Fighting the Duplicates in Hashing: Conflict Detection-aware Vectorization of Linear Probing

Johannes Pietrzyk[1] Annett Ungethüm[1] Dirk Habich[1] Wolfgang Lehner[1]

Abstract: Hash tables are a core data structure in database systems, because they are fundamental for many database operators like hash-based join and aggregation. In recent years, the efficient vectorized implementation using SIMD (Single Instruction Multiple Data) instructions has attracted a lot of attention. Generally, all hash table implementations need to address what happens when collisions occur. In order to do that, the collisions have to be detected first. There are two types of collisions: (i) key duplicates and (ii) hash value duplicates. The second type is more complicated than the first type. In this paper, we investigate linear probing as a heavily applied hash table implementation and we present an extension of the state-of-the-art vectorized implementation with a hardware-supported duplicate or collision detection. For that, we use novel SIMD instructions which have been introduced with Intel's SIMD instruction set extension AVX-512. As we are going to show, our approach outperforms the state-of-the-art vectorized version for the key handling, but introduces novel challenges for the value handling. We conclude the paper with some ideas how to tackle that challenge.

Keywords: Hashing; Linear Probing; Vectorization; Conflict Detection

1 Introduction

The key objective of database systems is to reliably manage data, where high query throughput and low query latency are still core challenges [Ab16, BFT16, Do13, Ou17]. To satisfy these requirements, database systems constantly adapt to novel hardware features [BKM08]. In the recent past, we have seen numerous hardware advances, in particular with respect to *memory*, *processing elements*, and *interconnects* having a huge impact on the design of database systems [Le17, LUH18, OL18]. For example, with growing capacities of main memory, efficient analytical in-memory data processing becomes viable and is now state-of-the-art [BKM08, Fa17] on the one hand. On the other hand, vectorization is a common approach to improve the processing performance of CPUs by parallelizing computations over vector registers. This vectorization is done using SIMD extensions (SIMD stands for Single Instruction Multiple Data) such as Intel's SSE (Streaming SIMD Extensions) or AVX (Advanced Vector Extensions) and have been available in modern processors for several years. SIMD instructions apply one operation to multiple elements of so-called vector registers at once. Thus, the efficient *vectorized* implementation of database operations using SIMD instructions has attracted a lot of attention in recent years [La16, LB15, PRR15, ZR02].

[1] Technische Universität Dresden, Institut for Systems Architecture, Dresden Database Systems Group, Nöthnitzer Straße 46, 01187 Dresden, firstname.lastname@tu-dresden.de

doi:10.18420/btw2019-04

In the past years, hardware vendors have regularly introduced new SIMD instruction set extensions operating on ever wider registers. For instance, Intel's Advanced Vector Extensions (AVX) operates on vector registers of size 256 bits, while Intel's newest extension set AVX-512 uses now 512-bit vector registers. The wider the vector registers, the more data elements can be stored and processed in one vector. For example, Intel's SSE 128-bit vector register can store four 32-bit data elements, AVX 256-bit vector can store eight ($2x$), and AVX-512 512-bit vector can store 16 ($4x$) of such data elements. Consequently, the SIMD instructions operating on these wider vector registers can also process $2x$ respectively $4x$ the number of data elements in one instruction, which promises significant speedups. In [Ha18], we investigated the influence of the wider vector registers on data compression. As we have shown, the achieved speedups of wider vector registers are sub-optimal in most cases, since the algorithms quickly become memory-bound when computations are accelerated through wider vector registers processing more data elements at once. Thus, an open challenge in this domain is the development of appropriate approaches which exploit the capabilities of newer SIMD extensions to the maximum extent.

To tackle that challenge for lightweight data compression algorithms, we presented a novel hardware-oriented approach in [Un18]. The starting point of this novel approach was, that in addition to an increased vector width of 512-bit, AVX-512 also offers a variety of new instructions. One of the new instruction feature sets is called `Conflict Detection (AVX-512CD)` which allows the vectorization of loops with possible address conflicts. Some key features of `AVX-512CD` are (i) the generation of conflict free subsets, i.e. subsets which contain no equal elements, and (ii) the count of leading zero bits of the elements in a vector. In [Un18], we described the application of these CD instructions for run-length encoding. In particular, we have clearly shown that the CD-based implementation is up to 3.2 times faster for sequences of integers with short run lengths.

Our Contribution: Based on these experiences, we introduce our approach for the application of the `Conflict Detection` instruction to hashing, which is completely different from the data compression domain, in this paper. Generally, hashing is a core primitive for many database operators such as hash-based joins and aggregations [PRR15, RAD15]. The main task of hashing is to distribute entries (key/values) across an array of buckets (hash table). Given a key, the algorithm computes a bucket that suggests where the entry can be found. All hash table implementations need to address what happens when collisions occur. In order to do that, the collisions have to be detected, which sounds like a perfect match to `Conflict Detection`. In particular, we evaluate linear probing as a heavily applied hash table implementation [PRR15, RAD15]. Based on that, our main findings can be summarized as follows:

1. `Conflict Detection` can be used to speedup linear probing, whereby the specific SIMD instructions can be utilized at different positions within the hashing implementation. On the one hand, duplicate keys within one vector register can be detected to reduce unnecessary work. On the other hand, duplicate hash values are extractable within one vector register to reduce expensive `Gather` and `Scatter` operations.

2. However, the application of `Conflict Detection` to hashing comes at a price of difficulty and IO-costly value handling approaches. We will elaborate that aspect in our evaluation in more detail.

Outline of the Paper: The remainder of this paper is organized as follows: In Section 2, we present all essential background information starting by a short description of linear probing followed by a detailed explanation of the state-of-the-art vectorized implementation. This section closes with a short description of new and non-standard vector instructions which have been introduced with AVX-512. Based on that, we introduce our novel vectorized linear probing implementation in Section 3. Then, we present selective results of our exhaustive evaluation on two different hardware systems in Section 4. Finally, we close with related work in Section 5 and a summary including future work in Section 6.

2 Background

Basically, hash tables are a core and a heavily-used data structure in in-memory database systems, because they are required to efficiently execute join and aggregation operations [PRR15]. For example, in a hash join, a hash table of the smaller input relation is built, in which the hash table entries consist of the join attribute as key and the rest as payload [PRR15]. Once the hash table is built, the larger input relation is scanned and join partners are looked up using the hash table. The first phase in this hash join is usually called *build phase*, while the second is called *probe phase*.

In these hash tables, hash functions play an important role [PRR15, RAD15]. Specifically, a hash function is used to map keys to hash table positions allowing to quickly locate the keys in constant time. However, the domain of a hash function (the set of possible keys) is larger than its range (the number of hash table buckets), and so it will map several keys to the same bucket which could result in collisions. That means, all hash table implementations need to address what happens when collisions occur. A common collision strategy is *open addressing*, which allows keys to *leak out* from their preferred bucket and spill over into another bucket [Be18, RAD15].

Based on that, this background section is organized as follows: In Section 2.1, we briefly describe the general idea of linear probing. Then, we introduce the state-of-the-vectorized implementation of linear probing as introduced in [PRR15] in Section 2.2. We close this background section with a description of new and non-standard vector instructions which have been introduced with Intel's latest SIMD extension AVX-512 in Section 2.3.

2.1 Linear Probing

Linear Probing (LP) is the simplest strategy for collision handling in open-addressing. Generally, the hash table structure for open addressing is an array T whose bucket $T[i]$ stores

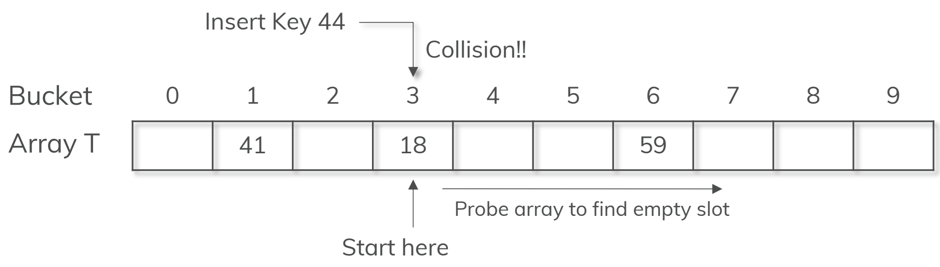


Fig. 1: Illustration of Linear Probing with Collision Example.

a single key as depicted in Fig. 1. Then, an arbitrary hash function h is used to map each key into a bucket of T where the key should be stored. A hash collision occurs when the hash function maps a key into a bucket that is already occupied by a different key. LP resolves this collision by placing the new key into the closest following empty bucket. That means, for a given key x, the buckets of T are examined, beginning with the bucket at position $h(x)$ (where h is the hash function) and continuing to the adjacent buckets $h(x)+1$, $h(x)+2$, ..., until finding either an empty bucket or a bucket whose stored key equals x. An example is given in Fig. 1. Here, key 44 shall be inserted in array bucket 3 leading to a collision. From this bucket, the next free bucket will be used to store the key 44 in linear probing. In our example, the next free bucket would be 4, which is then the storage bucket for this key. This linear scan procedure (probe) is always executed for lookup as well as insertion.

LP has two excellent advantages: (i) low code complexity based on the simplicity of the approach and (ii) very good cache efficiency due to the linear scan [RAD15]. Based on that, we decided to use LP for our case study on applying `Conflict Detection` to hashing.

2.2 State-of-the-Art Vectorized Implementation of Linear Probing

To speed up the overall performance of hash tables, vectorized hash tables use a SIMD register of width n to process multiple keys k_i at once. Based on that idea, Polychroniou et al. [PRR15] presented a vectorized version of LP. We will denote this SIMD LP implementation as basic or state-of-the-art variant, respectively, thereby this approach consists of several phases as illustrated in Fig. 2. The phases are repeated multiple times until all keys are finally processed. The phases are:

Load Phase: In this phase several keys k_i are loaded into a SIMD register v in each iteration. We will denote these keys as active keys. In the first iteration, n keys are transferred from memory to the vector register, whereby n is the size of the vector register. In the next iterations, keys that were successfully inserted into the hash table of the previous iterations are replaced by new keys using a selective load. This selective load exchanges vector lanes by loading contiguous keys from unaligned memory based on a k-bit mask.

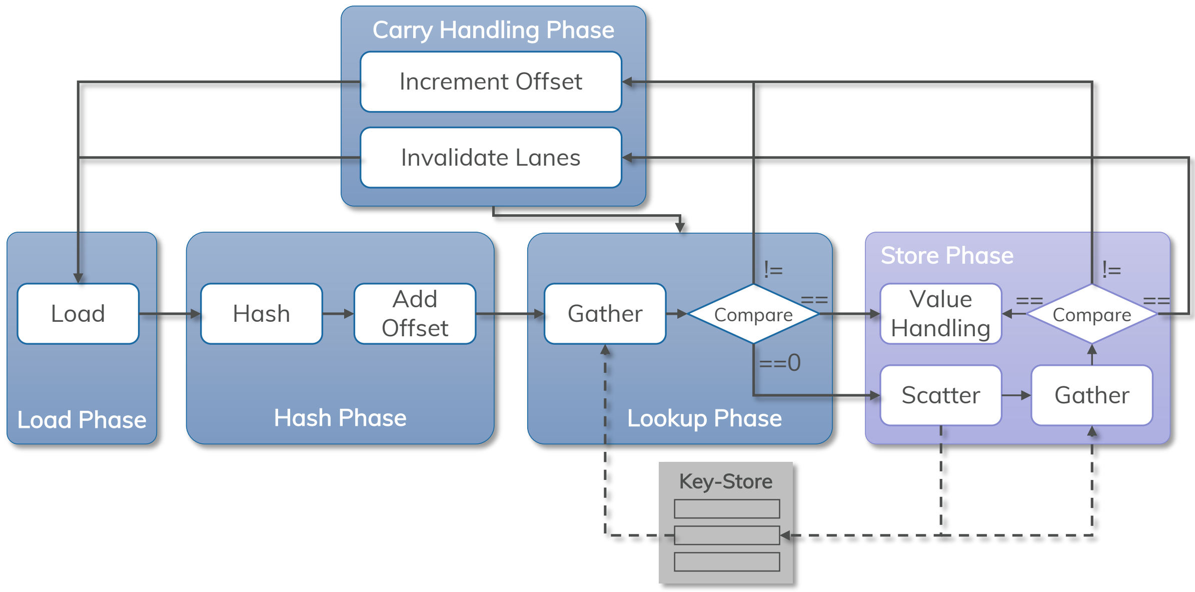


Fig. 2: Control Flow of State-of-the-Art Vectorized Implementation of Linear Probing.

Hash Phase: For each active key k_i within the SIMD register, the hash table bucket is computed using a hash function h and an offset (initial state equals zero). This requires two vector operations.

Lookup Phase: The determined buckets of the keys are used in a `Gather` operation to load the current content of the hash table. The loaded keys are stored in a second SIMD register and compared with the active keys, whereby three results are possible:

(1) Bucket is empty (loaded key is zero): There is currently no key in the hash table bucket stored. Thus, the new key can be stored at that bucket (*Store Phase*) as well as invalidated (*Carry Handling Phase*) and the corresponding value has to be stored.

(2) Bucket contains the same key (Match): The key is already in the corresponding bucket of the hash table. That means, the key can be invalidated (*Carry Handling Phase*) and only the corresponding value has to be stored.

(3) Bucket contains different key (Mismatch): The hash table contains already a different key at that bucket. Thus, the new key can neither be stored nor invalidated and has to remain in the vector register (*Carry Handling Phase*).

Store Phase: Active keys with an identified empty bucket have to be stored using a `Scatter` operation. However, different keys in the vector register can have equal hash values which would lead to a conflict. If different vector lanes are stored to the same hash table bucket, the lane with the highest lane index remains at the specific bucket. To detect that, the stored active keys at the buckets are gathered again and compared with the active keys. If the gathered key equals the scattered key, the key can be invalidated and the value has to be stored (key successfully inserted into the hash table). If the gathered key is not equal to the scattered key, the key has to remain in the vector register (*Carry Handling Phase*).

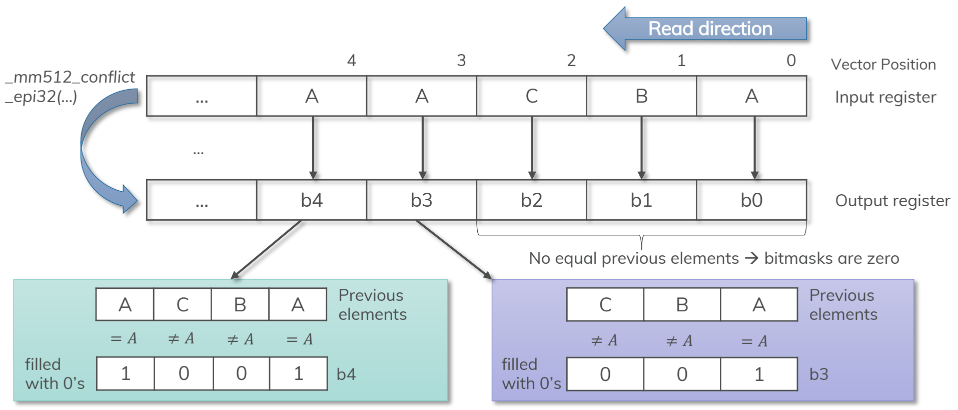


Fig. 3: Example for the *_mm512_conflict_epi32* intrinsic.

Carry Handling Phase: Successfully inserted keys which are marked as invalidated during the whole process can be exchanged in the next iteration. Additionally, the corresponding offsets are set to zero. Lanes which could not be scattered successfully remain in the vector register and corresponding offsets are incremented one by one for the next iteration.

This state-of-the-art vectorization uses standard and well-known SIMD operations like `Scatter`, `Gather`, and compare functions.

2.3 Novel SIMD Instructions

The newest version of Intel's instruction set extension for vectorization is AVX-512. In this extension, the width of vector registers is 512-bit. That means for the state-of-the-art vectorized LP implemention, that 16 keys (each key has a width of 32-bit) can be processed at once in each iteration. Aside from an increased vector width, AVX-512 also offers a variety of new instructions. One of the new instruction feature sets is called *Conflict Detection (AVX-512CD)* allowing the vectorization of loops with possible address conflicts. This instruction feature set is currently supported by Intel Xeon Phi Knights Landing (KNL) as well as on current Xeon processors.

As already presented in [Un18], some core features of AVX-512CD are (i) the generation of conflict free subsets, i.e. subsets which contain no equal elements (no duplicates), and (ii) the count of leading zeros of the elements in a vector. For example, the intrinsic `_mm512_conflict_epi32` creates a vector register containing a conflict free subset of a given source register. An example for this is shown in Fig. 3. In other words and as illustrated in this figure, this intrinsic transforms a vector register with 16 32-bit elements (illustrated by A, B and C) into a new vector register with 16 bitmasks (each represented by 32-bit values). Each bitmask encodes the positions of equal previous elements in the vector. The bitmasks for the first three elements A, B, and C are zero in our example, because there are no equal

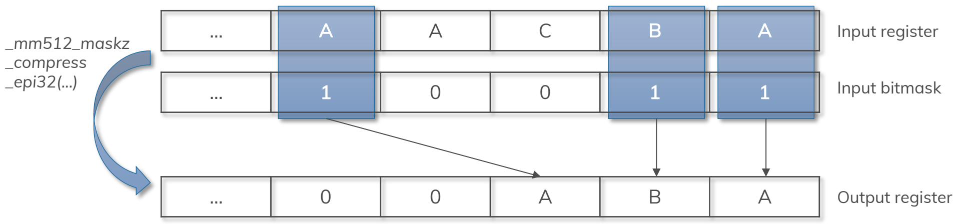


Fig. 4: Example for the *_mm512_maskz_compress_epi32* intrinsic.

previous elements. The *A* element at the third position in the input register is in conflict (equal to) with the element at position 0 in the input register. Thus, the least significant bit of the corresponding bitmask is set to 1, the rest of the bitmask is filled with zeros. The element *A* at position 4 is in conflict with the previous elements at positions 3 and 0 (equal to previous elements). Therefore, the corresponding bits in the bitmask are set to 1, all other bits are zero. A second interesting CD-feature is the intrinsic `_mm512_lzcnt_epi32`, which counts leading zeros. Given a vector of 16 values, this intrinsic counts the number of leading zeros for all values at once and writes the results in a vector register with 16 values.

Another newly introduced functionality is a set of compress instructions, e.g. `_mm512_maskz_compress_epi32`. They are part of the foundation instruction set of AVX512 (AVX-512F). The input of these compress instructions is a vector and a bitmask. Then, the elements in the input vector, which are marked by the bitmask, are stored contiguously in the output vector as depicted in Fig. 4. Using this compress instruction, the result vector contains no reserved space of the unmarked elements in the input register.

3 CD-aware Vectorized Implementation of Linear Probing

The above presented state-of-the-art vectorized implementation of linear probing is well-engineered, but the implementation has a major shortcoming. This shortcoming is related to ever increasing widths of vector registers. On the one hand, with wider vector registers, more keys can be processed simultaneously. For instance, with 128-bit wide vector registers only 4 keys, but now with 512-bit wide vector registers 16 keys are processable simultaneously. On the other hand, with more keys in parallel, the probability of collisions within one vector register increases for two reasons:

1. With more keys in parallel, the probability of duplicate keys within one vector register increases the risk of collisions at the end.
2. With more keys in parallel, the probability that different keys are mapped to a single bucket within one vector register increases the risk of collisions at the end.

Fundamentally, when more hash collisions occur in each iteration, more iterations are needed to process all input data, because more keys have to be moved to the next iteration. At the same time, more iterations also mean that more `Gather` and `Scatter` operations are

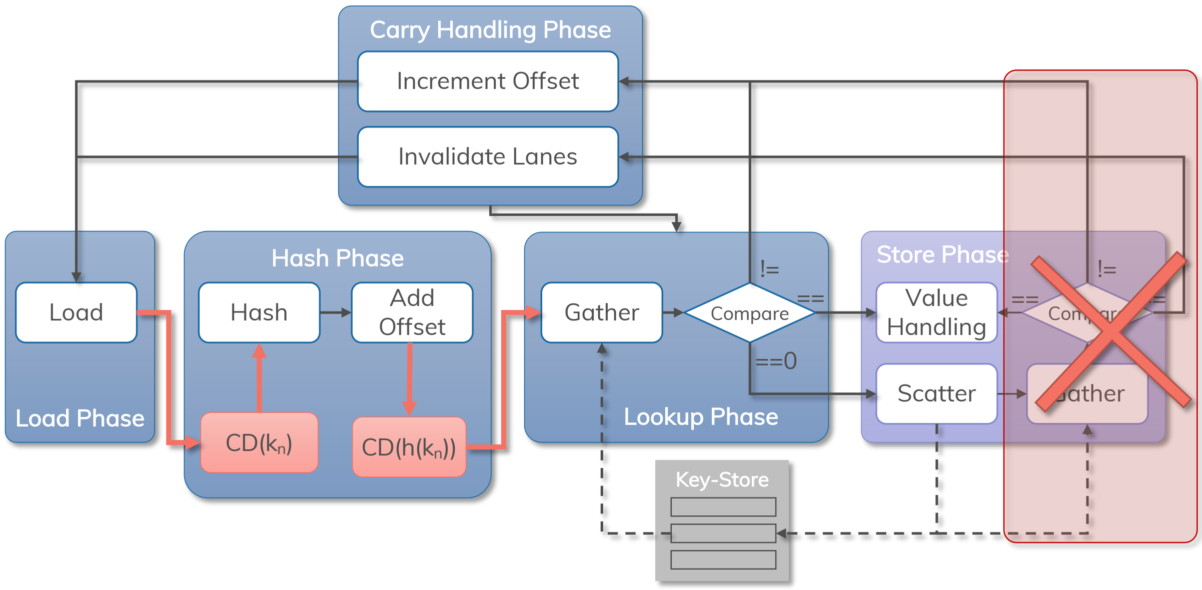


Fig. 5: Control Flow of state-of-the-art Vectorized Linear Probing using Conflict Detection.

performed leading to decreasing performance, finally. Nevertheless, this is a perfect setting for the `Conflict Detection` capability as described next.

3.1 CD-aware Hash Table Data Structures

Before we introduce our `Conflict Detection`-aware (CD-aware) vectorized implementation of linear probing in detail, we describe our underlying data structure. The hash table usually has to hold a set of key/value pairs. The keys are inserted into a so-called *key-store* as illustrated in Fig. 2 consisting of a fixed size number of buckets realized by an array. The corresponding values are stored within a separate memory location (value-store). Since a single key can exist multiple times within the given input dataset, the value-store has to hold every value for the corresponding key. Through the assumption that the total number of occurrences of a single key is unknown in advance, the value-store is realized as a fixed sized array of dynamic containers.

3.2 Handling of Bucket Duplicates

To overcome the mentioned shortcoming, we add two `Conflict Detection` instructions to the `Hash Phase` as illustrated in Fig. 5. With these instructions, the number of non-sequential memory accesses through `Gather` and `Scatter` operations in the subsequent phases are minimized.

The first `Conflict Detection` instruction $CD(k_n)$ is placed directly after the `Load Phase` as highlighted in Fig. 5. If the vector register of an iteration contains duplicate keys, we already know that only one lane has to be used for the further steps. Lanes containing duplicate keys can be automatically invalidated. Since same keys can result in different buckets through offset addition and only the left most lane remain valid, it is feasible to use the compress intrinsic provided by AVX-512F to arrange valid lanes in a contiguous manner. The associated values of duplicate keys have to be preserved until the key can be written to memory. Because the total number of occurrences of a key within a given dataset is not known without further investigations, the values are stored within a temporal dynamic sized buffer. As a consequence, the buffer has to be resized when duplicate keys are detected. When the key is successfully stored into the key-store, the values from the corresponding buffer are appended to the corresponding value-store entry.

After this first `Conflict Detection` instruction, we are sure that the vector register contains only unique keys, whereby already some lanes could be invalidated which limits the exploitation of parallelization in this iteration. However, different keys in the vector register can result in the same buckets within the key-store after the `hash phase`. There are two possible reasons (i) through a pure hash collision or (ii) through the addition of the offset. This situation also has to be detected and solved. For this detection, we use the second `Conflict Detection` instruction $CD(h(k_n))$ as shown in Fig. 5. Based on that, we know the vector lanes with a conflicting position. But the conflicting lanes cannot be invalidated immediately, because each of these different keys could already be in the key-store. Thus, these keys are used in the next `Lookup Phase`. If an empty bucket is found, the key and its assigned value corresponding to the first occurrence (as a result of `Conflict Detection`) of the specific bucket is transferred into the hash table and the lane is invalidated. The remaining lanes containing conflicting buckets reside in the vector register and are treated in the `Carry Handling phase`.

Based on our procedure, the `Store Phase` can be simplified as depicted in Fig. 5. In this `Store Phase`, we do not have to load the buckets again to detect conflicts, because these conflicts are now determined during the `Hash Phase`. In conclusion, the amount of random access IO-operations can be reduced by using the conflict detection intrinsic. Also duplicate keys are substituted within the next iteration resulting in a higher degree of vector lane utilization. Still non-valid lanes are present within a single iteration step which leads to non-optimal data parallelism.

3.3 Handling of Key Duplicates

To address the non-optimal vector lane utilization through conflicting keys mentioned in section 3.2, the `Load phase` has to be adapted as depicted in Fig. 6. Instead of executing one load operation per iteration, contiguous keys are transferred to a vector register (v_0) until the vector contains only distinct elements. We call this approach `FetchD`. To avoid selective and non-cache friendly loads in our `FetchD` approach, a second full buffer vector register (v_b) is loaded at once. Then, distinct keys are identified using a conflict detection. This buffer is then used to fill up invalidated lanes from the register containing the keys for further

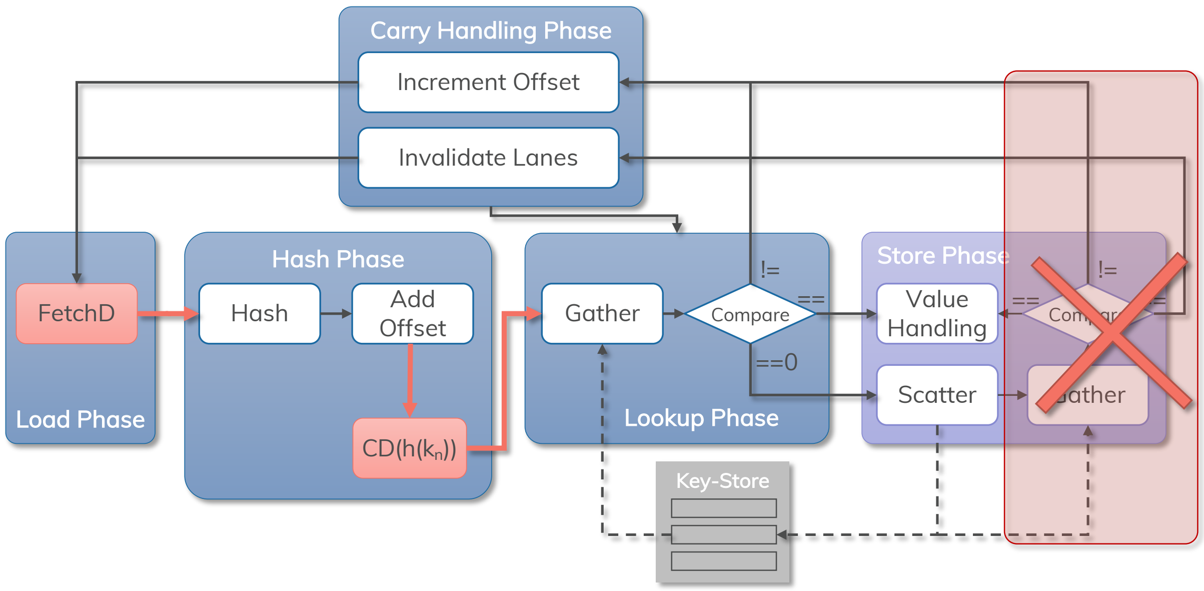


Fig. 6: Control Flow of the CD-aware Vectorized Implementation of Linear Probing.

processing. This is done using `_mm512_maskz_compress_epi32` to organize distinct keys of vector v_b in a contiguous manner. In a second step, invalidated lanes of v_0 are substituted with elements from the v_b using `_mm512_mask_expand_epi32`. These two steps are repeated until v_0 contains only distinct elements or all elements are processed. In regard to the values, a similar approach as described in the section above is used. Nevertheless, the buffer vector register needs its own temporal value buffer which is merged with the value buffer from v_0 corresponding to the replaced key lanes. This leads to additional IO-operations during the *Load Phase*.

In summary, the discussed concepts can be used to utilize vector registers with the maximum amount of parallel computation on the one hand. On the other hand, organizing the values require scalar IO-operations which can be considered to be expensive in comparison to vectorized operations.

4 Experimental Evaluation

Before we summarize the core results of our exhaustive evaluation in Sections 4.1 and 4.2, we briefly describe our overall evaluation setup. Basically, we used two different hardware platforms as depicted in Tab. 1. The first platform is a Xeon Phi™ 7250 Knights Landing (KNL), while the second is a Xeon® Gold 6130 (SKL). Both platforms provide the AVX-512 vector register extension as well as the special instruction set AVX-512CD. The cache sizes of the KNL and SKL are the same for L1 and L2. While every core of the SKL has access to a 22 MB dimensioned L3-Cache, the KNL has no L3-Cache. Instead, a high bandwidth memory which is located on the chip can be used in the KNL.

Name	**Xeon Phi™**	**Xeon® Gold**
Prozessor Model	7250	6130
Base Clock Frequency	1.4 GHz	2.1 GHz
Nodes x Cores x Threads	4 x 17 x 4	4 x 16 x 2
L1 Size	32 KB	32 KB
L2 Size	1 MB	1 MB
L3 Size	-	22 MB
AVX-512	F, PF, ER, CD	F, DQ, CD, BW, VL

Tab. 1: Hardware Platform Specifications.

Furthermore, all vectorized implementations of linear probing were done in C/C++ by ourselves, thereby we distinguish between the following implementations:

Basic: State-of-the-art vectorized implementation of linear probing as introduced in [PRR15] (see also Section 2.2).

CDHashProbe: This is our first proposed CD-aware vectorized implementation with two CD instructions in the *Hash Phase* and only a `Scatter` operation in the *Store Phase* (see Fig. 5).

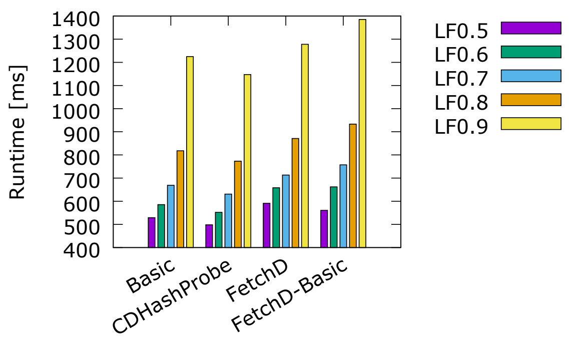
FetchD: This is our second proposed CD-aware vectorized implementation with the `FetchD` approach in the *Load Phase* and a single CD instruction in the *Hash Phase* (see Fig. 6).

FetchD-Basic: This is an enhanced state-of-the-art vectorized implementation using our `FetchD` instead of a selective load in the *Load Phase* including a `Scatter` and `Gather` operation in the *Store Phase*.

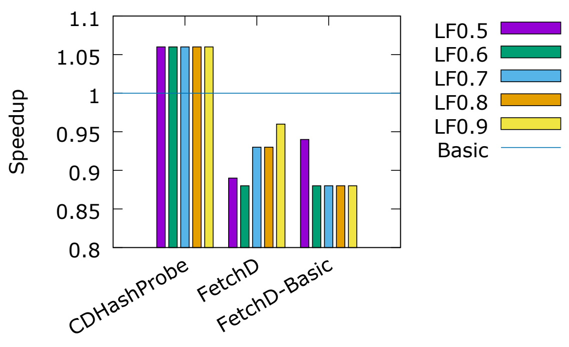
In all implementations and experiments, we used the vectorized version of `Mumur3` as our main hash function. Since in our work we focus on pushing the achieved data parallelism for the hash build phase through vectorization to a maximum extent, we ran all experiments single threaded. For this, we compiled all implementations with gcc (KNL: version 7.0.1, SKL: version 7.2.0) with the optimization flags `-Ofast -mavx512f -mavx512cd`. We also evaluated the novel `Compress` instruction of AVX-512. Unfortunately, the impact was very marginal and therefore, we do not include this aspect in our evaluation.

4.1 Evaluation Result for Hashing without Value Handling

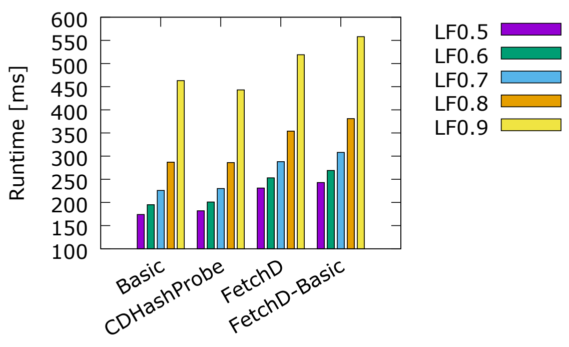
In our first series of experiments, we investigated linear probing without value handling which can be used for `aggregation`, `anti-join` and `exists` operators in in-memory database systems. As clearly mentioned previously, there are two possible types of collisions: (i) bucket duplicates and (ii) key duplicates. In the following, we separately evaluate both types.



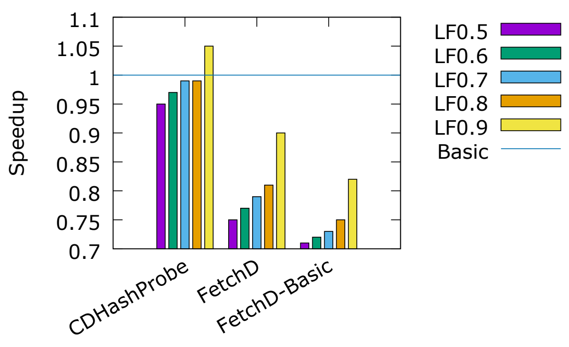
(a) KNL Runtimes



(b) KNL Speedups



(c) SKL Runtimes

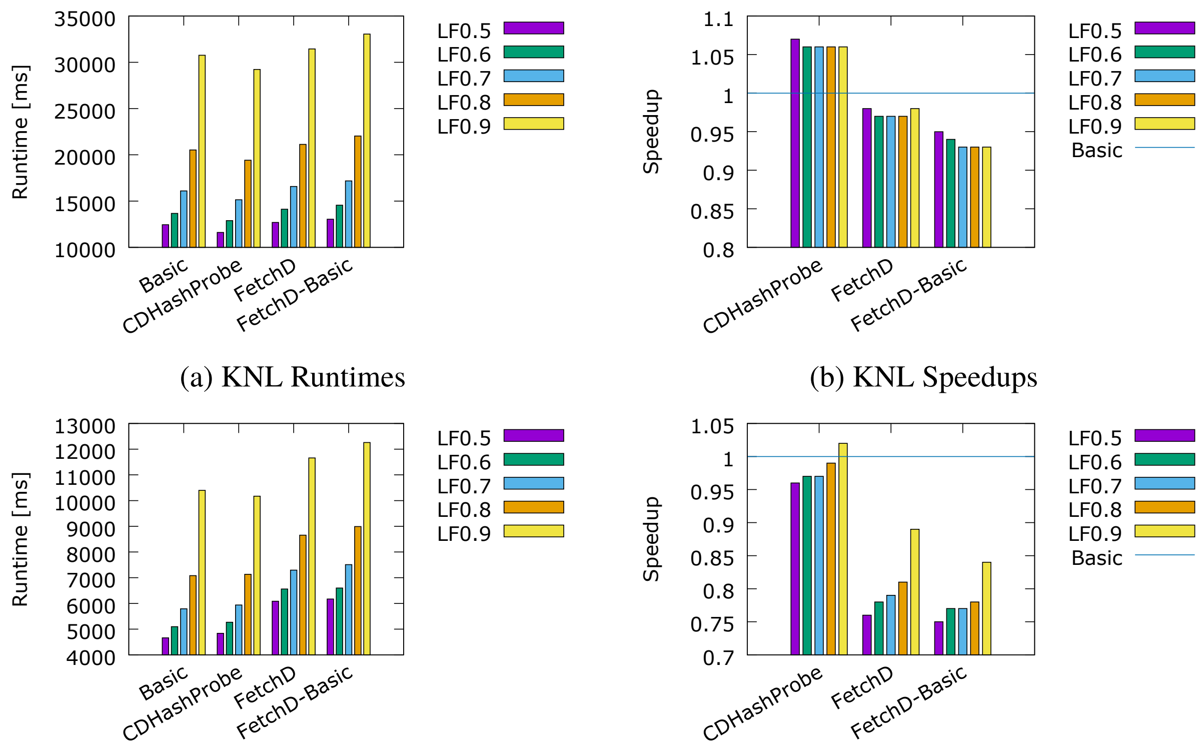


(d) SKL Speedups

Fig. 7: Runtime Results and Speedups for a Key-Store with 1024KB Size and Input Data Consisting of Unique Keys.

4.1.1 Bucket Duplicates

To evaluate the influence of bucket duplicates on the runtime behaviour of the different linear probing implementation, we generated various data sets containing different numbers of unique keys and varied the load factor of the key store from 0.5 to 0.9 in increments of 0.1. With unique keys, we explicitly restrict ourselves to bucket duplicates and we would expect, that our first CD-aware implementation *CDHashProbe* (see Fig. 5) outperforms *Basic*, while our second CD-aware implementation *FetchD* offers too much overhead in this case. Generally, with higher load factors of the key-store (hashmap), the number of bucket duplicates increases leading to higer runtimes. Fig. 7(a) and (c) show runtimes for KNL as well as SKL on a data set size consisting of unique keys and a key-store size of 1024KB, so that the key-store fits in the L2-cache on both hardware platforms. As we can see, the runtimes for each implementation increases with increasing load factors and SKL is faster than KNL as expected. Fig. 7(b) and (d) depict the speedups of our approaches compared to the *Basic* implementation. On KNL, our *CDHashProbe* approach is slightly faster than *Basic* in all cases. In contrast to that, our *CDHashProbe* only outperforms the *Basic* approach on SKL for high load factors. The reason for that is that the `CD` instruction is an expensive operation and this is only beneficial when the effort is less than the additional `Gather` operation in *Basic*. Moreover, *FetchD* and *FetchD-Basic* are slower than the *Basic*



(a) KNL Runtimes

(b) KNL Speedups

(c) SKL Runtimes

(d) SKL Speedups

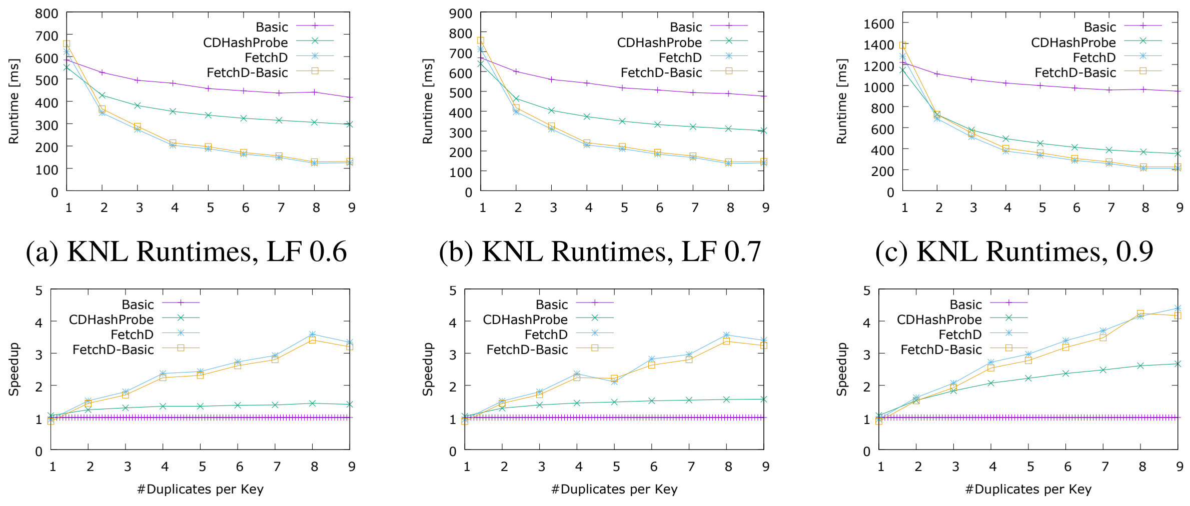
Fig. 8: Runtime Results and Speedups for a Key-Store with 16MB Size and Input Data Consisting of Unique Keys.

implementation in all cases. Of course, since we do not have key duplicates, the treatment of this introduces additional overhead.

Fig. 8 shows the results for data sets with unique keys, a key-store size of 16MB and varying load factors. As we can see, the same observations are also visible for higher amount of data as well. From this set of experiments, we can conclude that our CD-aware implementation for bucket duplicates *CDHashProbe* slightly outperforms the *Basic* approach.

4.1.2 Key Duplicates

In order to evaluate the influence of key duplicates on the runtime behaviour of the different linear probing implementation, we generated various data sets containing different numbers of repeating keys in sequence. Furthermore, we also varied the load factor of the key-store from 0.5 to 0.9 in increments of 0.1. With repeating keys, we explicitly investigate the influence of key duplicates in a best case scenario and we would expect, that our second CD-aware implementation *FetchD* (see Fig. 6) outperforms the other implementations. While Fig. 9 shows the results for KNL, Fig. 10 depicts the results for SKL. In both cases, we



(a) KNL Runtimes, LF 0.6 (b) KNL Runtimes, LF 0.7 (c) KNL Runtimes, 0.9

(d) KNL Speedups, LF 0.6 (e) KNL Speedups, LF 0.7 (c) KNL Speedups, LF 0.9

Fig. 9: KNL-Results for Key-Store of Size 1024KB and Varying Number of Repeating Keys.

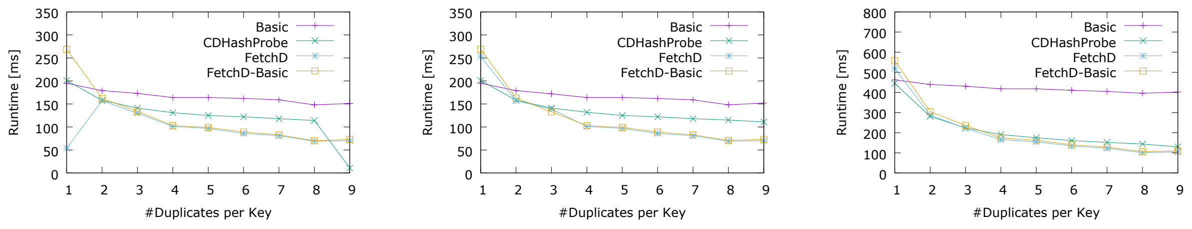
illustrate the runtimes for load factors of 0.6, 0.7, and 0.9 as well as the speedups compared to the *Basic* implementation.

As we can see, the presented implementation improvements benefit from a higher amount of duplicates within the processed data, while *FetchD* has the highest impact on the performance. With increasing load factors, the speedup of our CD-aware implementations also increases compared to the *Basic* implementation. For example, we can improve the performance for data with a high number of repetitive keys up to a factor of 18 (load factor 0.9; repetition sequence length of 100) on the KNL and up to factor a factor of 10 on the SKL. Moreover, we observed better runtimes of all investigated scenarios (data size, number of duplicate key, load factors) when the underlying key-store is quite small, so it can fit into small levels of cache, but this is already well-known.

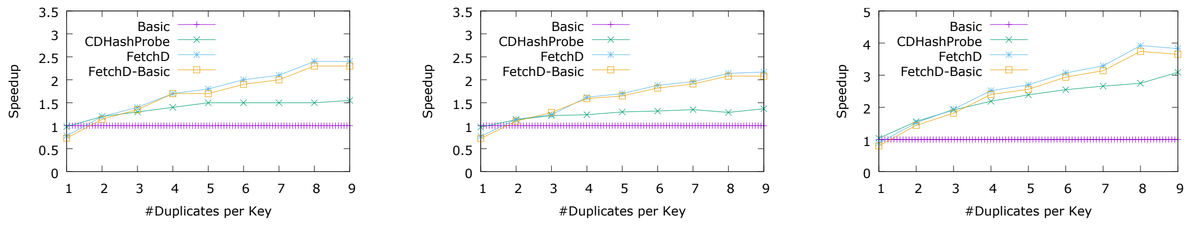
4.1.3 Intermin Conclusion

As already shown by [PRR15], the load factor should not exceed 0.6 with regard to memory consumption and total execution time. In our previous presented evaluation results, we always included this specific load factor for a key-store size of 1MB in our considerations, so that the key-store perfectly fits in the L2-Cache of our hardware platforms. From these evaluations, we are able to conclude the following two aspects:

1. If the input data only consists of unique keys, our first CD-aware implementation *CDHashProbe* performs slightly better than the *Basic* implementation.
2. However, already with a small amount of duplicate keys in the input data, our second CD-aware improvement pays more off.



(a) SKL Runtimes, LF 0.6 (b) SKL Runtimes, LF 0.7 (c) SKL Runtimes, LF 0.9



(d) SKL Speedups, LF 0.6 (e) SKL Speedups, LF 0.7 (c) SKL Speedups, LF 0.9

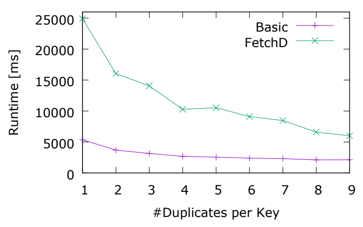
Fig. 10: SKL-Results for Key-Store of Size 1024KB and Varying Number of Repeating Keys.

Thus, our experiments show that using new instructions like `Conflict Detection` instead of random access IO operations can improve the total performance if actual conflicts occur. Furthermore, the presented approach *FetchD* which needs additional instructions and branches amortizes as soon as duplicate keys are within the range of a vector register through a cache friendly access pattern and a high degree of data parallelism.

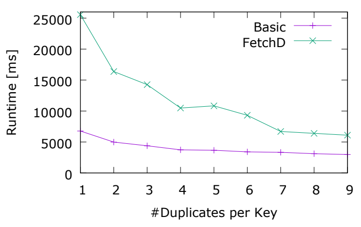
To conclude, the presented CD-aware optimization's for linear probing can improve the total performance if the processed data contains duplicate keys or duplicate buckets. The influence of `Conflict Detection` as well as *FetchD* grows with higher load factors and the amount of duplicates. This arises from the fact that without value handling, repeating keys within vector registers are redundant and can be discarded. However if values have to be treated, the values of repetitive keys have to be handled.

4.2 Evaluation Results for Hashing with Value Handling

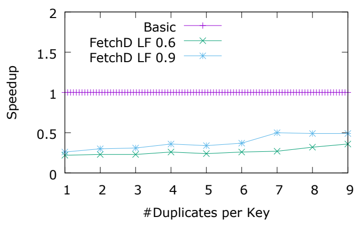
To precisely evaluate our proposed value handling approach for the CD-aware implementations, we repeated all our previously introduced experiments with enabled value handling. In particular, the value handling is important in order to execute hash joins. Fig. 11 shows the results on the KNL as well as on the SKL hardware platform. In the depicted experiment, we used a key-store of size 1MB, thereby we only compare the *Basic* with the *FetchD* implementation. Through the need of dynamic temporal buffers, a growing amount of memory re-allocations and copying for all our CD-aware implementations for value handling, the performance of our *FetchD* is lower then the *Basic* one. As we can see, the negative impact on the performance gets slightly better with a growing number of duplicates and a higher load factor. Nevertheless, a more sophisticated value handling approach for *FetchD* has to be found to be competitive or even better than the *Basic* implementation.



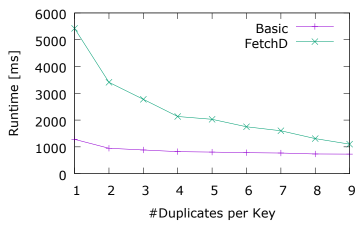
(d) KNL Runtimes, LF 0.6



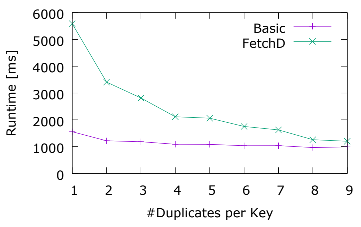
(e) KNL Runtimes, LF 0.9



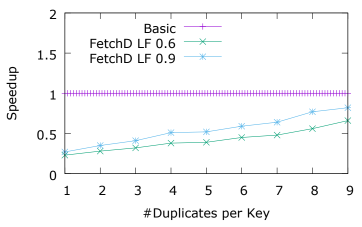
(f) KNL Speedups



(d) SKL Runtimes, LF 0.6



(e) SKL Runtimes, LF 0.9



(f) SKL Speedups

Fig. 11: Value Handling Evaluation Results with a Key-Store of Size 1MB.

5 Related Work

Fundamentally, related work in this domain is manifold, because the efficient utilization of SIMD (Single Instruction Multiple Data) instructions in database systems is a very active research field [Ha18, La16, LB15, PRR15, SWL11, Un18, ZR02].

For example, SIMD instructions are frequently applied in lightweight data compression algorithms [Da17, LB15]. In this specific domain, null suppression (NS) is the most studied lightweight compression approach, whereby the basic idea is the omission of leading zeros in the bit representation of integers [LB15, SGL10]. There are different techniques addressing the efficient implementation using SIMD instructions [Da17, LB15, SGL10]. However, most of the vectorized implementations of lightweight data compression algorithms have been developed for a fixed vector width of 128 bits (corresponding to Intel's SIMD extension SSE). In [Ha18], we systematically investigated the impact of different SIMD instruction set extensions with vector sizes of 128-, 256-, and 512-bits on the behavior of lightweight data compression algorithms. To obtain implementations for wider vector sizes (AVX2 and AVX-512), the 128-bit implementation can be used as foundation. In a straightforward transformation, the 128-bit SIMD operations can be substituted by the corresponding operations for 256 or 512-bit vectors. As we have shown, this is possible in almost all cases, since many instructions offered by SSE are also offered by AVX2 and AVX-512 on wider vectors. Fundamentally, two effects are observable: (i) NS algorithms working on wider vector registers are more vulnerable to outliers in the data, which can affect both, the compression ratio as well as the performance negatively and (ii) the speed ups are generally sub-optimal in most cases, since the algorithms quickly become memory-bound when the computations are accelerated through wider vector registers processing more data elements at once. To overcome that, novel approaches are necessary. In [Un18], we presented a novel approach for RLE encoding using `Conflict Detection`. Aside from our work, [La18] introduced efficient refill algorithms for vector registers by using the latest SIMD instruction set, AVX-512. On the other hand, SIMD instructions are also used in other database operations like scans [LP13], aggregations [ZR02], hashing [PRR15, RAD15] or joins [Ba13]. To best of our knowledge, none of these approaches uses AVX-512 CD, although the operations could benefit from CD.

From a hashing perspective, the papers [PRR15], [RAD15] and [Be18] are highly relevant. The state-of-the-art vectorized implementation of linear probing is presented in [PRR15] as described in Section 2. Richter et al. [RAD15] exhaustively studied a variety of common hash table implementations—including linear probing—in a five-dimensional requirements space: (i) data-distribution, (ii) load factor, (iii) dataset size, (iv) read/write-ratio, and (v) un/successful-ratio. As they have shown, there exists no single best-performing hash table implementation and each hash table implementation has its own application area. In [Be18], the authors translated the state-of-the-art vectorized implementation of linear probing to OpenCL with the aim to reduce code complexity and to ensure portability. For that, they realized essential primitives like Gather, Scatter, Selective Load and Selective Store in OpenCL. It would be interesting to see how the translation of the `Conflict Detection` would look like.

6 Conclusion and Future Work

Hash tables are a core data structure in in-memory database systems, because they are fundamental for many database operators like hash-based join and aggregation. In recent years, the efficient vectorized implementation using SIMD (Single Instruction Multiple Data) instructions has attracted a lot of attention. Generally, all hash table implementations need to address what happens when collisions occur. In order to do that, the collisions have to be detected first. There are two types of collisions: (i) key duplicates and (ii) hash value duplicates (hash collisions). The second type is more complicated than the first type. In this paper, we investigated linear probing as a heavily applied hash table implementation and we presented an extension of the state-of-the-art vectorized implementation with a hardware-supported duplicate or collision detection. For that, we use novel SIMD instructions which have been introduced with Intel's SIMD instruction set extension AVX-512. As we have shown, our approach outperforms the state-of-the-art vectorized version for the key handling, but introduces novel challenges for the value handling.

Further research should investigate different methods of value handling. The usage of dynamic sized buffers should be replaced through a fixed sized buffer. Based on that, costly memory reallocations and copy operations can be reduced as well as handling the values could be done using SIMD scatter instructions. One opportunity for that would be to process the given dataset twice, collecting statistics for the dataset within the first run. Then, this information can be used in a further step to allocate a constant sized value store which can hold up all values having to be inserted. As a side effect of this approach, the result of the first run can be used further e.g., for database operators like aggregation.

References

[Ab16] Abadi, Daniel; Agrawal, Rakesh; Ailamaki, Anastasia; Balazinska, Magdalena; Bernstein, Philip A.; Carey, Michael J.; Chaudhuri, Surajit; Dean, Jeffrey; Doan, AnHai; Franklin, Michael J.; Gehrke, Johannes; Haas, Laura M.; Halevy, Alon Y.; Hellerstein, Joseph M.; Ioannidis, Yannis E.; Jagadish, H. V.; Kossmann, Donald; Madden, Samuel; Mehrotra, Sharad; Milo, Tova; Naughton, Jeffrey F.; Ramakrishnan, Raghu; Markl, Volker; Olston, Christopher; Ooi, Beng Chin; Ré, Christopher; Suciu, Dan; Stonebraker, Michael; Walter, Todd; Widom, Jennifer: The Beckman report on database research. Commun. ACM, 59(2):92–99, 2016.

[Ba13] Balkesen, Cagri; Alonso, Gustavo; Teubner, Jens; Özsu, M. Tamer: Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. PVLDB, 7(1):85–96, 2013.

[Be18] Behrens, Tobias; Rosenfeld, Viktor; Traub, Jonas; Breß, Sebastian; Markl, Volker: Efficient SIMD Vectorization for Hashing in OpenCL. In: Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018. pp. 489–492, 2018.

[BFT16] Breß, Sebastian; Funke, Henning; Teubner, Jens: Robust Query Processing in Co-Processor-accelerated Databases. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. pp. 1891–1906, 2016.

[BKM08] Boncz, Peter A.; Kersten, Martin L.; Manegold, Stefan: Breaking the memory wall in MonetDB. Commun. ACM, 51(12):77–85, 2008.

[Da17] Damme, Patrick; Habich, Dirk; Hildebrandt, Juliana; Lehner, Wolfgang: Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In: Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017. pp. 72–83, 2017.

[Do13] Do, Jaeyoung; Kee, Yang-Suk; Patel, Jignesh M.; Park, Chanik; Park, Kwanghyun; DeWitt, David J.: Query processing on smart SSDs: opportunities and challenges. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013. pp. 1221–1230, 2013.

[Fa17] Faerber, Franz; Kemper, Alfons; Larson, Per-Åke; Levandoski, Justin J.; Neumann, Thomas; Pavlo, Andrew: Main Memory Database Systems. Foundations and Trends in Databases, 8(1-2):1–130, 2017.

[Ha18] Habich, Dirk; Damme, Patrick; Ungethüm, Annett; Lehner, Wolfgang: Make Larger Vector Register Sizes New Challenges?: Lessons Learned from the Area of Vectorized Lightweight Compression Algorithms. In: Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018. pp. 8:1–8:6, 2018.

[La16] Lang, Harald; Mühlbauer, Tobias; Funke, Florian; Boncz, Peter A.; Neumann, Thomas; Kemper, Alfons: 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 Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016. pp. 311–326, 2016.

[La18] Lang, Harald; Kipf, Andreas; Passing, Linnea; Boncz, Peter A.; Neumann, Thomas; Kemper, Alfons: Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. In: Proceedings of the 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, June 11, 2018. pp. 5:1–5:8, 2018.

[LB15] Lemire, Daniel; Boytsov, Leonid: Decoding billions of integers per second through vectorization. Softw., Pract. Exper., 45(1):1–29, 2015.

[Le17] Lehner, Wolfgang: The Data Center under your Desk - How Disruptive is Modern Hardware for DB System Design? PVLDB, 10(12):2018–2019, 2017.

[LP13] Li, Yinan; Patel, Jignesh M.: BitWeaving: fast scans for main memory data processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013. pp. 289–300, 2013.

[LUH18] Lehner, Wolfgang; Ungethüm, Annett; Habich, Dirk: Diversity of Processing Units - An Attempt to Classify the Plethora of Modern Processing Units. Datenbank-Spektrum, 18(1):57–62, 2018.

[OL18] Oukid, Ismail; Lersch, Lucas: On the Diversity of Memory and Storage Technologies. Datenbank-Spektrum, 18(2):121–127, 2018.

[Ou17] Oukid, Ismail; Booss, Daniel; Lespinasse, Adrien; Lehner, Wolfgang; Willhalm, Thomas; Gomes, Grégoire: Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems. PVLDB, 10(11):1166–1177, 2017.

[PRR15] Polychroniou, Orestis; Raghavan, Arun; Ross, Kenneth A.: Rethinking SIMD Vectorization for In-Memory Databases. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015. pp. 1493–1508, 2015.

[RAD15] Richter, Stefan; Alvarez, Victor; Dittrich, Jens: A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. PVLDB, 9(3):96–107, 2015.

[SGL10] Schlegel, Benjamin; Gemulla, Rainer; Lehner, Wolfgang: Fast integer compression using SIMD instructions. In: Proceedings of the Sixth International Workshop on Data Management on New Hardware, DaMoN 2010, Indianapolis, IN, USA, June 7, 2010. pp. 34–40, 2010.

[SWL11] Schlegel, Benjamin; Willhalm, Thomas; Lehner, Wolfgang: Fast Sorted-Set Intersection using SIMD Instructions. In: International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2011, Seattle, WA, USA, September 2, 2011. pp. 1–8, 2011.

[Un18] Ungethüm, Annett; Pietrzyk, Johannes; Damme, Patrick; Habich, Dirk; Lehner, Wolfgang: Conflict Detection-Based Run-Length Encoding - AVX-512 CD Instruction Set in Action. In: 34th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2018, Paris, France, April 16-20, 2018. pp. 96–101, 2018.

[ZR02] Zhou, Jingren; Ross, Kenneth A.: Implementing database operations using SIMD instructions. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002. pp. 145–156, 2002.