Survey of Vectorized Database Techniques

CS 541

Group: VectDB2

Phase 2

Shivam Bhat (PUID : 33760929)
Venkat Sai (PUID : 34779773)
Akshay Ramakrishnan (PUID : 33660675)

Roadmap

In Milestone 1, we introduced the notion of vectorization in databases by covering all the basic operations and methods through which vectorization is achieved. We also covered the indexing implemented in such databases. We then gave a brief introduction to the database management system known as vectorwise that implements these methods.

In this milestone, we hope to cover more of the complex operations performed through vectorization. We also discuss the drawbacks of the previously introduced indexing and showcase a new indexing method and data structure. We then finally show how bloom filters can be implemented with the help of vectorization.

1. Rethinking SIMD Vectorization for In-Memory Databases

SIGMOD '15: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data

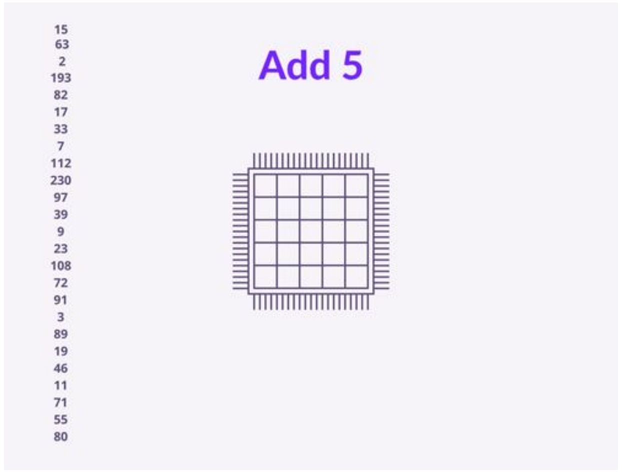
Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross

Source: Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645

Quick Recap

What is SIMD?

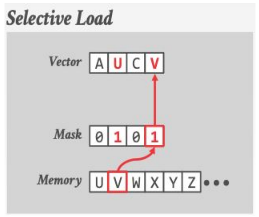
Single / same
Instruction / operation on
Multiple
Data (lanes)

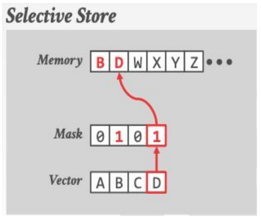


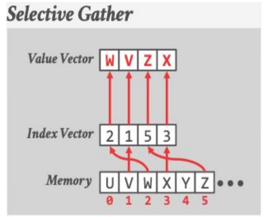
Quick Recap

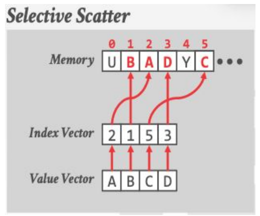
Fundamental Operations:

1. Selective Load
2. Selective Store
3. Selective Gather
4. Selective Scatter



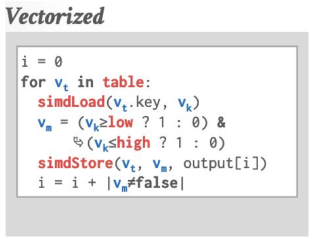






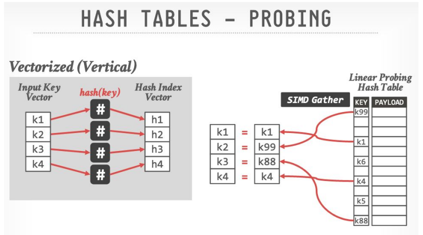
Vectorized Operator:

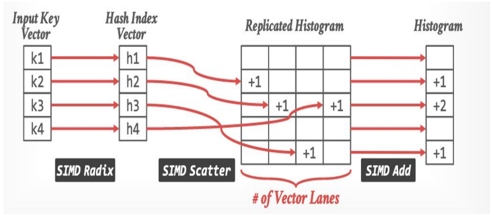
1. Selection Scans
2. Hash Tables
3. Partitioning



Vectorized

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &
       ↳(v_k≤high ? 1 : 0)
  simdStore(v_t, v_m, output[i])
  i = i + |v_m≠false|
```





Source: Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645

Query optimization

SELECT COUNT(*) **FROM** table
WHERE age > 20
GROUP BY city;

```
agg = dict()
for t in table:
    if t.age > 20:
    agg[t.city]['count']++
for t in agg:
    emit(t)
```

For each batch, the **SIMD vectors** may contain tuples that are no longer valid (Tuples that did not match the predicate checks and were hence disqualified by the previous check)

They are still present in cpu registers since we are still operating on them.

Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last

Proceedings of the VLDB Endowment, Volume 11

Prashanth Menon, Todd C. Mowry, and Andrew Pavlo.

Relaxed Operator Fusion

In this paper authors present a query processing model called “relaxed operator fusion” that **allows the DBMS to introduce staging points in the query plan** where intermediate results are temporarily materialized.

This allows the DBMS to take advantage of inter-tuple parallelism inherent in the plan using a **combination of prefetching and SIMD vectorization to support faster query execution** on data sets that exceed the size of CPU-level caches.

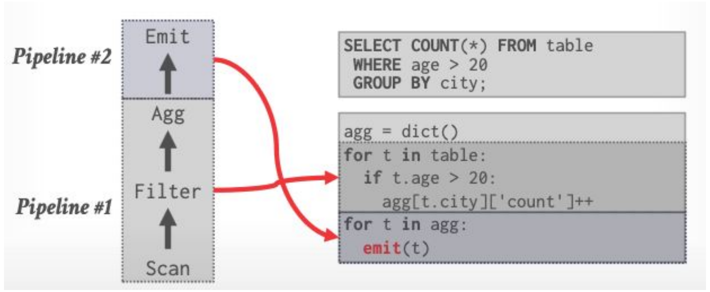
Author’s evaluation shows that their approach reduces the execution time of OLAP queries by up to **2.2x** and achieves up to **1.8x** better performance compared to other in-memory DBMSs.

Relaxed Operator Fusion

ROF enables the DBMS to structure the generated query code such that portions are combined together to take advantage of SIMD vectorization and software-level prefetching.

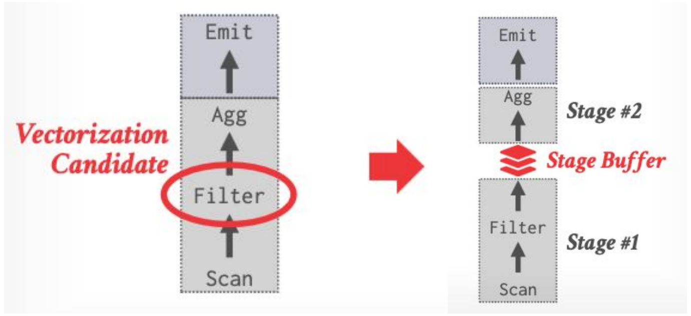
1. **Software Prefetch**: instructions can move blocks of data from memory into the CPU caches before they are needed, thereby hiding the latency of expensive cache misses
2. **SIMD** instructions can exploit vector-style data parallelism to boost computational throughput .

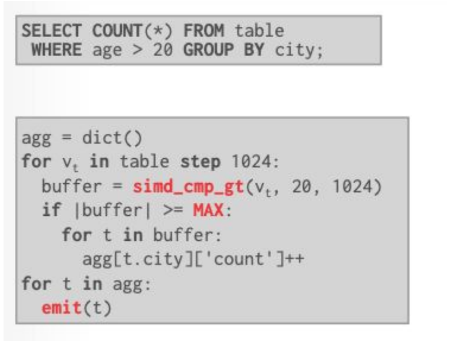
Relaxed Operator Fusion



```
SELECT COUNT(*) FROM table
 WHERE age > 20
 GROUP BY city;
```

```
agg = dict()
for t in table:
  if t.age > 20:
    agg[t.city]['count']++
for t in agg:
  emit(t)
```



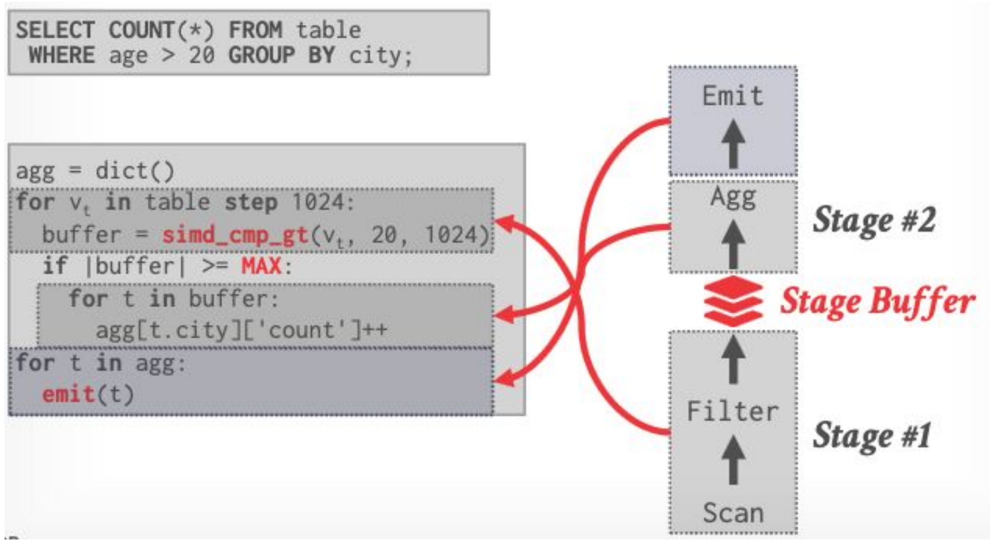


```
SELECT COUNT(*) FROM table
 WHERE age > 20 GROUP BY city;
```

```
agg = dict()
for v_t in table step 1024:
  buffer = simd_cmp_gt(v_t, 20, 1024)
  if |buffer| >= MAX:
    for t in buffer:
      agg[t.city]['count']++
for t in agg:
  emit(t)
```

Source: Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645
Andy P. CMU 15-721 Course

Relaxed Operator Fusion



```
SELECT COUNT(*) FROM table
 WHERE age > 20 GROUP BY city;
```

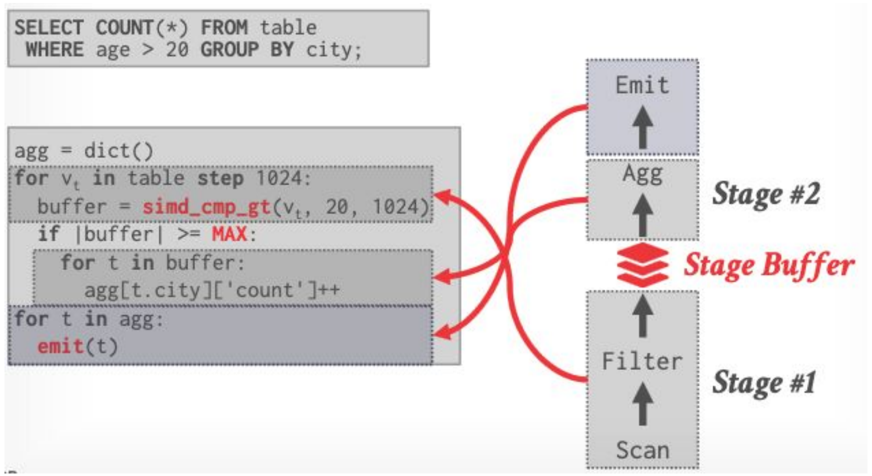
```
agg = dict()
for v_t in table step 1024:
  buffer = simd_cmp_gt(v_t, 20, 1024)
  if |buffer| >= MAX:
    for t in buffer:
      agg[t.city]['count']++
for t in agg:
  emit(t)
```

Source: Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645
Andy P, CMU 15-721 Course

Observations

We run a tighter outer loop which may occasionally jump into the inner loop.

We can give a hint to the CPU - “I am going to fetch this soon, can you fetch it for me?”



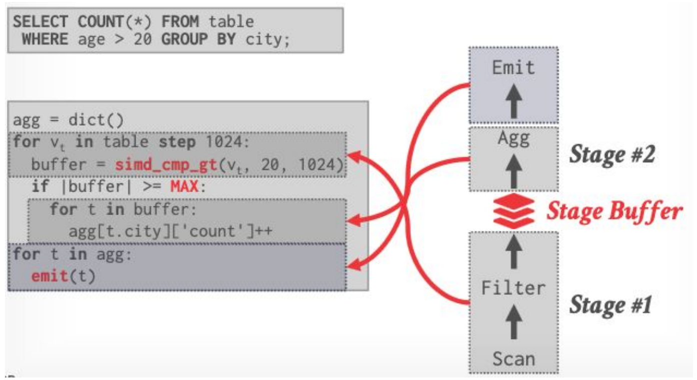
```
SELECT COUNT(*) FROM table
 WHERE age > 20 GROUP BY city;
```

```
agg = dict()
for v_t in table step 1024:
  buffer = simd_cmp_gt(v_t, 20, 1024)
  if |buffer| >= MAX:
    for t in buffer:
      agg[t.city]['count']++
for t in agg:
  emit(t)
```

Prefetching

The DBMS can tell the CPU to grab the next vector while it works on the current batch - Non Blocking Call

1. DBMS instead decomposes pipelines into stages.
2. A stage is a partition of a pipeline in which all operators are fused together.
3. Stages within a pipeline communicate solely through cache-resident vectors of tuple IDs.
4. Tuples are processed sequentially through operators in any given stage one-at-a-time.
5. If the tuple is valid in the stage, its ID is appended into the stage's output vector.



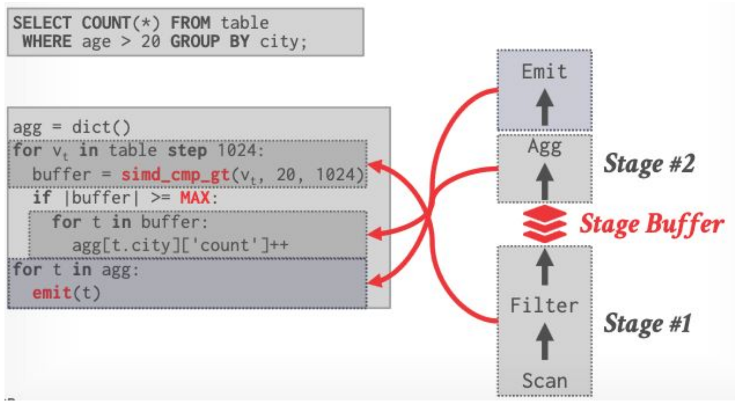
```
SELECT COUNT(*) FROM table
 WHERE age > 20 GROUP BY city;
```

```
agg = dict()
for v_t in table step 1024:
  buffer = simd_cmp_gt(v_t, 20, 1024)
  if |buffer| >= MAX:
    for t in buffer:
      agg[t.city]['count']++
for t in agg:
  emit(t)
```

Prefetching

1. Processing remains within a stage while the stage’s output vector is not full. If and when this vector reaches capacity, processing shifts to the next stage in the pipeline, where the output vector of the previous stage serves as the input to the current.

2. Since there is always exactly one active processing stage in ROF, we ensure both input and output vectors (when sufficiently small) will remain in the CPU’s caches.



```
SELECT COUNT(*) FROM table
 WHERE age > 20 GROUP BY city;

agg = dict()
for v_t in table step 1024:
  buffer = simd_cmp_gt(v_t, 20, 1024)
  if |buffer| >= MAX:
    for t in buffer:
      agg[t.city]['count']++
for t in agg:
  emit(t)
```

Source: Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645
Andy P, CMU 15-721 Course

Where do I put my stage?

A balance between time

Time required for processing **(Tp)** Vs Time to Prefetch Data **(Tf)**

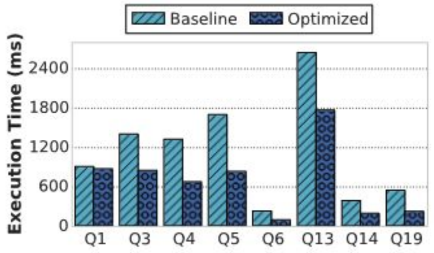
Tp<Tf : We are coming back prematurely to prefetch data

Tp>Tf : Too much time is spent in processing so by the time we come back this long untouched cache is already evicted by CPU leading to a **cache miss**.

Advantage

Distinguishing characteristic between ROF and traditional vectorized processing

“ROF stages always deliver a full vector of input to the next stage in the pipeline, unlike vectorized processing that may deliver input vectors restricted by a selection vector.”



Hash Tables - Linear Probing

1. Hash tables are used in database systems to execute joins and aggregations since they allow constant time key lookups.
2. In hash join, one relation is used to build the hash table and the other relation probes the hash table to find matches
3. Using SIMD instructions in hash tables has been proposed as a way to build bucketized hash tables.
4. Rather than comparing against a single key, we place multiple keys per bucket and compare them to the probing key using SIMD vector comparisons

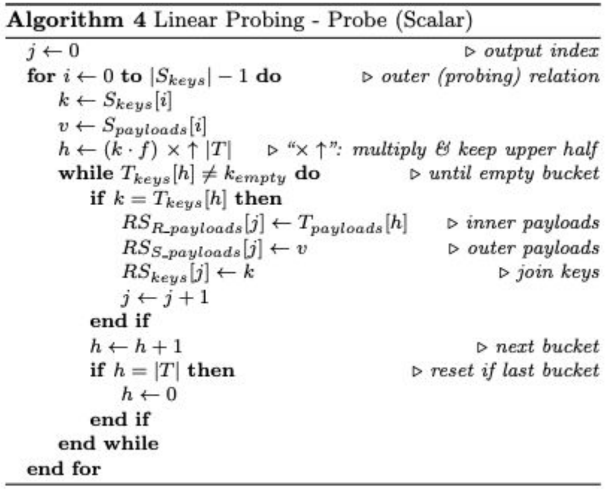
Hash Tables - Linear Probing

Hash table variants the paper discusses are

1. Linear probing
2. Double hashing
3. Cuckoo hashing

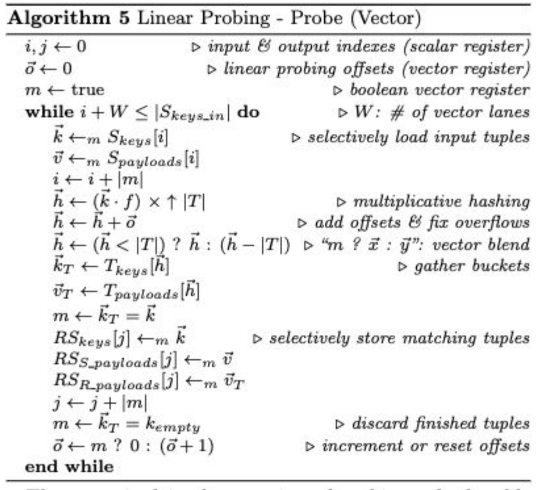
Linear Probing

Linear probing is an open addressing scheme that, to either insert an entry or terminate the search, traverses the table linearly until an empty bucket is found.



Algorithm 4 Linear Probing - Probe (Scalar)

```
$j \leftarrow 0$                                              ▷ output index
for $i \leftarrow 0$ to $|S_{keys}| - 1$ do                     ▷ outer (probing) relation
    $k \leftarrow S_{keys}[i]$
    $v \leftarrow S_{payloads}[i]$
    $h \leftarrow (k \cdot f) \times\!\uparrow |T|$      ▷ "$\times\uparrow$": multiply & keep upper half
    while $T_{keys}[h] \neq k_{empty}$ do                       ▷ until empty bucket
        if $k = T_{keys}[h]$ then
            $RS_{R\_payloads}[j] \leftarrow T_{payloads}[h]$    ▷ inner payloads
            $RS_{S\_payloads}[j] \leftarrow v$                  ▷ outer payloads
            $RS_{keys}[j] \leftarrow k$                         ▷ join keys
            $j \leftarrow j + 1$
        end if
        $h \leftarrow h + 1$                                    ▷ next bucket
        if $h = |T|$ then                                       ▷ reset if last bucket
            $h \leftarrow 0$
        end if
    end while
end for
```



Algorithm 5 Linear Probing - Probe (Vector)

```
$i, j \leftarrow 0$                      ▷ input & output indexes (scalar register)
$\vec{o} \leftarrow 0$                    ▷ linear probing offsets (vector register)
$m \leftarrow$ true                       ▷ boolean vector register
while $i + W \leq |S_{keys\_in}|$ do       ▷ W: # of vector lanes
    $\vec{k} \leftarrow_m S_{keys}[i]$                 ▷ selectively load input tuples
    $\vec{v} \leftarrow_m S_{payloads}[i]$
    $i \leftarrow i + |m|$
    $\vec{h} \leftarrow (\vec{k} \cdot f) \times\!\uparrow |T|$          ▷ multiplicative hashing
    $\vec{h} \leftarrow \vec{h} + \vec{o}$                      ▷ add offsets & fix overflows
    $\vec{h} \leftarrow (\vec{h} < |T|)\ ?\ \vec{h} : (\vec{h} - |T|)$   ▷ "$m\ ?\ \vec{x} : \vec{y}$": vector blend
    $\vec{k}_T \leftarrow T_{keys}[\vec{h}]$                    ▷ gather buckets
    $\vec{v}_T \leftarrow T_{payloads}[\vec{h}]$
    $m \leftarrow \vec{k}_T = \vec{k}$
    $RS_{keys}[j] \leftarrow_m \vec{k}$              ▷ selectively store matching tuples
    $RS_{S\_payloads}[j] \leftarrow_m \vec{v}$
    $RS_{R\_payloads}[j] \leftarrow_m \vec{v}_T$
    $j \leftarrow j + |m|$
    $m \leftarrow \vec{k}_T = k_{empty}$                ▷ discard finished tuples
    $\vec{o} \leftarrow m\ ?\ 0 : (\vec{o} + 1)$          ▷ increment or reset offsets
end while
```

Source: Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645

The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases

Viktor Leis, Alfons Kemper, Thomas Neumann - **2013 IEEE 29th ICDE**

We are going to introduce a new **data structure ART** which performs better for range queries in comparison to hashing by leveraging prefetching, **SIMD parallelism** and thread parallelism.

Leis, V., Kemper, A., & Neumann, T. (2013). The adaptive radix tree: Artful indexing for main-memory databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde.2013.6544812

Vectorization in ART

1. ART is an improvement over Hash tables in terms of vectorization. *ART has a predictive memory layout which enables it to use SIMD instruction* better compared to Hash tables.

2. The main advantage of ART would lie in the usage of prefixes. During a search operation ART can fit more keys into a SIMD register than the standard Hash table as it only fits the prefixes rather than the whole keys. More in the next slides.

Source: https://www.the-paper-trail.org/post/art-paper-notes/

Hashing: Is it the way to go?

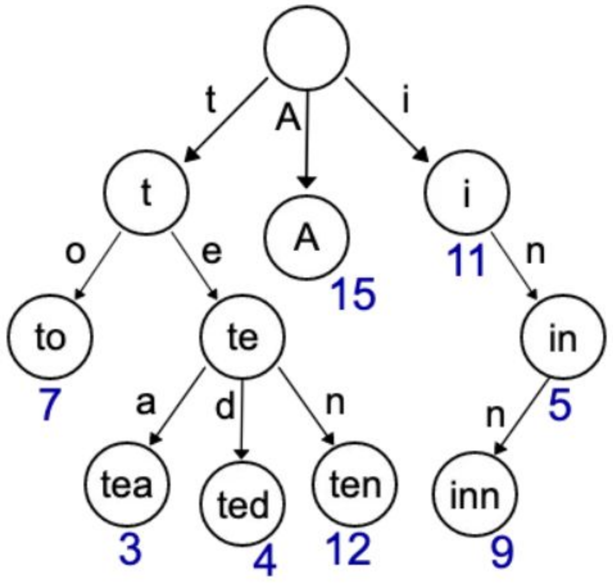
1. Hash table resorts to linear search when it comes to range queries.
2. Hash tables have a worst-case lookup time of O(n), where n is the number of keys. This can happen if all the keys happen to hash to the same index, leading to a long linked list that needs to be traversed.
3. When the number of keys in a hash table grows too large or too small, the table needs to be resized, which can be an expensive operation.

Tries

1. Tries are an unloved third data structure for building key-value stores and indexes, after search trees and hash tables.
2. Yet they have a number of very appealing properties that make them worthy of consideration.
3. Weighing against those advantages is the heavy memory cost that vanilla radix tries can incur, because each node contains a pointer for every possible value of the ‘next’ character in the key.

Source: https://www.the-paper-trail.org/post/art-paper-notes/

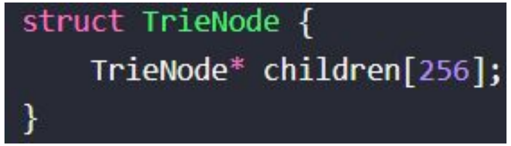
How Trie Look Like



Source: Wikipedia

Everything wrong with Trie

1. Here we're assuming an 8-bit alphabet, with 256 possible characters per letter, or a *fan-out* of 256.
2. Imagine a radix trie containing foo, fox and fat. The root node would have one valid pointer, to f, which would have two valid pointers, to a and o. a would have a pointer to t, and o would have pointers to o and x.
3. So our trie would have 6 nodes, but a total of 6 * 26 = 156 pointers, of which 150 / 156 = 96% are empty

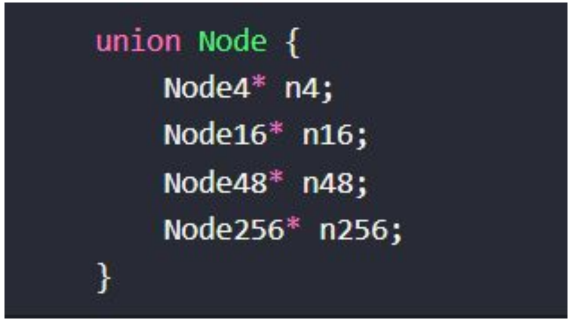


```
struct TrieNode {
    TrieNode* children[256];
}
```

Source: https://www.the-paper-trail.org/post/art-paper-notes/

ART : Fixing the Trie

1. The most significant change that ART makes to the standard trie structure is that it introduces the ability to change the data structure used for each internal node depending on how many children the node actually has, rather than how many it might have.
2. Each node stores a prefix of the key which could be used in SIMD instructions for search rather than the whole key.



```
union Node {
    Node4* n4;
    Node16* n16;
    Node48* n48;
    Node256* n256;
}
```

Leis, V., Kemper, A., & Neumann, T. (2013). The adaptive radix tree: Artful indexing for main-memory databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde.2013.6544812

Parallelism

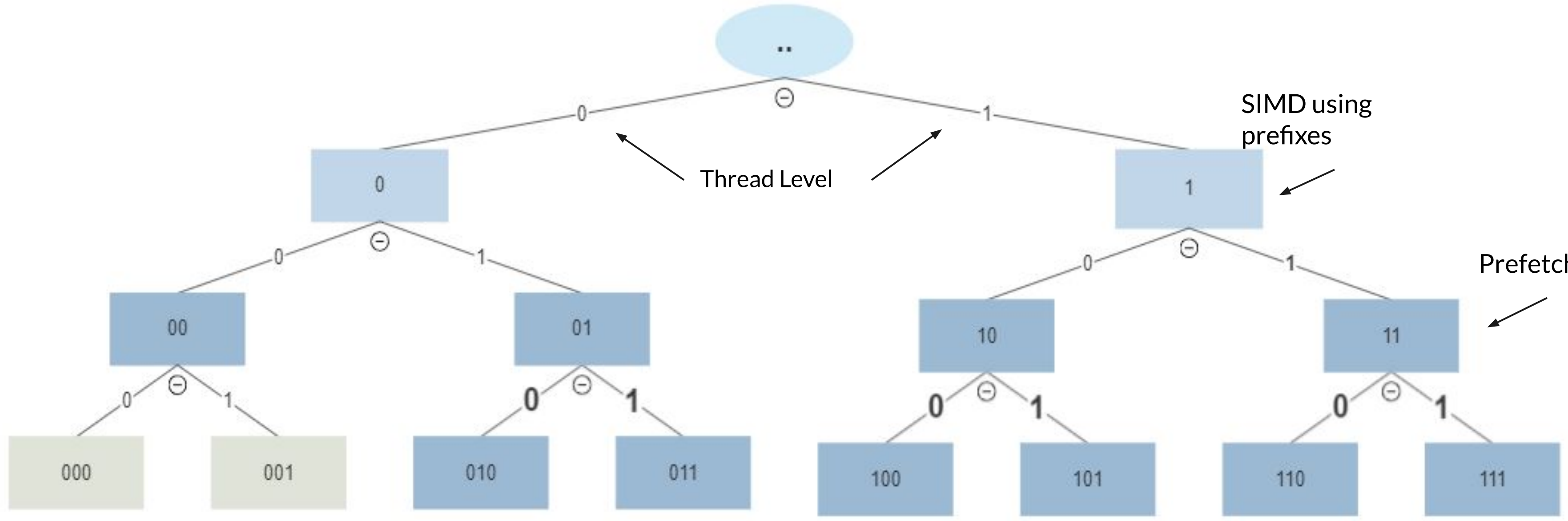
1. ART makes use of **hardware prefetching** by using the _*mm_prefetch instruction* provided by Intel's SIMD instruction set. (A node can fit into cache).
2. In terms of SIMD parallelism, **ART can take advantage of SIMD instructions** to compare multiple keys in parallel during a range query.
3. In terms of thread-level parallelism, ART can be easily **parallelized by splitting the tree into multiple subtrees and processing each subtree in parallel using multiple threads.**

Leis, V., Kemper, A., & Neumann, T. (2013). The adaptive radix tree: Artful indexing for main-memory databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde.2013.6544812

Adaptive Radix Tree - Best of Both the Worlds

1. Suppose we have an ART tree with 1 million keys and we want to perform a range query to find all keys between 500,000 and 750,000.
2. With SIMD parallelism, ART can use SIMD instructions to compare multiple keys in parallel during the range query.
3. With thread-level parallelism, ART can be split into multiple subtrees and processed in parallel using multiple threads.

Range Search Using ART



Vectorization

Let’s take a simple probing operation. Let’s say both Hashing and ART have arrived at this bucket.

Now if the SIMD register is 16 bits long. Hashing would need to do 2 SIMD compares for it to arrive at 0010. But at the same time, if ART uses 2 bit prefix in this bucket, it can arrive at the answer using just 1 SIMD compare.

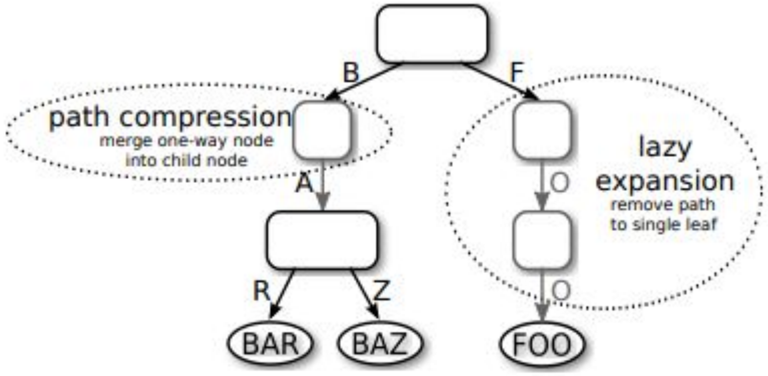
Keys	Values
1011	5
1010	3
1000	8
1001	6
1111	9
1110	5
0111	8
0010	8

Methods to overcome searching Overhead

1. **Lazy Expansion:** When the sequence of nodes ends in a leaf, *lazy expansion* is used - don't bother to write out the full sequence, just create a single node that has the key suffix and the value.
2. **Path Compression:** This also collapses the sequence of nodes, but into the node at the end of the sequence that has more than one child.

Leis, V., Kemper, A., & Neumann, T. (2013). The adaptive radix tree: Artful indexing for main-memory databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde.2013.6544812

Path Compression and Lazy Expansion



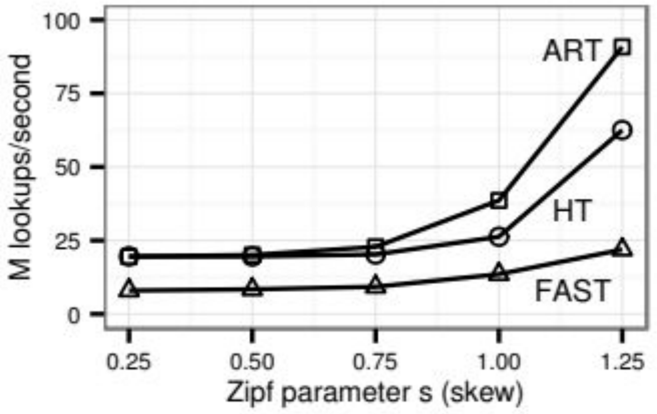
B
F
path compression
merge one-way node into child node
A
lazy expansion
remove path to single leaf
O
O
R
Z
BAR
BAZ
FOO

Fig. 6. Illustration of lazy expansion and path compression.

Leis, V., Kemper, A., & Neumann, T. (2013). The adaptive radix tree: Artful indexing for main-memory databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde.2013.6544812

ART's Performance

1. Usually when one considers any tree-based search structure, they do so only if range queries are needed, because otherwise the received wisdom is that a hash table's performance will wipe the floor with the tree.
2. Things get more interesting when the access pattern is skewed, as it allows ART to take advantage of cache effects, whereas hash tables have no data locality when there is locality in the query set.



Leis, V., Kemper, A., & Neumann, T. (2013). The adaptive radix tree: Artful indexing for main-memory databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde.2013.6544812

Comparision

- In simple terms: a perfectly balanced binary search tree has depth $\log_2 N$ (N is the number of keys in the tree or trie). So we need to know the value of N for which $\log_2 N > k/s$, which is true when $N > 2^{k/s}$.
- In concrete terms, let's say we have 64-bit keys, and are using 8-bits per character (standard ASCII). Then a trie will be shallower than a binary search tree when they contain more than 256 entries.
- ART performs better than a chained hash table implementation when the key set is 'dense'; i.e. all integers from 1..*N* are in the set.

3. Vectorized Bloom Filters for Advanced SIMD Processors

DaMoN '14: Proceedings of the Tenth International Workshop on **Data Management on New Hardware**;

Authors: Orestis Polychroniou, Kenneth A. Ross

Source: Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14). Association for Computing Machinery, New York, NY, USA, Article 6, 1–6. https://doi.org/10.1145/2619228.2619234

Establishing the Problem Statement:

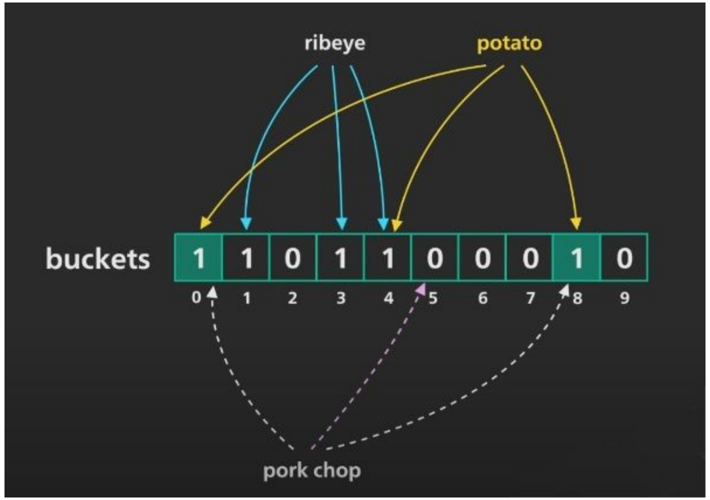
1. We have seen SIMD being useful in operations where data residency is sequential. For example, we have seen SIMD being useful in scan operations, sorting, etc.
2. The question remains as to whether SIMD is useful in operations that involve random access patterns.
3. Mainstream hardware offers non-contiguous SIMD loads termed “gathers”.
4. Using these SIMD operations, the paper presents an efficient technique for bloom filter probing.

Source: Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14). Association for Computing Machinery, New York, NY, USA, Article 6, 1–6. https://doi.org/10.1145/2619228.2619234

What is a Bloom Filter?

1. A Bloom Filter is a probabilistic data structure that determines whether an element is part of a set.
2. They use a bit array and multiple hash functions to represent a set of items.
3. When a new item is added to the set, it is hashed by each of the hash functions and the corresponding bits in the bit array are set to 1.
4. To test if an item is in the set, it is hashed by each of the hash functions and the corresponding bits are checked. If all of them are set to 1, the item is probably in the set.

How do they work?



Source: YouTube: Bloom Filters | Algorithms You Should Know #2 | Real-world Examples: https://www.youtube.com/watch?v=V3pzxngeLqw&t=268s

Why is it useful for databases?

1. **Bloom filters** are crucial to analytical databases for **performing joins** **between tables that have vastly different cardinalities.**
2. Consider a case where we join a smaller table (with lesser cardinality) with a larger table (with higher cardinality).
3. We can create a bloom filter using the keys of the smaller table and the keys of the larger table are probed through the filter to remove unwanted tuples.
4. So, a large amount of tuples are filter in the larger table before joining with the smaller table. This process is also known as **semi-joins.**

Source: Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14). Association for Computing Machinery, New York, NY, USA, Article 6, 1–6. https://doi.org/10.1145/2619228.2619234

Scalar Implementation

1. In the scalar implementation, we load **one key at a time** and we iteratively check over all the hash functions.
2. If the bits of all the hash functions are set, then we can conclude that the key probably belongs to the set.
3. If not, the key is discarded

```
for (o = i = 0 ; i != size ; ++i) {
  // load key and check one function at a time
  key = keys[i];
  hash = key * factor_1;  // 1st function
  hash >>= shift;
  // 1st function branch (1st prediction state)
  asm goto ("btl  %1, (%0)\n\t"
            "jnc  %l[failure]"
  :: "r"(filter), "r"(hash) : "cc" : failure);
  hash = key * factor_2;  // 2nd function
  hash >>= shift;
  // 2nd function branch (2nd prediction state)
  asm goto ("btl  %1, (%0)\n\t"
            "jnc  %l[failure]"
  :: "r"(filter), "r"(hash) : "cc" : failure);
  [...]  // hard-code more functions if applicable
  // tuples qualifies since all bits are set
  pays_out[o] = pays[i];
  keys_out[o++] = key;
  failure:; }
```

Source: Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14). Association for Computing Machinery, New York, NY, USA, Article 6, 1–6. https://doi.org/10.1145/2619228.2619234

How is this vectorized?

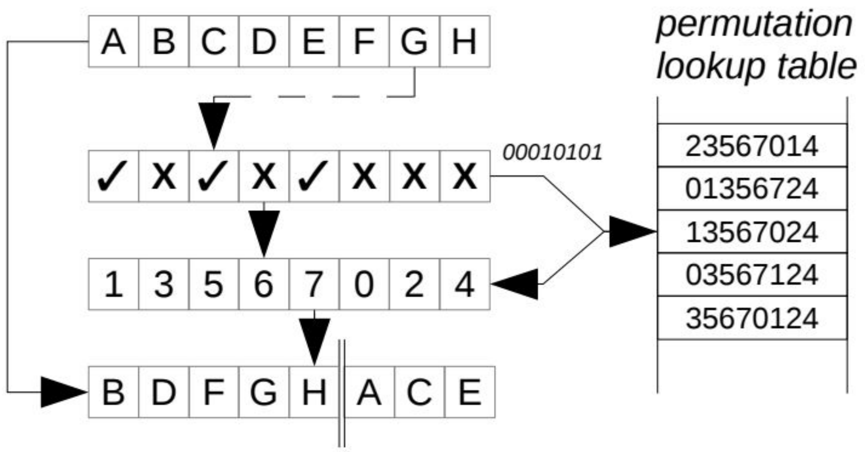
1. Ideology is that testing all the hash functions for every key is a wasteful approach because tuples which have failed a bit test would not be able to abort early.
2. Hence, we vectorize access “across” keys. The gather operation should check one hash function for ***W*** keys given that our SIMD vector can hold ***W*** keys

Source: Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14). Association for Computing Machinery, New York, NY, USA, Article 6, 1–6. https://doi.org/10.1145/2619228.2619234

1. The outline of the entire process is given as follows:
    a. Load ***W*** keys, one for each SIMD lane. (If any of the keys failed in the previous iteration, replace them by gathering new keys.
    b. Hash the keys using **multiplicative hashing.**
    c. Once we obtain the set bits, we need to **shuffle**.
    d. Why do we shuffle?
    e. How is it shuffled? **Permutation Lookup Table**

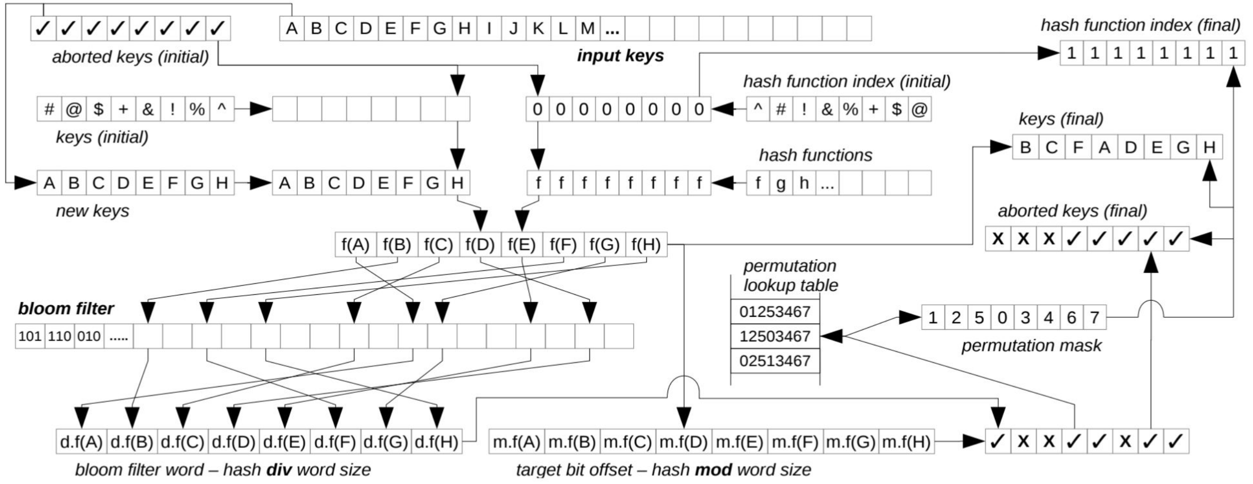
Source: Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14). Association for Computing Machinery, New York, NY, USA, Article 6, 1–6. https://doi.org/10.1145/2619228.2619234

Shuffling operation:



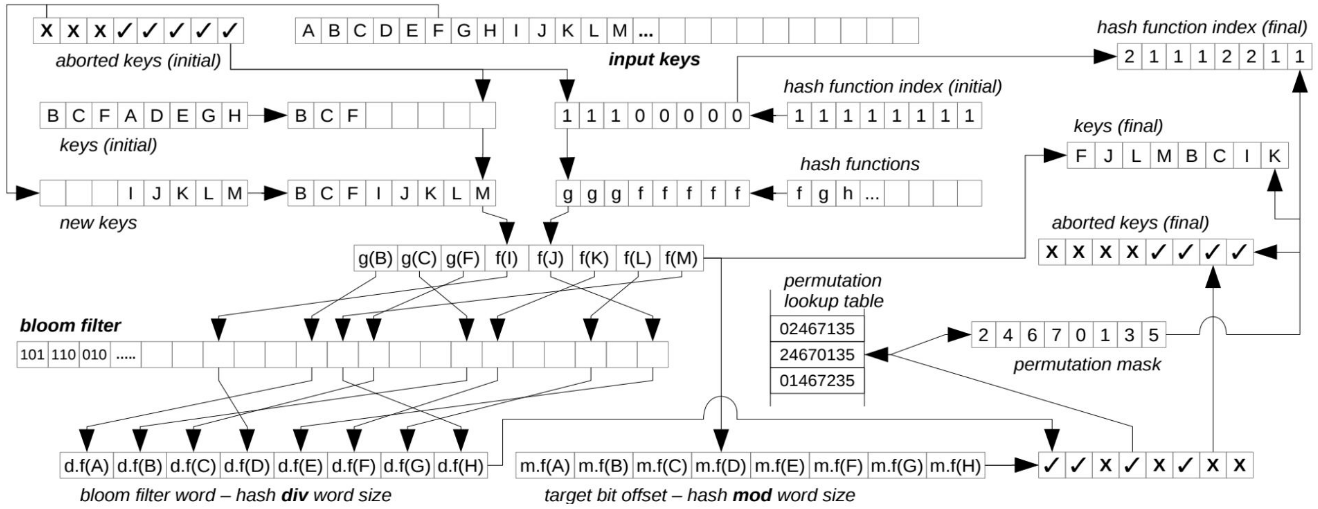
Source: Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14). Association for Computing Machinery, New York, NY, USA, Article 6, 1–6. https://doi.org/10.1145/2619228.2619234

First Loop of Vectorized Bloom Filter Probing



Source: Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14). Association for Computing Machinery, New York, NY, USA, Article 6, 1–6. https://doi.org/10.1145/2619228.2619234

Second Loop of Vectorized Bloom Filter Probing



Source: Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14). Association for Computing Machinery, New York, NY, USA, Article 6, 1–6. https://doi.org/10.1145/2619228.2619234

Performance Measure:

The authors' evaluation indicates a significant performance improvement over scalar code that can exceed **3X** (3.3X to be exact) when the Bloom filter is **cache-resident.**

Source: Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14). Association for Computing Machinery, New York, NY, USA, Article 6, 1–6. https://doi.org/10.1145/2619228.2619234

Future Work

1. Analysis of various vectorized operators. (Going further into it)
2. Overcoming challenges for vectorization in databases
3. Vectorization Implementation in modern databases
4. Deep diving into surveyed papers and papers mentioned in Milestone 1 and 2.

References

1. Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645
2. Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. Proc. VLDB Endow. 11, 1 (September 2017), 1–13. https://doi.org/10.14778/3151113.3151114
3. Leis, V., Kemper, A., & Neumann, T. (2013). The adaptive radix tree: Artful indexing for main-memory databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/icde.2013.6544812 https://db.in.tum.de/~leis/papers/ART.pdf
4. Beating hash tables with trees? The ART-ful radix trie https://www.the-paper-trail.org/post/art-paper-notes/
5. Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In Proceedings of the Tenth International Workshop on Data Management on New Hardware (DaMoN '14). Association for Computing Machinery, New York, NY, USA, Article 6, 1–6. https://doi.org/10.1145/2619228.2619234

Thanks