Survey of Vectorized Database Techniques

CS 541

Group: VectDB2

Phase 3

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 Milestone 2, we covered more of the complex operations performed through vectorization. We also discussed the drawbacks of the previously introduced indexing and showcase a new indexing method and data structure. We then finally showed how bloom filters can be implemented with the help of vectorization.

Roadmap

In Milestone 3, we continue to discuss previous operations that utilize vectorization and the corresponding increases in efficiency. We speak about Cuckoo Hashing in detail, and the process of vectorizing it. We later discuss more about Datablocks, which is a columnar storage format that allows for faster OLAP and OLTP queries. Finally, we shed light on another application of vectorization, specifically, filtering encoded data using SIMD algorithms.

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

Fundamental Operations:

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

Vectorized Operator:

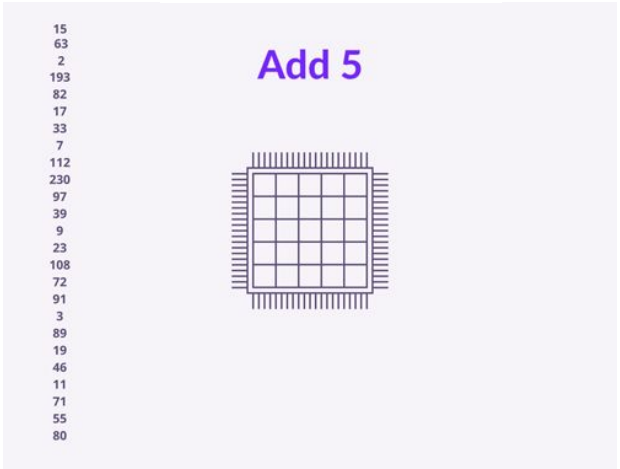
1. Selection Scans
2. Hash Tables
3. Partitioning

Relaxed Operator Fusion

1. ROF-Query Restructure
2. Staging Buffers
3. Software Prefetch
4. Vectorized Linear Hashing & Probing

What is SIMD?

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



Cuckoo Hashing

1. It's the only hash table scheme that has been vectorized in previous work as a means to allow multiple keys per bucket (horizontal vectorization)

2. Authors study cuckoo hashing to compare their (vertical vectorization) approach against previous work

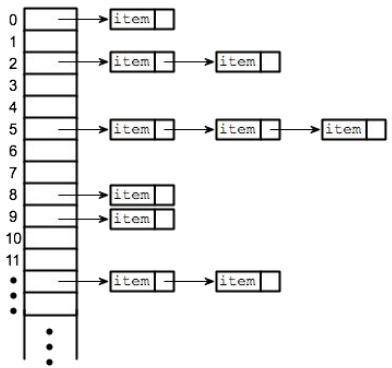
3. Paper also show that complicated control flow logic, such as cuckoo table building, can be transformed to data flow vector logic.

K. A. Ross. Efficient hash probes on modern processors. ICDE, pages 1297–1301, 2007
M. Zukowski et al. Architecture-conscious hashing. DaMoN, 2006

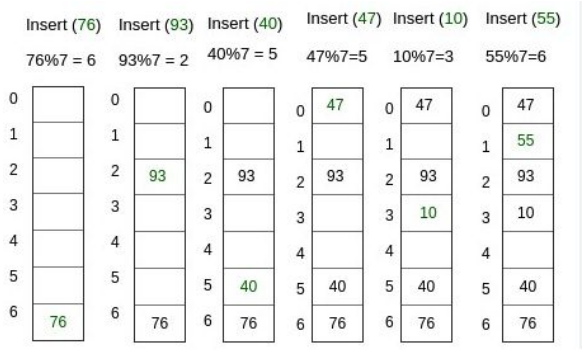
Cuckoo Hashing

General strategies towards resolving hash collisions:

1. Closed Addressing or Chaining : Colliding elements are stored in an auxiliary data structure like a linked list.
2. Open Addressing: This scheme allows elements to overflow out of their target bucket and into other spaces.



Closed Addressing



Open Addressing

Cuckoo Hashing

Although the lookup cost as O(1), the expected worst-case cost of a lookup in Open Addressing (with linear probing) is Ω (log n) and Θ(log n / log log n) in simple chaining (Source : Standford Lecture Notes). To close the gap of expected time and worst case expected time, two ideas are used:

- **Multiple-choice hashing**: Give each element multiple choices for positions where it can reside in the hash table
- **Relocation hashing**: Allow elements in the hash table to move after being placed

Cuckoo hashing applies the idea of multiple-choice and relocation together and guarantees O(1) worst case lookup time.

Cuckoo hashing is another hashing scheme that uses multiple hash functions.

Cuckoo Hashing- Idea

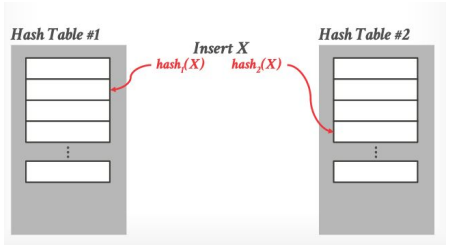
Cuckoo Hashing scheme uses multiple tables with different hash functions.

- On insert, check every table and pick anyone that has a free slot.
- If no table has a free slot, evict the element from one of them and then re-hash it find a new location.

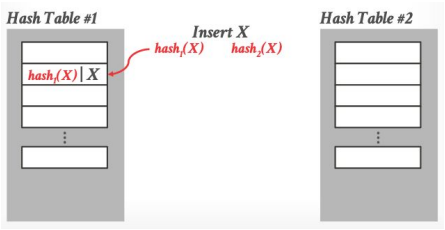
Table lookups are always O(1) because only one location per hash table is checked

Cuckoo Hashing - In action

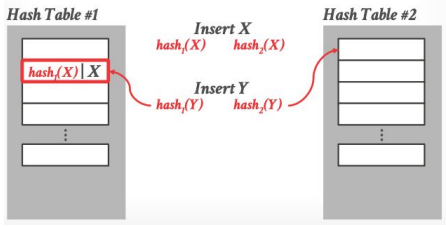
1



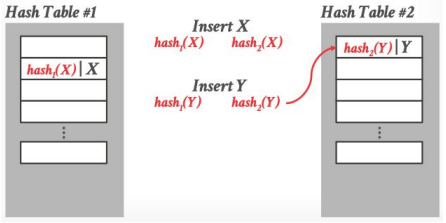
2



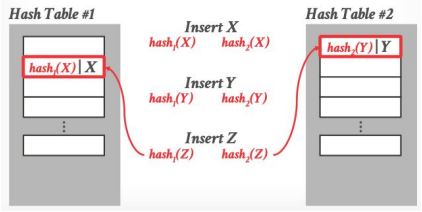
3



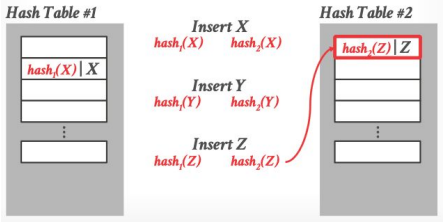
4



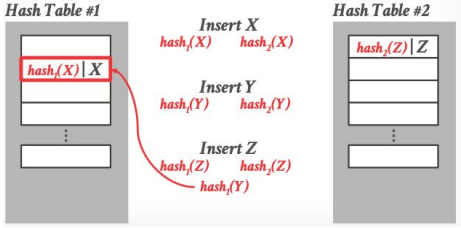
5



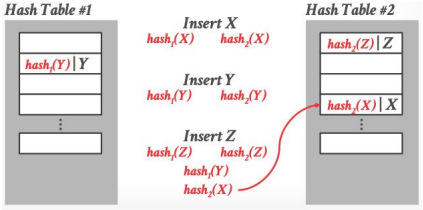
6



7



8



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

Vectorizing Cuckoo Hashing

Cuckoo hashing probing can be implemented in two ways.

- We check the second bucket only if the first bucket does not match.
- Always access both buckets and blend their results using bitwise operations
- The latter approach eliminates branching at the expense of always accessing both buckets.
- This approach has been shown to be faster than other variants on CPUs

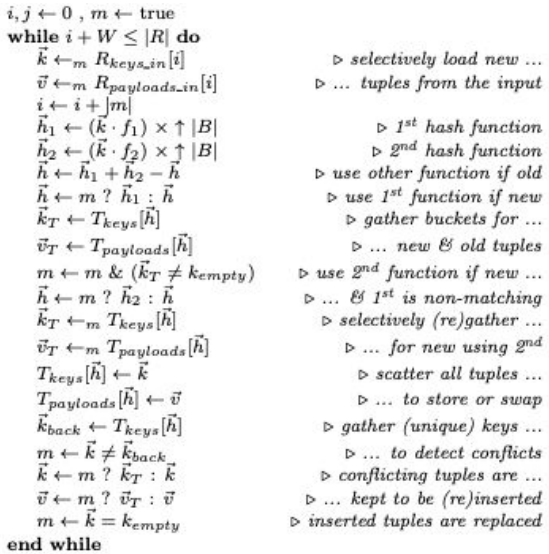
$j \leftarrow 0$
for $i \leftarrow 0$ **to** $|S| - 1$ **step** W **do**
  $\vec{k} \leftarrow S_{keys}[i]$ ▷ *load input tuples*
  $\vec{v} \leftarrow S_{payloads}[i]$
  $\vec{h_1} \leftarrow (\vec{k} \cdot f_1) \times \uparrow |T|$ ▷ 1^{st} *hash function*
  $\vec{h_2} \leftarrow (\vec{k} \cdot f_2) \times \uparrow |T|$ ▷ 2^{nd} *hash function*
  $\vec{k}_T \leftarrow T_{keys}[\vec{h_1}]$ ▷ *gather* 1^{st} *function bucket*
  $\vec{v}_T \leftarrow T_{payloads}[\vec{h_1}]$
  $m \leftarrow \vec{k} \neq \vec{k}_T$
  $\vec{k}_T \leftarrow_m T_{keys}[\vec{h_2}]$ ▷ *gather* 2^{nd} *function bucket ...*
  $\vec{v}_T \leftarrow_m T_{payloads}[\vec{h_2}]$ ▷ *... if* 1^{st} *is not matching*
  $m \leftarrow \vec{k} = \vec{k}_T$
  $RS_{keys}[j] \leftarrow_m \vec{k}$ ▷ *selectively store matches*
  $RS_{S\_payloads}[j] \leftarrow_m \vec{v}$
  $RS_{R\_payloads}[j] \leftarrow_m \vec{v}_T$
  $j \leftarrow j + |m|$
end for

Vectorized cuckoo table probing

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

Vectorizing Cuckoo Hashing

1. Building a cuckoo hashing table is more complicated.
2. If both bucket choices are not empty, we create space by displacing the tuple of one bucket to its alternate location.
3. This process may be repeated until an empty bucket is reached
4. This algorithm reuses vector lanes to load new tuples from the input. The remaining lanes are either previously conflicting or displaced tuples.
5. The newly loaded tuples gather buckets using one or both hash functions to find an empty bucket.
6. The tuples that were carried from the previous loop use the alternative hash function compared to the previous loop.



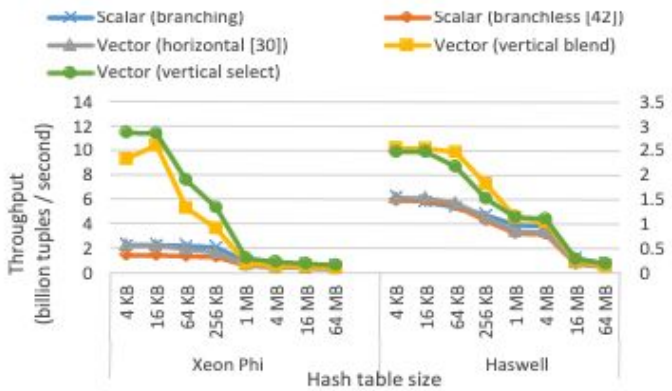
$i, j \leftarrow 0$, $m \leftarrow$ true
while $i + W \leq |R|$ **do**
- $\vec{k} \leftarrow_m R_{keys\_in}[i]$ ▷ *selectively load new ...*
- $\vec{v} \leftarrow_m R_{payloads\_in}[i]$ ▷ *... tuples from the input*
- $i \leftarrow i + |m|$
- $\vec{h}_1 \leftarrow (\vec{k} \cdot f_1) \times \uparrow |B|$ ▷ *1st hash function*
- $\vec{h}_2 \leftarrow (\vec{k} \cdot f_2) \times \uparrow |B|$ ▷ *2nd hash function*
- $\vec{h} \leftarrow \vec{h}_1 + \vec{h}_2 - \vec{h}$ ▷ *use other function if old*
- $\vec{h} \leftarrow m \ ? \ \vec{h}_1 : \vec{h}$ ▷ *use 1st function if new*
- $\vec{k}_T \leftarrow T_{keys}[\vec{h}]$ ▷ *gather buckets for ...*
- $\vec{v}_T \leftarrow T_{payloads}[\vec{h}]$ ▷ *... new & old tuples*
- $m \leftarrow m \ \&\ (\vec{k}_T \neq k_{empty})$ ▷ *use 2nd function if new ...*
- $\vec{h} \leftarrow m \ ? \ \vec{h}_2 : \vec{h}$ ▷ *... & 1st is non-matching*
- $\vec{k}_T \leftarrow_m T_{keys}[\vec{h}]$ ▷ *selectively (re)gather ...*
- $\vec{v}_T \leftarrow_m T_{payloads}[\vec{h}]$ ▷ *... for new using 2nd*
- $T_{keys}[\vec{h}] \leftarrow \vec{k}$ ▷ *scatter all tuples ...*
- $T_{payloads}[\vec{h}] \leftarrow \vec{v}$ ▷ *... to store or swap*
- $\vec{k}_{back} \leftarrow T_{keys}[\vec{h}]$ ▷ *gather (unique) keys ...*
- $m \leftarrow \vec{k} \neq \vec{k}_{back}$ ▷ *... to detect conflicts*
- $\vec{k} \leftarrow m \ ? \ \vec{k}_T : \vec{k}$ ▷ *conflicting tuples are ...*
- $\vec{v} \leftarrow m \ ? \ \vec{v}_T : \vec{v}$ ▷ *... kept to be (re)inserted*
- $m \leftarrow \vec{k} = k_{empty}$ ▷ *inserted tuples are replaced*

end while

Vectorized Cuckoo Hashing - Building

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

Vectorized Cuckoo Hashing is faster



Probe cuckoo hashing table

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

Vectorization Issue

We generally cannot do SIMD operation variable width column generally

This is because SIMD needs fixed alignment for the values of the column to be loaded into a SIMD register in a single instruction.

However we can use dictionary encoding compression scheme to encode your variable width column values into fixed width dictionary values and then write your query processing algorithm on top of it, on top of the dictionary values.

This way instead of working on the actual variable width non-compressed column values, we are working on fixed width dictionary encoded values.

2. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation

SIGMOD '16: Proceedings of the 2016 International Conference on Management of Data June 2016
Authors:Harald Lang, Tobias Mühlbauer, Florian Funke, Peter Boncz,, Thomas Neumann, Alfons Kemper

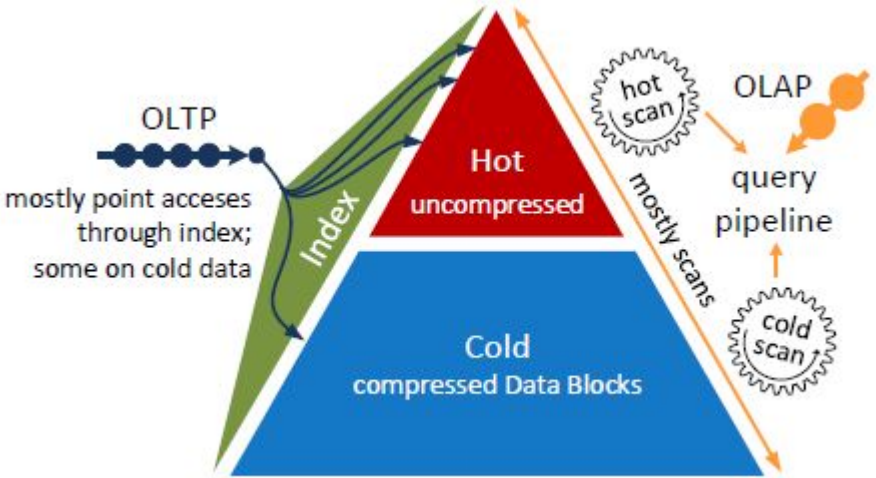
Source: Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

Goal of The Paper

- Data Blocks, a novel compressed columnar storage format for hybrid database systems,
- Light-weight indexing on compressed data for improved scan performance,
- SIMD optimized algorithms for predicate evaluation, and
- A blueprint for the integration of multiple storage layout combinations in a compiling tuple-at-a-time query engine by using vectorization.

Source: Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

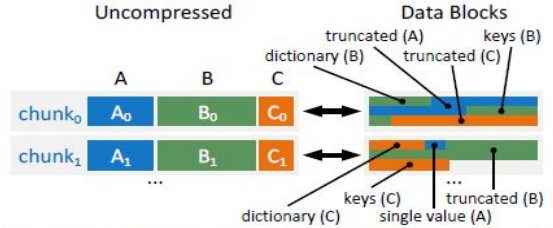
OLTP and OLAP



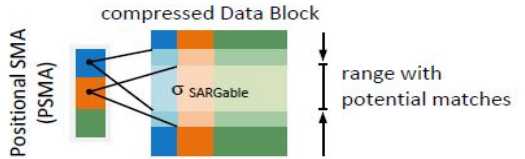
High Level Idea behind Data Blocks.

Source: Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

Data Blocks



(a) Optimal compression techniques are chosen for each column in each block.



(b) Data Blocks store not only min and max values of attributes to skip blocks as a whole, but also light-weight indexes called Positional SMAs (PSMAs) that narrow the scan ranges on a per-attribute basis.

Source: Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

Vectorized Scans In Compiling Query Engines

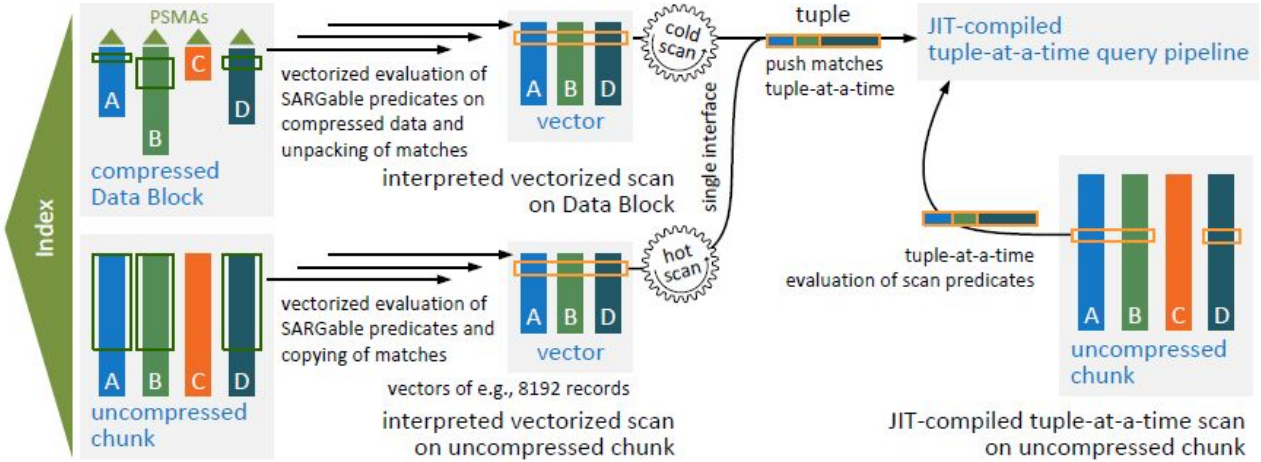
1. A query is broken down into multiple pipelines where each pipeline loads a tuple out of a materialized state (e.g., a base relation or a hash table)

2. It then performs the logic of all operators that can work on it without materialization, and finally materializes the output into the next pipeline breaker (e.g., a hash table).

3. This was discussed in the last milestone as relaxed operator fusion in pipelines.

Source: Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

Vectorized Scans In Compiling Query Engines



Source: Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

Finding and Unpacking Matches

1. The scan yields a vector which contains the positions (or offsets) of the matching tuples.
2. These matches are unpacked by their positions before being pushed to the consuming operator.
3. This vector-at-a-time processing is repeated until no more matches are found in the Data Block.

Source: Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

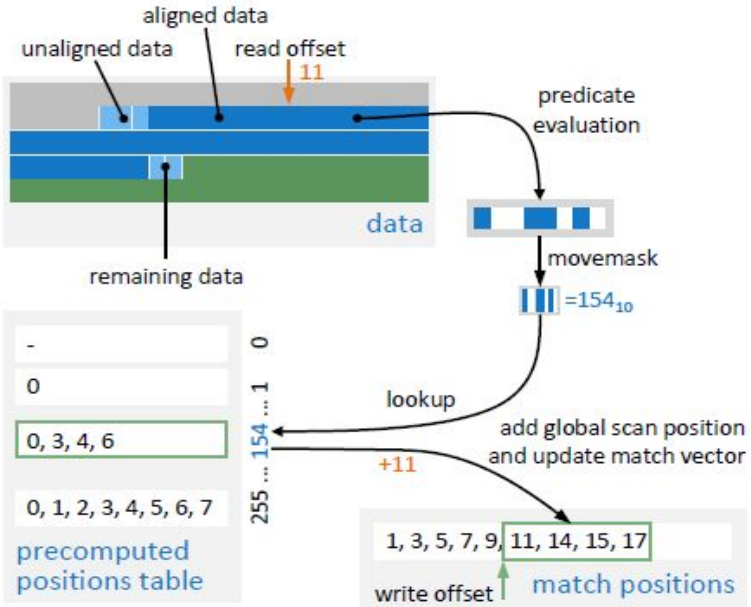
Finding Matches using SIMD Instructions

1. When SIMD instructions are employed, the necessary comparisons are performed on multiple attributes in parallel.

2. When n elements are processed in parallel, then the resulting bit-mask is stored in a SIMD register wherein all the bits of the n individual SIMD lanes are either set to 0 or 1.

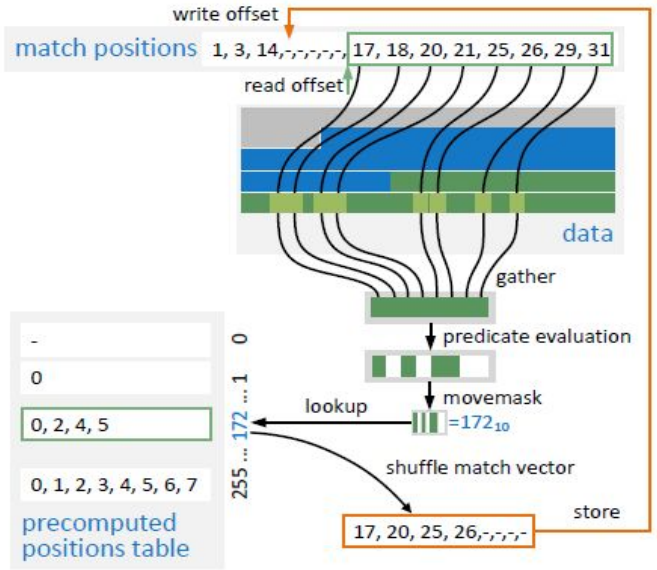
3. Therefore, we make use of a pre-computed table to map the bit-masks to positions whereby a movemask instruction provides the necessary offset.

Source: Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925



Example on how matches are found without restrictions.

Source: Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925



Example on how matches are found with restrictions.

Source: Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

Performance of Vectorized Scans

1. Scan performance highly depends on the selectivity of the preceding restrictions due to non-contiguous memory access pattern
2. For 64-bit integer values, the reduction does not benefit from SIMD instructions. At higher selectivities, the SIMD implementation performs even worse than the scalar code
3. The SIMD optimized algorithms presented here only work for integer data.

Source: Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

3. Boosting data filtering on columnar encoding with SIMD

DaMoN '18: Proceedings of the Tenth International Workshop on **Data Management on New Hardware**;

Authors: Hao Jiang, Aaron J. Elmore

Source: Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

Establishing the Problem Statement

1. In columnar databases, data is generally stored in an encoded format to save storage space and reduce I/O.
2. The common encoding formats include: Dictionary encoding, Delta encoding, Run-length encoding, and Bit-packed encoding
3. In typical open-source columnar data formats, the encoded data needs to be decoded first to memory because queries can be executed on that data.
4. The proposed solution is **SBoost** - A columnar data store which leverages SIMD algorithms to execute filter operators directly on encoded data.

Source: Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

Data Encoding Schemes

1. **Bit-Packed Encoding:** Bit-Packed Encoding stores a number using as few bits as possible.

```
Before:
[--------|--------|--------|--------|--------|--------|--------|--------]
[00000000 00000000 11111101 11101110|00000000 00000000 00000000 10100001]

After:
[11111110 11110111 01000000 00101000|01...... ........ ........ ........]
```

Source: Data Compression: Bit-Packing 101: https://kinematicsoup.com/news/2016/9/6/data-compression-bit-packing-101

2. **Run-length Encoding:** Run-length Encoding encodes a consecutive run of repeating values as a pair *(val, run-length)*. The list [a, a, b, c, c, c, d, d, d, d] will be encoded as [a, 2, b, 1, c, 3, d, 4].

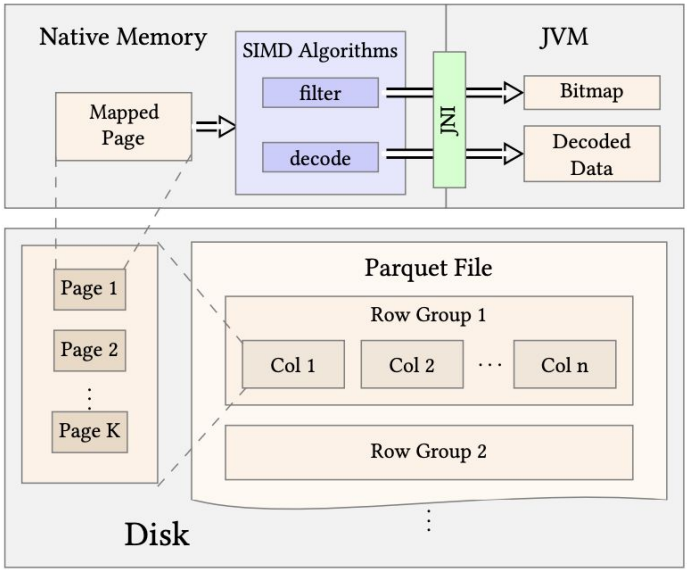
3. **Dictionary Encoding:** Uses a bijective mapping to map column values (which may be of variable length) to integer and bit-packs the result.

4. **Delta Encoding:** Delta Encoding stores the delta between consecutive numbers. Given a list of numbers $[a_0, a_1, a_2, \ldots, a_n]$, delta encoding encode it as a list $[b_0 = a_0, b_1 = a_1 - a_0, b_2 = a_2 - a_1, \ldots, b_n = a_n - a_{n-1}]$.

Source: Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

System Design

1. The Parquet file consists of multiple row groups consisted of fix sizes pages.
2. When a filter or decode operation is to the executed on these pages, they are mapped to a off-heap memory.
3. The corresponding SIMD operations are invoked through the JNI to process all the data items in the page and the result is then passed back to the JVM.



Source: Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

Proposed SIMD Algorithms

1. The authors of this paper have defined **filter** and **decode** APIs for the previously mentioned encoding schemes.
2. The columns which aren't projected in the result are not decoded.
3. Under this, the authors have proposed algorithms for **"equal-to" (=)** and **"less-than" (<)**.
4. The authors believe that other selection predicates could be constructed from these two operators applied in conjunction with logical operators.

Source: Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

Filtering Operation for Bit-Packed Encoded Integers

Equality Operator (=):

Let **X** = a SIMD vector of **n** entries
Let **M** = a mask where the MSB of each entry is 1, the remaining are 0.
Let **A** = a SIMD vector where the scalar to be matched **(a)** is replicated **n** times.
Then,

$$D = X \oplus A$$

$$R = D \mid ((D \ \& \sim M) + \sim M)$$

Returns **R** as the a sparse bitmap of the equality test result and $\mathbf{X_i = a}$, **iff** $\mathbf{(R_i)_{MSB} == 0}$

Source: Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

Example in action

Consider a scalar $a = 011$ against which the equality is checked. Consider $X = X_1X_2$ where $X_1 = 101$ and $X_2 = 011$. Consequently, $n = 2$. Therefore our SIMD vectors are:

$$X = 101011$$

$$M = 100100$$

$$A = 011011$$

Performing the operation mentioned in the previous slide:

$$D = 101011 \oplus 011011 = 110000$$

Computing $R = D \mid ((D \,\&\, \neg M) + \neg M)$,

$$(D \,\&\, \neg M) = 110000 \,\&\, 011011 = 010000$$

$$(D \,\&\, \neg M) + \neg M = 010000 + 011011 = 101011$$

$$R = 110000 \mid 101011 = 111011$$

Proof

The algorithm checks whether $x = a$ by examining if $d = x \oplus a = 0$. Let d_{rb} be the remaining bits in d excluding MSB, $d_{rb} = d \ \& \sim m$, $d \neq 0$ if and only if one of the following is true:

- $d_{msb} = 1$
- $d_{rb} \neq 0 \iff (d_{rb} + \sim m)$ generates a carry to MSB
  $\iff (d_{rb} + \sim m)_{msb} = 1$
  $\iff ((d \ \& \sim m) + \sim m)_{msb} = 1$

Let $r = d \mid ((d \ \& \sim m) + \sim m)$, we see

$$x = a \iff d = 0 \iff r_{msb} = 0$$

Source: Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

Less-Than Operator (<)

1. Follows a similar kind of SIMD algorithm:

$$U = (X \mid M) - (A \,\&\, {\sim}M)$$
$$R = ({\sim} A \,\&\, (X \mid U)) \mid (X \,\&\, U)$$

then return R as a sparse bitmap satisfying

$$X_i < a \iff (R_i)_{msb} = 0$$

Source: Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

Porting these filtering operations to other encoding schemes

Run-length Encoding:

1. The run-length encoded data comprises of consecutive number pairs *(val, run-length)*. We need to execute filter predicates on the *val* fields.
2. For example, when executing predicate x < 200 on a run-length encoded data sequence {105, 2, 339, 4, 242, 1, 132, 8}, the output is {1, 2, 0, 4, 0, 1, 1, 8}.
3. To port this into our existing algorithm, we can set the run-length fields of all the input parameters apart from X to be 0.

Source: Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

Porting these filtering operations to other encoding schemes

Dictionary Encoding:

1. The idea is to rewrite the predicates meant for dictionary encoded data to predicates for bit-packed encoded which can be efficiently processed by the algorithm mentioned above.
2. This is pretty straightforward for the equality operator but the complexity increases for the less-than operator.
3. We need to ensure that for every $a_i > a_j \Rightarrow d(a_i) > do(a_j)$. This presents the need for an **order-preserving dictionary**.

Source: Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

Performance Measure:

1. The data filter algorithm for bit-packed encoded integer and dictionary-bit-packed encoded integer / string can process over **18 billion** numbers per second
2. The algorithm for delta encoded integers and run-length encoded integers also achieves a throughput of over **1 billion** numbers per second.

Source: Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

Future Work

1. Study as to how these vectorized implementations are adapted in modern-day database engines and the challenges associated with it.
2. Study Industry White Papers to observe the adaption of these vectorization techniques
3. Present a holistic overview of how different vectorization techniques are tied together and how they complement each other.

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. Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

4. Hao Jiang and Aaron J. Elmore. 2018. Boosting data filtering on columnar encoding with SIMD. In Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18). Association for Computing Machinery, New York, NY, USA, Article 6, 1–10. https://doi.org/10.1145/3211922.3211932

Thanks