Flavors
Library of AI Powered Trie Structures for Fast Parallel Lookup

Session ID S8401

Albert Wolant
PwC, Warsaw University of Technology
Krzysztof Kaczmarski, PhD.
Warsaw University of Technology

GTC Silicon Valley, 27 March 2018

What we will talk about?

- What is Flavors and how it came to be?
- Overview of algorithms
- Experimental results and benchmarks
- Customization using machine learning

From where it came?

Session on GTC 2016:

Presentation	Media
Fast Detection of Neighboring Vectors **Krzysztof Kaczmarski (Warsaw University of Technology, Faculty of Mathematics and Information Science)** We'll present several methods for detecting pairs of vectors, which are in Hamming distance 1. This problem is an important part of the cell graph construction in motion planning in a space with obstacles. We'll begin with a naive square-time s ... Read More **Keywords:** Algorithms, Tools and Libraries, Robotics & Autonomous Machines, GTC 2016 - ID S6402	**Streaming:** > Watch Now

Session on GTC Europe 2017:

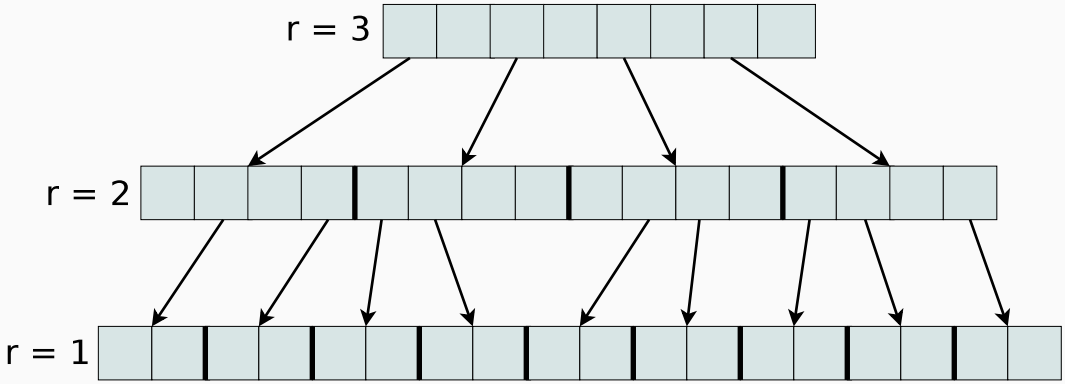
Presentation	Media
Flavors: Library for Fast Parallel Lookup Using Custom Radix Trees **Albert Wolant (WARSAW UNIVERSITY OF THECHNOLOGY)** Learn how to use configurable radix trees to perform fast parallel lookup operations on GPU. In this session you will find out how to use our library to create radix tree specifically tailored to your needs. You will also see examples on how to ... Read More **Keywords:** Algorithms, Tools and Libraries, GTC Europe 2017 - ID 23269	**Download:** > MP4 > PDF

What it can do?

- Flavors provides algorithm to build and search radix-tree with configurable bit stride on the GPU.
- Values can be of constant length or can vary in length.
- Search can be done to find values exactly or perform longest-prefix matching.

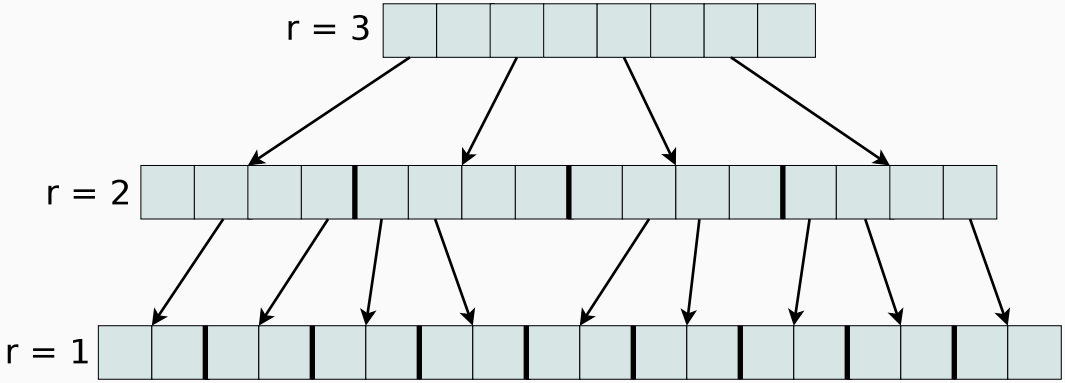
Tree for constant key length

Example of tree for bit strides sequence $\{3, 2, 1\}$.



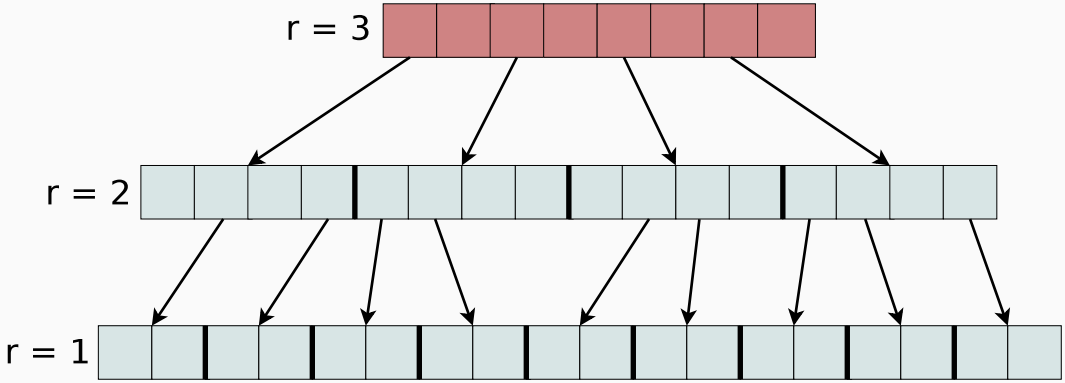
Tree for constant key length - searching example

Example search for key 010 — 01 — 1



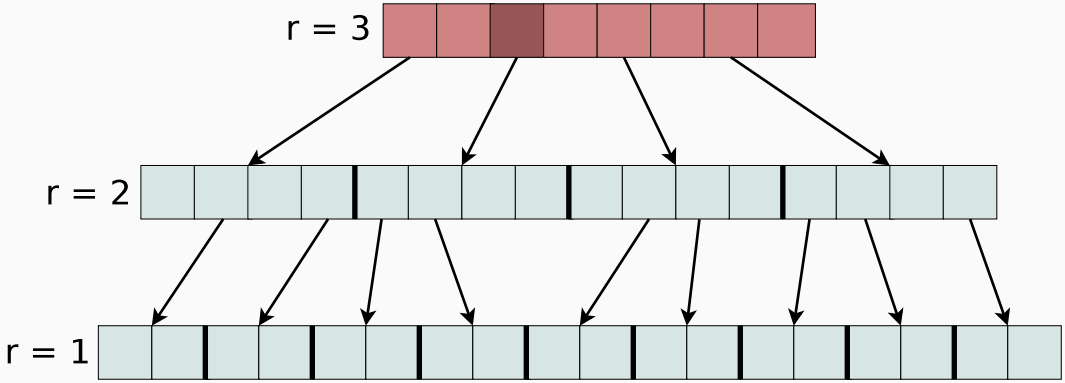
Tree for constant key length - searching example

Example search for key 010 — 01 — 1



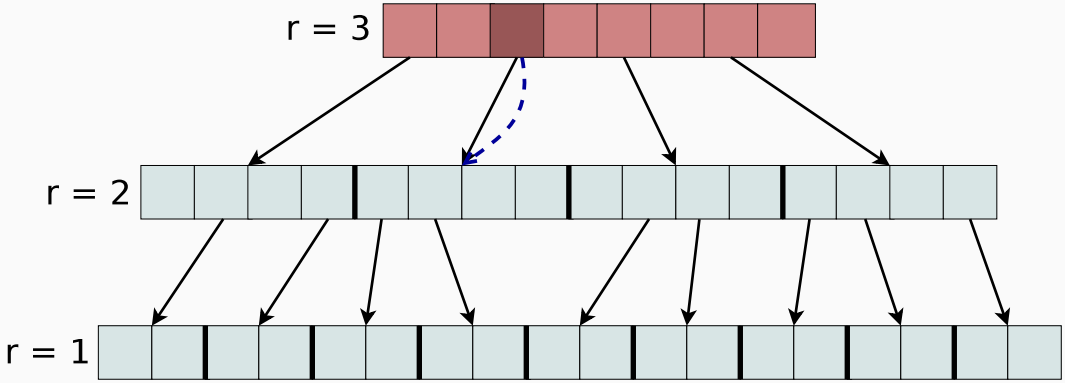
Tree for constant key length - searching example

Example search for key 010 — 01 — 1



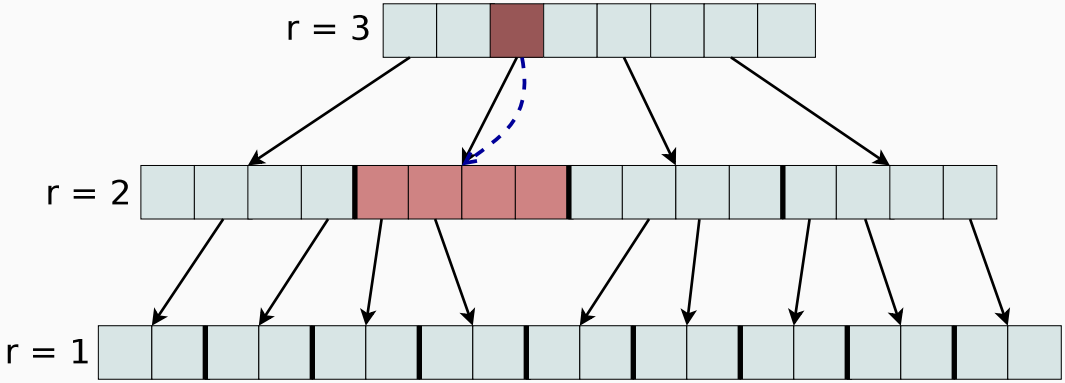
Tree for constant key length - searching example

Example search for key 010 — 01 — 1



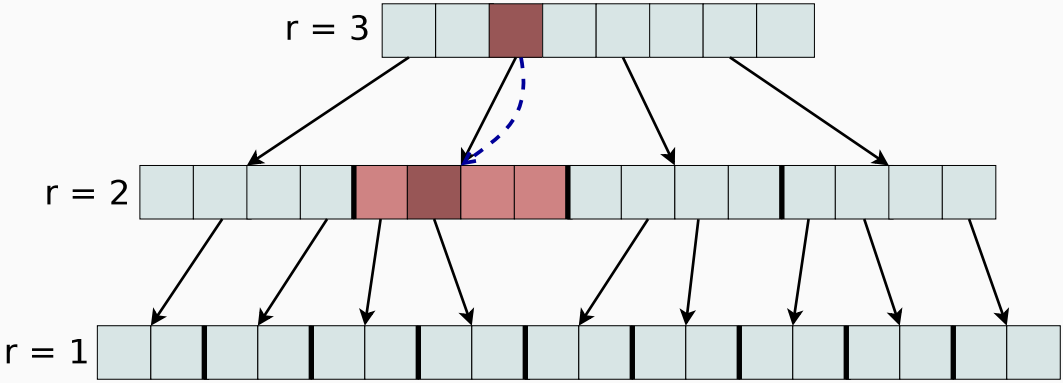
Tree for constant key length - searching example

Example search for key 010 — 01 — 1



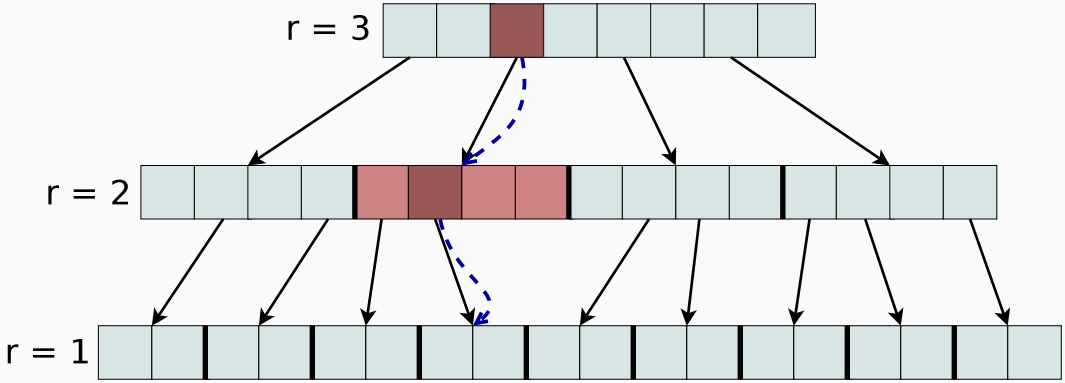
Tree for constant key length - searching example

Example search for key 010 — 01 — 1



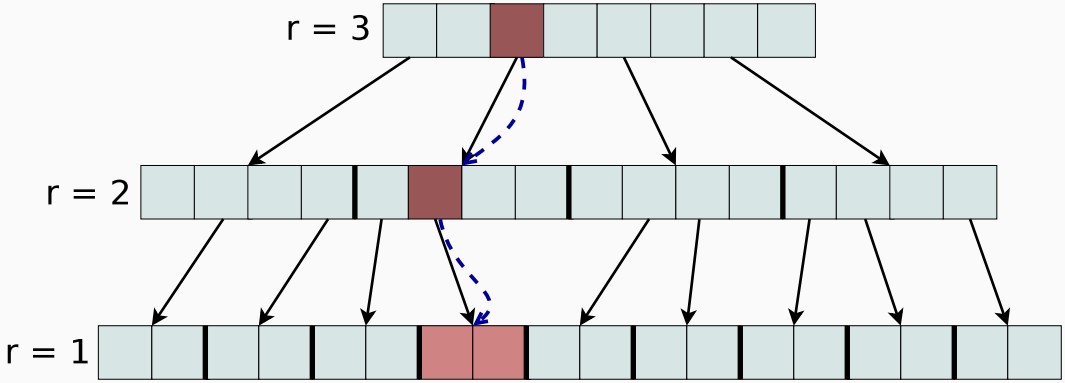
Tree for constant key length - searching example

Example search for key 010 — 01 — 1



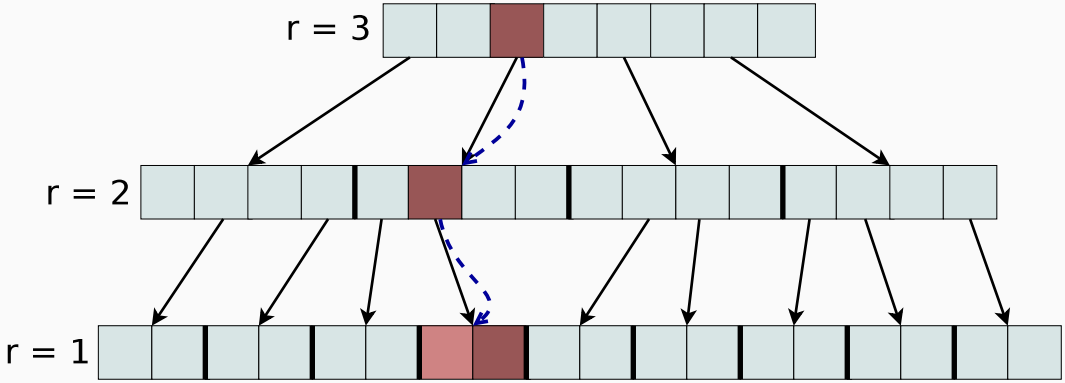
Tree for constant key length - searching example

Example search for key 010 — 01 — 1



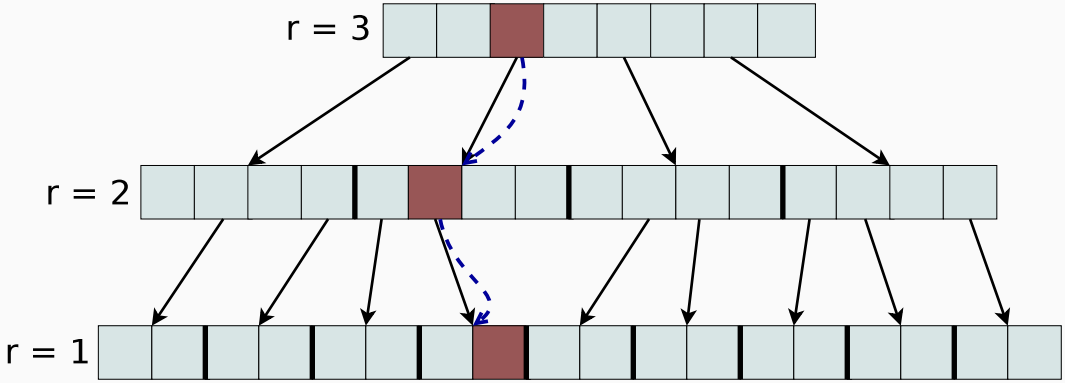
Tree for constant key length - searching example

Example search for key 010 — 01 — 1



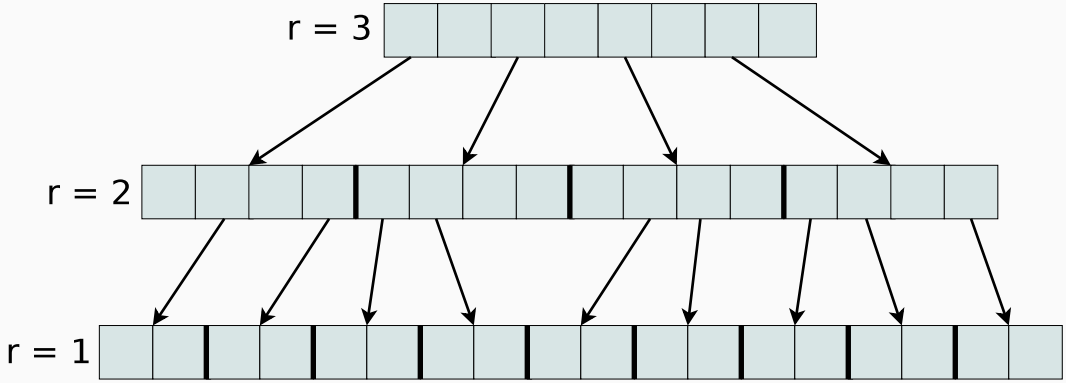
Tree for constant key length - searching example

Example search for key 010 — 01 — 1



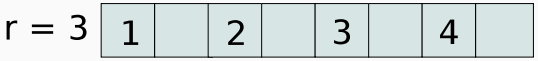
Tree for constant key length - node structure

In practice, cells hold indexes of nodes on next level instead of pointers. Last level keeps original indexes of keys.



Tree for constant key length - node structure

In practice, cells hold indexes of nodes on next level instead of pointers. Last level keeps original indexes of keys.



Tree construction - input data

L_1	L_2	L_3	
010	01	1	8
000	11	0	2
010	00	1	3
100	10	0	1
110	00	0	10
110	01	0	4
110	11	1	9
000	01	0	5
100	10	1	6
100	01	1	7

Tree construction - data sorting

L_1	L_2	L_3	P
000	01	0	8
000	11	0	2
010	00	1	3
010	01	1	1
100	01	1	10
100	10	0	4
100	10	1	9
110	00	0	5
110	01	0	6
110	11	1	7

Tree construction - values

L_1	L_2	L_3	P
0	1	0	8
0	3	0	2
2	0	1	3
2	1	1	1
4	1	1	10
4	2	0	4
4	2	1	9
5	0	0	5
5	1	0	6
5	3	1	7

Tree construction - nodes borders

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

values

L_1	L_2	L_3
1		
0		
0		
0		
0		
0		
0		
0		
0		
0		

nodes borders

Tree construction - nodes borders

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

values

L_1	L_2	L_3
1	1	
0	0	
0	0	
0	0	
0	0	
0	0	
0	0	
0	0	
0	0	
0	0	

nodes borders

Tree construction - nodes borders

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

values

L_1	L_2	L_3
1	1	
0	0	
0	1	
0	0	
0	1	
0	0	
0	0	
0	1	
0	0	
0	0	

nodes borders

Tree construction - nodes borders

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

values

L_1	L_2	L_3
1	1	1
0	0	0
0	1	1
0	0	0
0	1	1
0	0	0
0	0	0
0	1	1
0	0	0
0	0	0

nodes borders

Tree construction - nodes borders

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

Tree construction - nodes indexes

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes indexes

Tree construction - nodes indexes

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	2	3
1	2	4
1	3	5
1	3	6
1	3	6
1	4	7
1	4	8
1	4	9

nodes indexes

Tree construction - nodes indexes

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	2	3
1	2	4
1	3	5
1	3	6
1	3	6
1	4	7
1	4	8
1	4	9

nodes indexes

Tree construction - nodes allocation

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

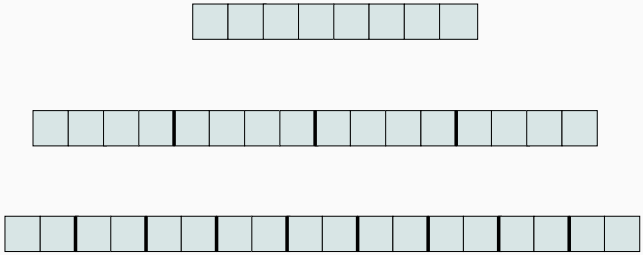
values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	2	3
1	2	4
1	3	5
1	3	6
1	3	6
1	4	7
1	4	8
1	4	9

nodes indexes



Tree construction - nodes allocation

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

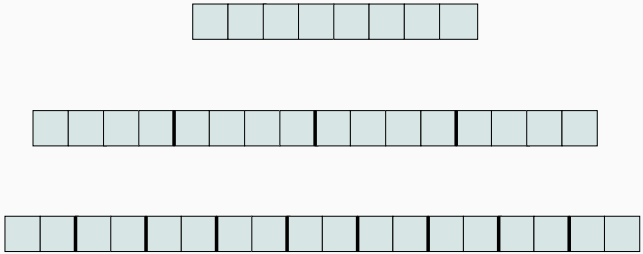
values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	2	3
1	2	4
1	3	5
1	3	6
1	3	6
1	4	7
1	4	8
1	4	9

nodes indexes



Last row of *nodesIndexes* array has node counts for each level. Since size of node on level is known (based on bit stride), memory for all nodes can be allocated.

Tree construction - nodes allocation

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

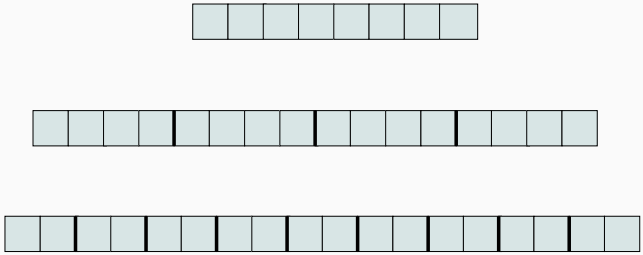
values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	2	3
1	2	4
1	3	5
1	3	6
1	3	6
1	4	7
1	4	8
1	4	9

nodes indexes



Last row of *nodesIndexes* array has node counts for each level. Since size of node on level is known (based on bit stride), memory for all nodes can be allocated.

Values in arrays above can also be used to link nodes between levels.

Tree construction - linking nodes

Let *V* be *values* array, *B* be *nodes borders*, and *N* be *nodes indexes*. Let's consider one cell of this arrays, in row *'key'* and column *'level'*.

Tree construction - linking nodes

Let V be *values* array, B be *nodes borders*, and N be *nodes indexes*. Let's consider one cell of this arrays, in row $'key'$ and column $'level'$.

Let $v = V[key][level]$ and $n = N[key][level]$ and $b = B[key][level]$.

Tree construction - linking nodes

Let V be *values* array, B be *nodes borders*, and N be *nodes indexes*. Let's consider one cell of this arrays, in row $'key'$ and column $'level'$.

Let $v = V[key][level]$ and $n = N[key][level]$ and $b = B[key][level]$.

Let C be 2D array of pointers to all of the nodes (for example $C[1][2]$ points to the beginning of second node on first level, since indexing is done from 1).

Tree construction - linking nodes

Let V be *values* array, B be *nodes borders*, and N be *nodes indexes*. Let's consider one cell of this arrays, in row $'key'$ and column $'level'$.

Let $v = V[key][level]$ and $n = N[key][level]$ and $b = B[key][level]$.

Let C be 2D array of pointers to all of the nodes (for example $C[1][2]$ points to the beginning of second node on first level, since indexing is done from 1).

Then:

$$C[level][n][v] \longleftarrow N[key][level + 1]$$

Tree construction - linking nodes

Let V be *values* array, B be *nodes borders*, and N be *nodes indexes*. Let's consider one cell of this arrays, in row $'key'$ and column $'level'$.

Let $v = V[key][level]$ and $n = N[key][level]$ and $b = B[key][level]$.

Let C be 2D array of pointers to all of the nodes (for example $C[1][2]$ points to the beginning of second node on first level, since indexing is done from 1).

Then:

$$C[level][n][v] \longleftarrow N[key][level + 1]$$

To avoid multiple writes to the same cell, above is done only, if b is equal to 1.

Tree construction - linking nodes

On last level, we do:

$$C[level][n] + v \longleftarrow P[key]$$

where P is permutation containing original indexes of keys.

Tree construction - linking nodes

On last level, we do:

$$C[level][n] + v \longleftarrow P[key]$$

where P is permutation containing original indexes of keys.

This operation is done for every key.

Tree construction - linking nodes example

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

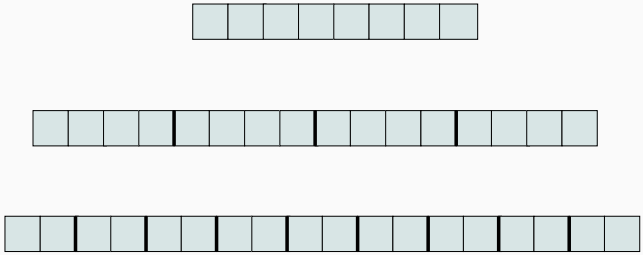
values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	2	3
1	2	4
1	3	5
1	3	6
1	3	6
1	4	7
1	4	8
1	4	9

nodes indexes



Tree construction - linking nodes example

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

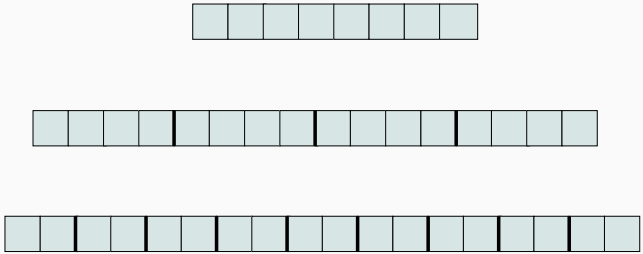
values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	2	3
1	2	4
1	3	5
1	3	6
1	3	6
1	4	7
1	4	8
1	4	9

nodes indexes



$level = 2$, $key = 8$,

Tree construction - linking nodes example

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

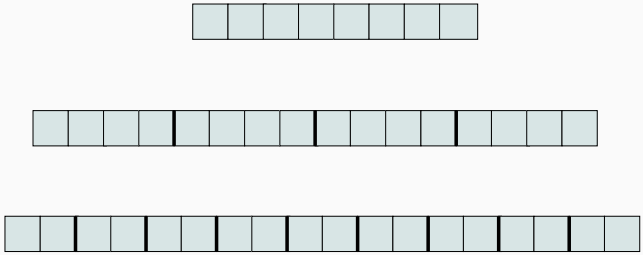
values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	2	3
1	2	4
1	3	5
1	3	6
1	3	6
1	4	7
1	4	8
1	4	9

nodes indexes



$level = 2, key = 8, v = 0, n = 4, b = 1$

Tree construction - linking nodes example

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

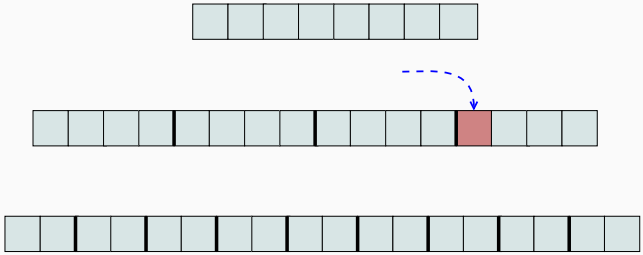
values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	2	3
1	2	4
1	3	5
1	3	6
1	3	6
1	4	7
1	4	8
1	4	9

nodes indexes



$level = 2, key = 8, v = 0, n = 4, b = 1$

$C[level][n] + v = C[2][4]$

Tree construction - linking nodes example

L_1	L_2	L_3
0	1	0
0	3	0
2	0	1
2	1	1
4	1	1
4	2	0
4	2	1
5	0	0
5	1	0
5	3	1

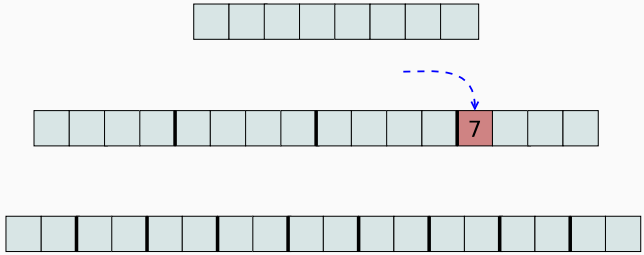
values

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	2	3
1	2	4
1	3	5
1	3	6
1	3	6
1	4	7
1	4	8
1	4	9

nodes indexes



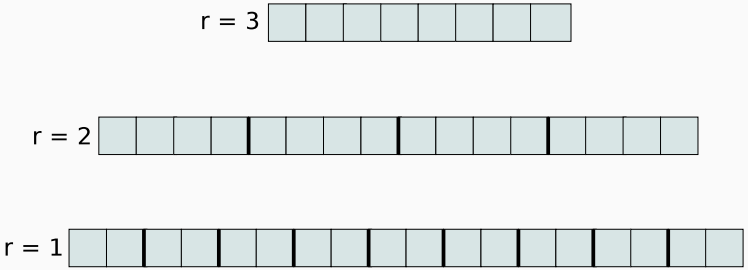
$level = 2$, $key = 8$, $v = 0$, $n = 4$, $b = 1$

$C[level][n] + v = C[2][4]$

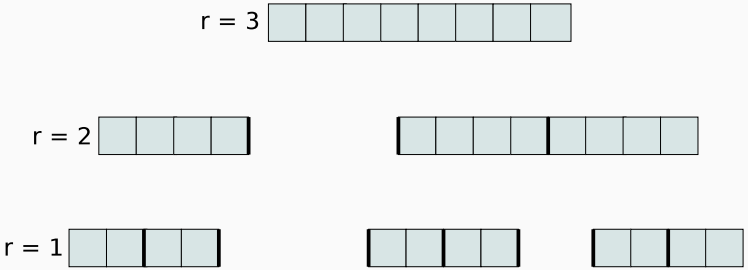
Tree construction - what about varying lengths?

L_1	L_2	L_3	
010	01	1	8
000	11	X	2
010	00	1	3
100	1X	X	1
110	XX	X	10
110	01	0	4
110	11	1	9
000	01	0	5
100	10	1	6
10X	XX	X	7

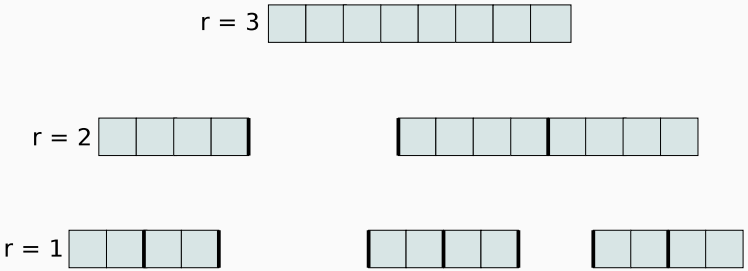
Tree construction - what about varying lengths?



Tree construction - removing empty nodes

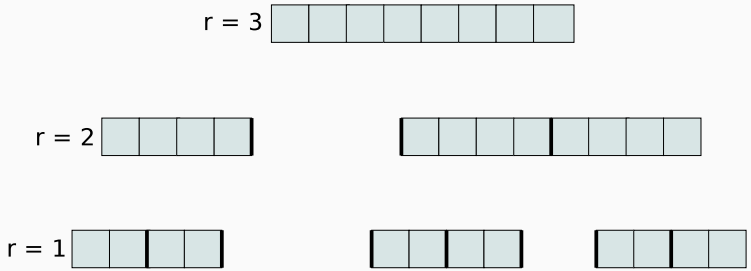


Tree construction - removing empty nodes



Some of the nodes are no longer needed, because masks that would occupy them were shorter.

Tree construction - removing empty nodes



Some of the nodes are no longer needed, because masks that would occupy them were shorter.

How to allocate memory and link nodes together?

Tree construction - removing empty nodes

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	2	3
1	2	4
1	3	5
1	3	6
1	3	6
1	4	7
1	4	8
1	4	9

nodes indexes

Tree construction - removing empty nodes

L_1	L_2	L_3
1	1	1
0	0	1
0	1	1
0	0	1
0	1	1
0	0	1
0	0	0
0	1	1
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	0	0
1	0	0
1	3	5
1	3	0
1	3	6
1	0	0
1	4	8
1	4	9

nodes indexes

After calculating *nodesIndexes* array, cells are cleared (values set to 0), if mask does not reach level.

Tree construction - removing empty nodes

L_1	L_2	L_3
1	1	1
0	0	1
0	0	0
0	0	0
0	1	1
0	0	1
0	0	0
0	1	0
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	0	0
1	0	0
1	3	5
1	3	0
1	3	6
1	0	0
1	4	8
1	4	9

nodes indexes

Then ’1’ in *nodesBorders* representing no longer needed nodes are cleared.

Tree construction - removing empty nodes

L_1	L_2	L_3
1	1	1
0	0	1
0	0	0
0	0	0
0	1	1
0	0	1
0	0	0
0	1	0
0	0	1
0	0	1

nodes borders

L_1	L_2	L_3
1	1	1
1	1	2
1	1	2
1	1	2
1	2	3
1	2	4
1	2	4
1	3	4
1	3	5
1	3	6

nodes indexes

After that, *nodesIndexes* can be recalculated and nodes allocated and linked exactly as before.

Tree construction - containers

For bit stride $\{3, 2, 1\}$, masks:

$010 - 0X - X$
$010 - 00 - X$

land in the same place in the tree.

Tree construction - containers

For bit stride {3, 2, 1}, masks:

$010 - 0X - X$
$010 - 00 - X$

land in the same place in the tree.

Solution is attaching containers for masks to each node.

Tree construction - containers

For bit stride {3, 2, 1}, masks:

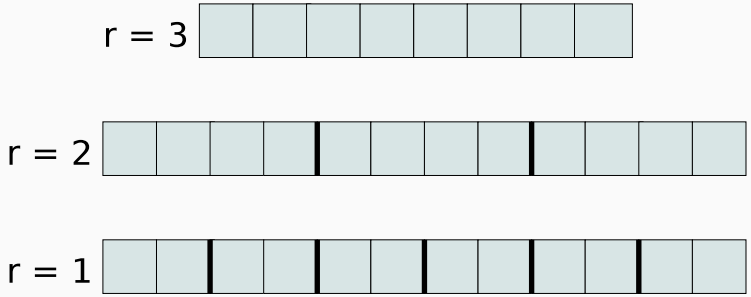
$010 - 0X - X$
$010 - 00 - X$

land in the same place in the tree.

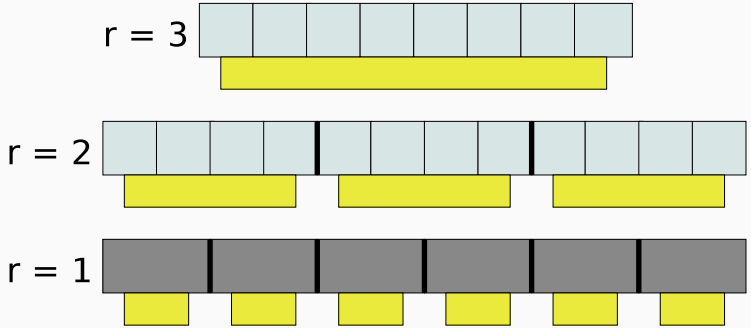
Solution is attaching containers for masks to each node.

Since masks are kept in this containers, last tree level, holding original indexes, is no longer needed. On this level we only need containers.

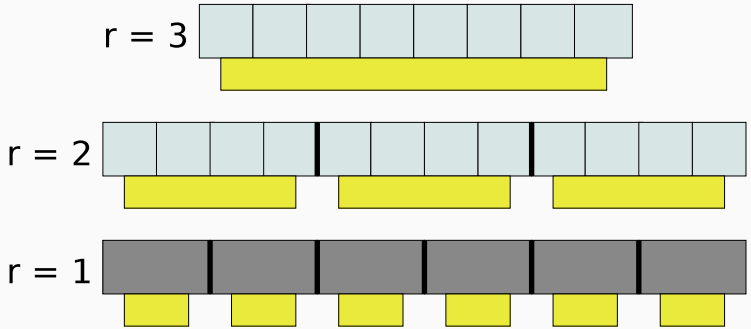
Tree construction - containers



Tree construction - containers



Tree construction - containers



Current implementation uses simple lists, kept in single array and each of them is sorted by masks length.

Tree construction - building lists

Lists are build in few steps:

Tree construction - building lists

Lists are build in few steps:

1. Masks are marked with index of node to which they belong (*nodesIndexes* on mask level, 0 otherwise).

Tree construction - building lists

Lists are build in few steps:

1. Masks are marked with index of node to which they belong (*nodesIndexes* on mask level, 0 otherwise).
2. Lists lengths are calculated, using reduce_by_key operation.

Tree construction - building lists

Lists are build in few steps:

1. Masks are marked with index of node to which they belong (*nodesIndexes* on mask level, 0 otherwise).
2. Lists lengths are calculated, using reduce_by_key operation.
3. All masks belong to some list, so memory for all lists is allocated. Only list start and list length is kept with the node (in yellow rectangle).

Tree construction - building lists

Lists are build in few steps:

1. Masks are marked with index of node to which they belong (*nodesIndexes* on mask level, 0 otherwise).
2. Lists lengths are calculated, using reduce_by_key operation.
3. All masks belong to some list, so memory for all lists is allocated. Only list start and list length is kept with the node (in yellow rectangle).
4. Lists starts are calculated by performing exclusive_scan operation.

Tree construction - building lists

Lists are build in few steps:

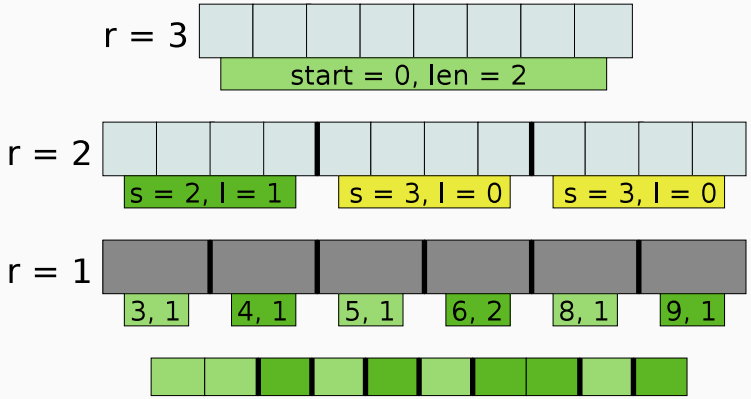
1. Masks are marked with index of node to which they belong (*nodesIndexes* on mask level, 0 otherwise).
2. Lists lengths are calculated, using reduce_by_key operation.
3. All masks belong to some list, so memory for all lists is allocated. Only list start and list length is kept with the node (in yellow rectangle).
4. Lists starts are calculated by performing exclusive_scan operation.
5. For every mask, special code is generated, based on their level, node and length.

Tree construction - building lists

Lists are build in few steps:

1. Masks are marked with index of node to which they belong (*nodesIndexes* on mask level, 0 otherwise).
2. Lists lengths are calculated, using reduce_by_key operation.
3. All masks belong to some list, so memory for all lists is allocated. Only list start and list length is kept with the node (in yellow rectangle).
4. Lists starts are calculated by performing exclusive_scan operation.
5. For every mask, special code is generated, based on their level, node and length.
6. Masks original indexes are sorted by this codes. This ensures them being in the right spot on the right list.

Tree construction - containers



Experimental results - test platform

All presented tests were performed on :

- GTX 1080
- Intel(R) i7 4790K
- CUDA 9
- Ubuntu 16.04

Experimental results - random keys build throughput

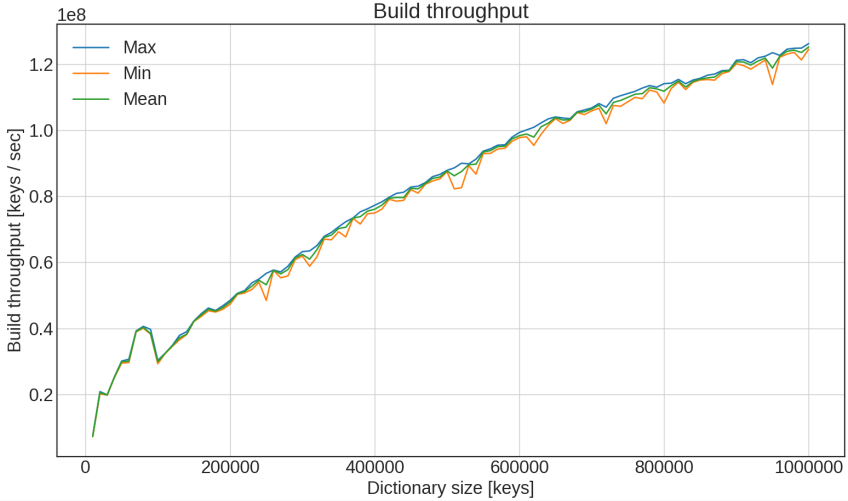


Figure 1: Build throughput of random keys.

Experimental results - random keys find throughput

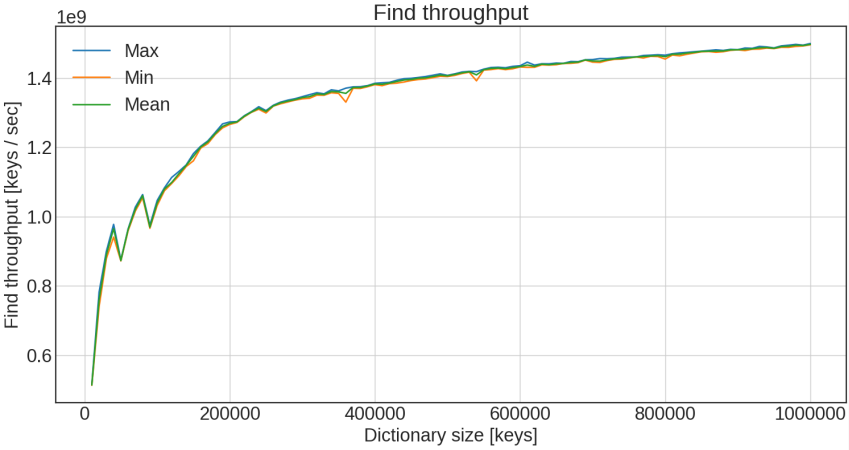


Figure 2: Find throughput of random keys.

Experimental results - random keys find latency

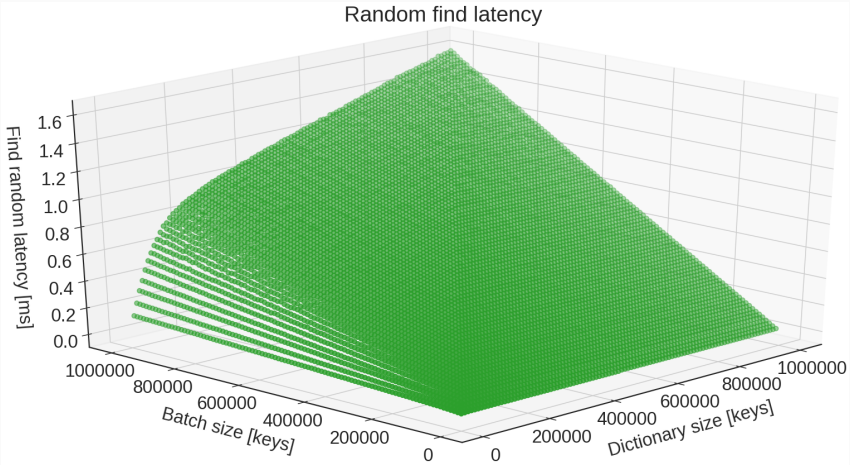


Figure 3: Find latency for finding random keys.

Experimental results - random keys build throughput for different lengths of keys

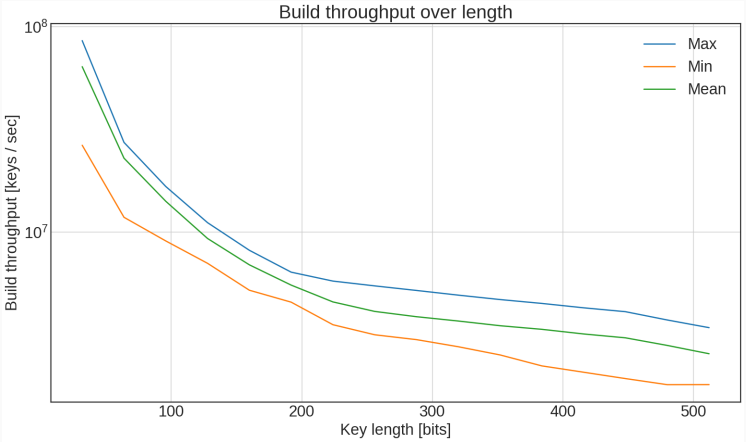


Figure 4: Build throughput of random keys for different lengths of keys.

Experimental results - random keys find throughput for different lengths of keys

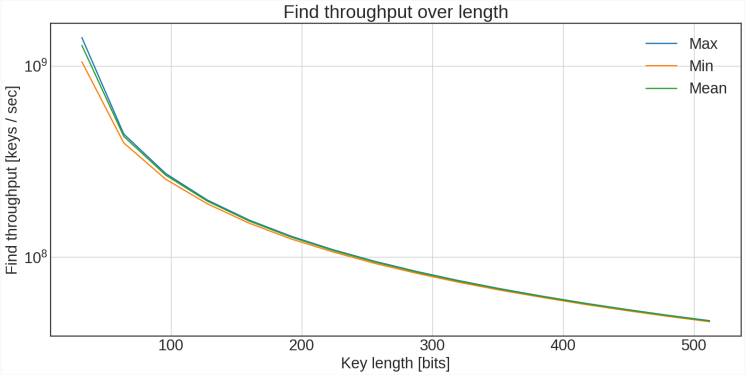


Figure 5: Find throughput of random keys for different lengths of keys.

Experimental results - tree match IP throughput

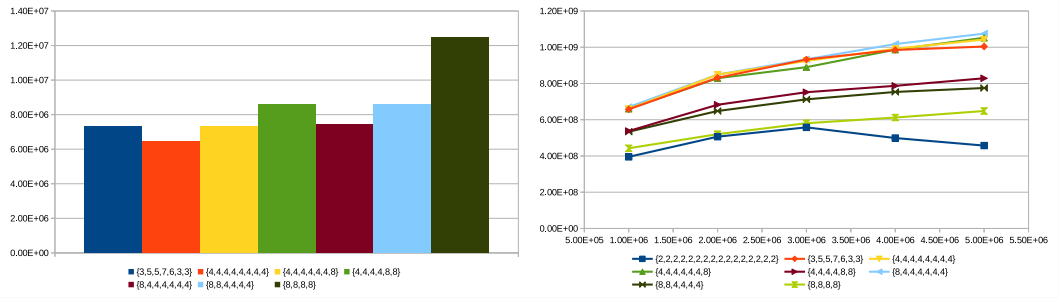


Figure 6: IP matching benchmark results.

Experimental results - STS benchmark

Simple benchmark for databases.

- inserting records to empty database and then reading them all
- in presented instance 1 mln records
- 19 digit keys, random values
- few other fields (2 integers, 2 floats, 2 dates)

Can be found on: *https://github.com/STSSoft/DatabaseBenchmark*

All results where taken from STS benchmark website.

Experimental results - STS benchmark

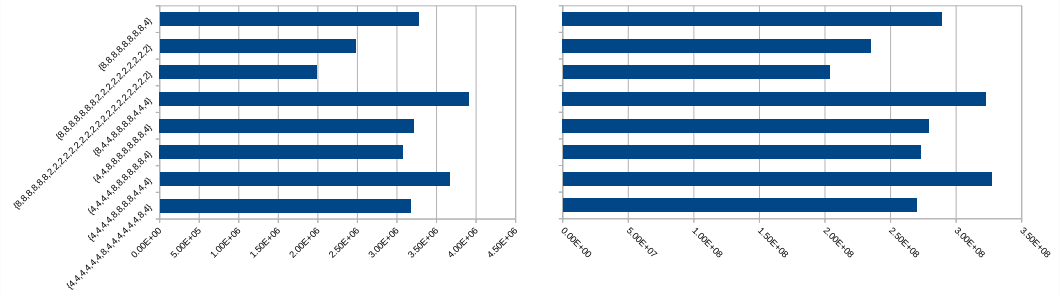


Figure 7: Results of STS database benchmark - throughput for inserting keys (left) and finding them (right) in keys per second for different configurations.

Experimental results - STS benchmark

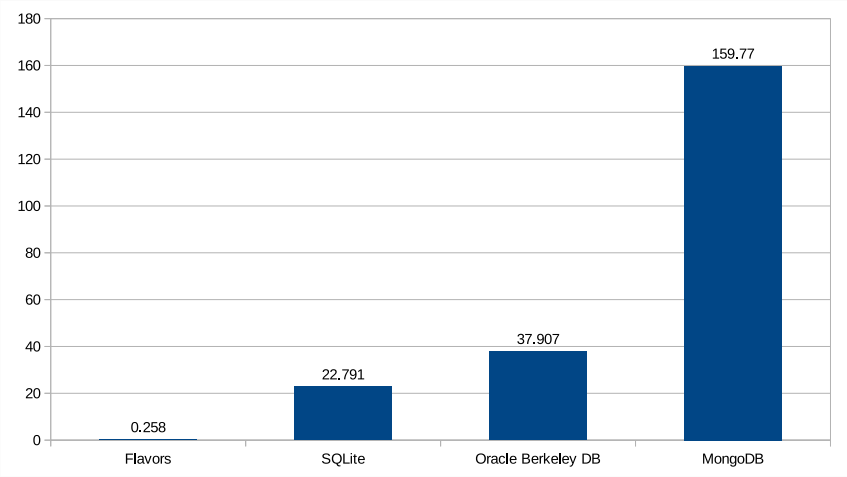


Figure 8: Benchmark times in milliseconds for different databases and Flavors

Experimental results - dictionary search

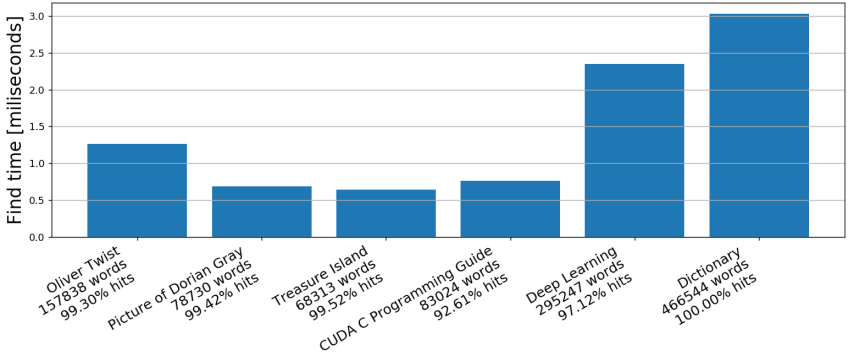


Figure 9: Times in milliseconds for finding all words from different books in english dictionary.

Why would you try it?

- For certain cases it is fast
- Beyond some point it scales reasonably for long keys
- Trees embody neighboring, unlike hash tables

Ongoing work

Obvious problem is, how to pick bit strides?

Ongoing work

Obvious problem is, how to pick bit strides?

There is algorithm to calculate bit strides, but:

- it is slow (dynamic programming)
- it aims to build tree with the smallest possible number of nodes

Ongoing work

Obvious problem is, how to pick bit strides?

Ongoing work

Obvious problem is, how to pick bit strides?

There is algorithm to calculate bit strides, but:

Ongoing work

Obvious problem is, how to pick bit strides?

There is algorithm to calculate bit strides, but:

- it is slow (dynamic programming)
- it aims to build tree with the smallest possible number of nodes

Ongoing work

Obvious problem is, how to pick bit strides?

There is algorithm to calculate bit strides, but:

- it is slow (dynamic programming)
- it aims to build tree with the smallest possible number of nodes

Problem is important for two reasons:

Ongoing work

Obvious problem is, how to pick bit strides?

There is algorithm to calculate bit strides, but:

- it is slow (dynamic programming)
- it aims to build tree with the smallest possible number of nodes

Problem is important for two reasons:

- performance
- ease of use

Ongoing work

Obvious problem is, how to pick bit strides?

There is algorithm to calculate bit strides, but:

- it is slow (dynamic programming)
- it aims to build tree with the smallest possible number of nodes

Problem is important for two reasons:

- performance
- ease of use

Experimental results - why configuration matters?

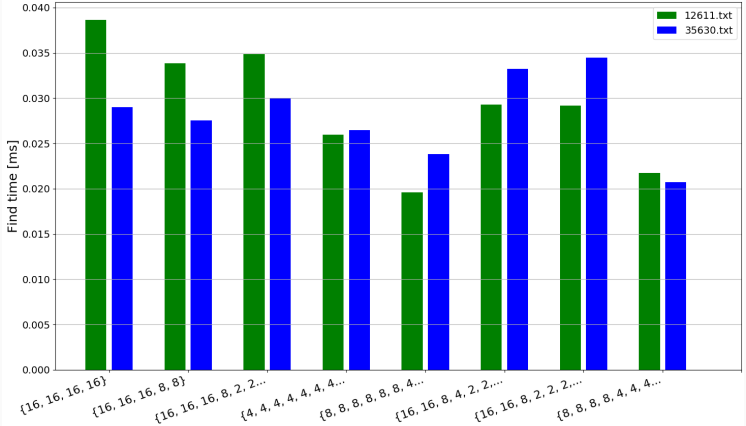


Figure 10: Times of find for two differnt dictionaries builded using different configurations.

Experimental results - top configurations

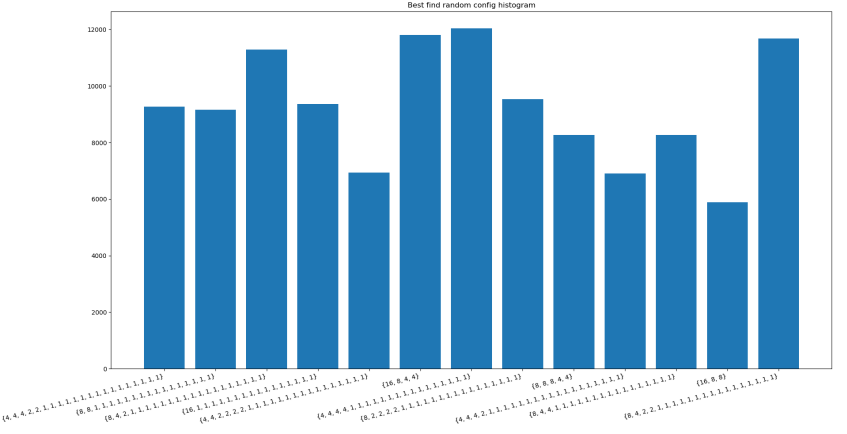


Figure 11: Histogram of best top 10 best configurations.

Ongoing work

Possible solution - machine learning

Ongoing work

Possible solution - machine learning

Aim would be to build a model, that could predict best possible configuration, based on data.

Ongoing work

Possible solution - machine learning

Aim would be to build a model, that could predict best possible configuration, based on data.

For now, I was able to reach about 87% accuracy using simple MLP network for predefined set of configurations.

Ongoing work

Possible solution - machine learning

Aim would be to build a model, that could predict best possible configuration, based on data.

For now, I was able to reach about 87% accuracy using simple MLP network for predefined set of configurations.

Working on model, that would generate configuration itself, based on the data.

You can find Flavors here:

https://github.com/wazka/flavors

Contact information:

albertwolant@gmail.com

@wazka3133

Thank you!

Q & A