1=1, row 1 and row 3

2D Prefix Sum

1D Prefix Sum

Version 1	0	0	3	3	8	8	8
Version 2	0	0	2	3	3	3	3
Version 3	0	0	0	0	4	4	5
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

2D Prefix Sum

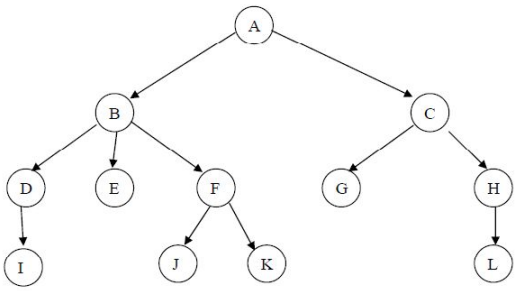
Version 1	0	0	3	3	8	8	8
Version 2	0	0	5	6	11	11	11
Version 3	0	0	5	6	15	15	16
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

Versions 2 to 3

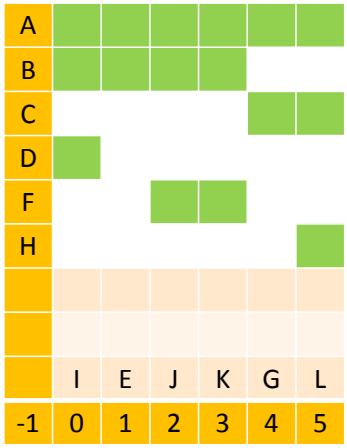
0	0	2	3	7	7	8

And reduce it to a 1D array by taking the difference. Now we take this 1D array,

Prefix Sum

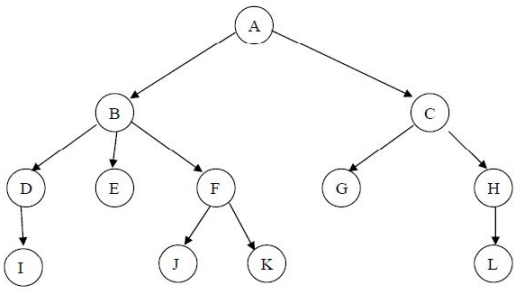


	A	B	C	D	E	F	G	H	I	J	K	L
Start	0	0	4	0	1	2	4	5	0	2	3	5
End	5	3	5	0	1	3	4	5	0	2	3	5



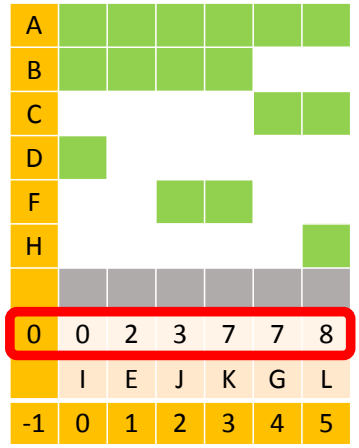
Go back to the original table

Prefix Sum



	A	B	C	D	E	F	G	H	I	J	K	L
Start	0	0	4	0	1	2	4	5	0	2	3	5
End	5	3	5	0	1	3	4	5	0	2	3	5

Versions 2 to 3



And put it in. And now we can carry out 1D prefix sum operations on it, except that this now applies for a range of versions instead of a single version.

2D Prefix Sum

1D Prefix Sum

K VERSIONS

Version 1	0	0	3	3	8	8	8
Version 2	0	0	2	3	3	3	3
Version 3	0	0	0	0	4	4	5
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

2D Prefix Sum

Version 1	0	0	3	3	8	8	8
Version 2	0	0	5	6	11	11	11
Version 3	0	0	5	6	15	15	16
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

N VERTICES

Now you might have noticed that our 2D prefix sum array is very big

2D Prefix Sum

- Versions: $0 \leq K \leq 1\,000\,000$
- Vertices: $0 \leq N \leq 1\,000\,000$

1D Prefix Sum

Version 1	0	0	3	3	8	8	8
Version 2	0	0	2	3	3	3	3
Version 3	0	0	0	0	4	4	5
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

K VERSIONS

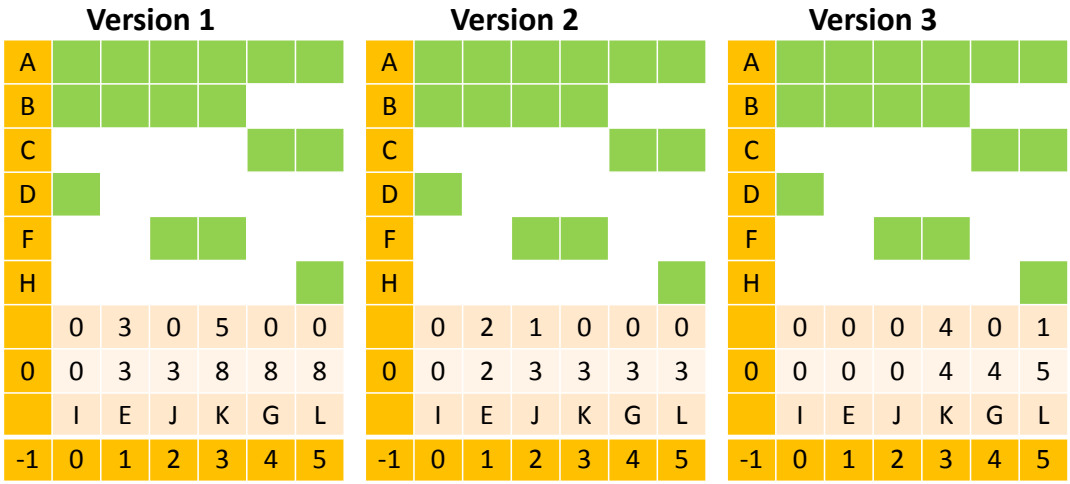
2D Prefix Sum

Version 1	0	0	3	3	8	8	8
Version 2	0	0	5	6	11	11	11
Version 3	0	0	5	6	15	15	16
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

N VERTICES

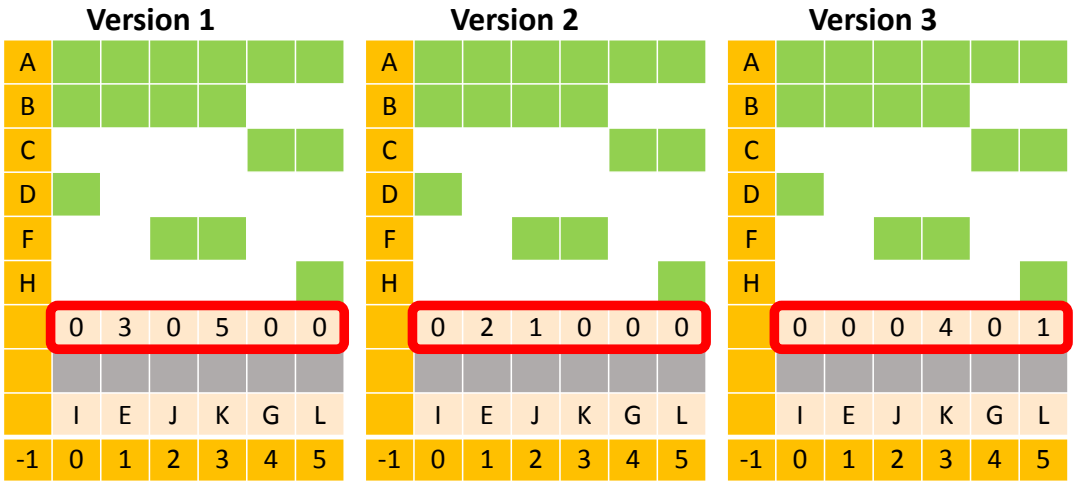
Unrealistically big. Additionally we notice lots of repeated numbers. This means we can save space by making it sparse!

Versioning



Back to our original diagram. We discard the 1D prefix sum

Versioning



And drop all the zeroes

Versioning

Version 1

0	3	0	5	0	0
	3		5		
I	E	J	K	G	L
0	1	2	3	4	5

Version 2

0	2	1	0	0	0
	2	1			
I	E	J	K	G	L
0	1	2	3	4	5

Version 3

0	0	0	4	0	1
			4		1
I	E	J	K	G	L
0	1	2	3	4	5

Keeping track of only the actual values

Sparse Prefix Sum

Version 1		3		5		
Version 2		2	1			
Version 3				4		1
	I	E	J	K	G	L
	0	1	2	3	4	5

Again, we stack the versions together

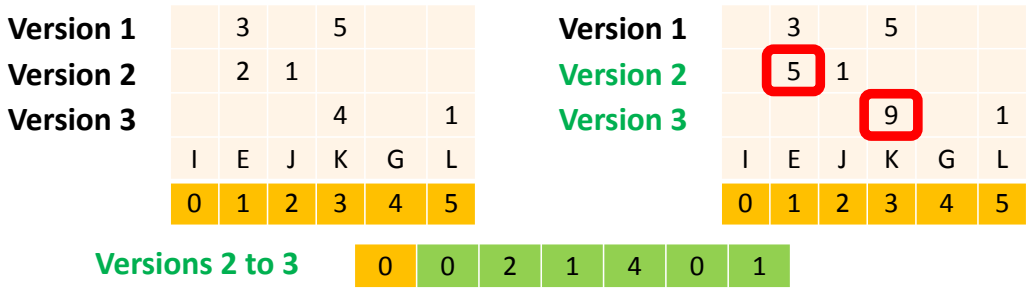
Sparse Prefix Sum

Version 1		3		5		
Version 2		2	1			
Version 3				4		1
	I	E	J	K	G	L
	0	1	2	3	4	5

Version 1		3		5		
Version 2		5	1			
Version 3				9		1
	I	E	J	K	G	L
	0	1	2	3	4	5

And do a prefix sum, but this time in the vertical axis instead of the horizontal

Sparse Prefix Sum



To query we have our first and final versions as per normal, and take the difference of the first and final versions and obtain a 1D array. For example …

Sparse Prefix Sum

Version 1		3		5		
Version 2		2	1			
Version 3				4		1
	I	E	J	K	G	L
	0	1	2	3	4	5

Version 1		3		5		
Version 2		5	1			
Version 3				9		1
	I	E	J	K	G	L
	0	1	2	3	4	5

Versions 2 to 3	0	0	2	1	4	0	1
	0	0	2	3	7	7	8

And then we apply a 1D prefix sum on that array

2D Prefix Sum

1D Prefix Sum

Version 1	0	0	3	3	8	8	8
Version 2	0	0	2	3	3	3	3
Version 3	0	0	0	0	4	4	5
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

2D Prefix Sum

Version 1	0	0	3	3	8	8	8
Version 2	0	0	5	6	11	11	11
Version 3	0	0	5	6	15	15	16
		I	E	J	K	G	L
	-1	0	1	2	3	4	5

Versions 2 to 3

0	0	2	3	7	7	8

Notice that we got back the same results as the 2D prefix sum

Sparse Prefix Sum

Version 1		3		5		
Version 2		2	1			
Version 3				4		1
	I	E	J	K	G	L
	0	1	2	3	4	5

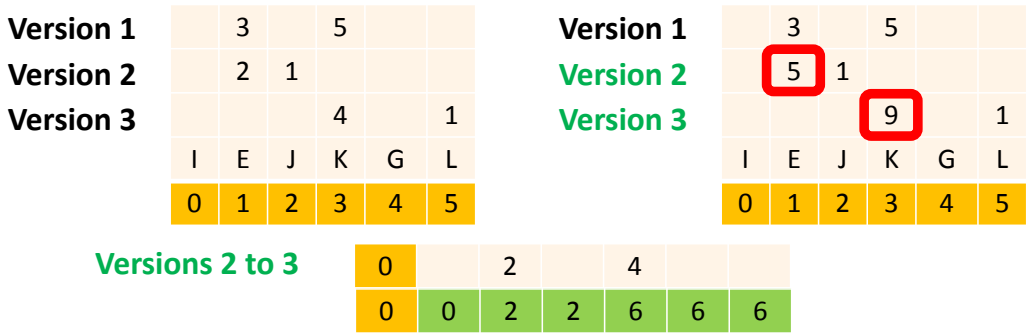
Version 1		3		5		
Version 2		5	1			
Version 3				9		1
	I	E	J	K	G	L
	0	1	2	3	4	5

Versions 2 to 3	0	0	2	1	4	0	1
	0	0	2	3	7	7	8

But we're now storing way fewer values.

Additionally we can perform additional operations on the 1D array before calculating the 1D prefix sum, such as removing all elements less than 2

Sparse Prefix Sum



Which allows us to specify custom range parameters on each leaf node attribute

Query process

1.	Sparse 2D matrix Dense 1D array (based on specified commit range)	O(NlogK)
2.	Filter each leaf node (file) by query parameters	O(N)
3.	Prefix sum on dense 1D array	O(N)
4.	Traverse tree and calculate attributes for non-leaf nodes (folders) to filter into the results set	O(N) × O(1)

To recap this is the query process of my algorithm

Creation process

1. Create a sparse 2D matrix for each attribute — O(KN)
2. Build complete tree $\boldsymbol{T}_{master}$ with all possible vertices — O(N)
3. Traverse $\boldsymbol{T}_{master}$ with Euler tour technique to get start and end ranges of all non-leaf nodes — O(N)

And this is the creation process

Implementation/API

- Python
- Sparse 2D matrix Dictionary of lists
- Tree Nested dictionaries
- *RepoStats* class
  - Creates the data structures for a given repo in initialization
  - Perform queries using class methods
  - Save and load *RepoStats* class instances using *pickle* (class methods can be updated dynamically and re-saved)
- Queries return a dot dictionary of outputs

These are the implementation details. I've provided a RepoStats class as an API to make use of the algorithm. It does the entire process from processing the local repository to answering queries.

Outputs

Output	Type	Description
Results	List [path]	List of paths based on query params
Files	Integer	Total number of files
Files modified	Integer	Number of files considered modified based on query params
Change count	Integer	Aggregate number of modifications across all "modified" files
Total % change	Float	Average % change across all "modified" files (unweighted)
Total % change (weighted)	Float	Average % change across all "modified" files (weighted by length of file)
Average	Dictionary {path: float}	% each file has changed (average)
Average variance	Dictionary {path: float}	% each file changed (variance)

These are the outputs of each query

Bonus Queries

Query	Query parameters	Outputs	Description
Modifications	First commit	Dictionary {hash: integer}	Number of modifications made by each commit to a path or within its subpaths
	Final commit		
	Path		
Existence	First commit	Dictionary {path: existence}	Whether each file existed within the commit range, came into existence, ceased to exist, or never existed
	Final commit		
	Path		

As a bonus, I have also implemented these 2 queries using my algorithm. For existence queries the output is out of all files that ever existed in the commit history.

Any Questions?