An Efficient Vectorized Hash Table for Batch Computations

Hesam Shahrokhi
University of Edinburgh, UK

Amir Shaikhha
University of Edinburgh, UK

Abstract

In recent years, the increasing demand for high-performance analytics on big data has led the research on batch hash tables. It is shown that this type of hash table can benefit from the cache locality and multi-threading more than ordinary hash tables. Moreover, the batch design for hash tables is amenable to using advanced features of modern processors such as prefetching and SIMD vectorization. While state-of-the-art research and open-source projects on batch hash tables made efforts to propose improved designs by better usage of mentioned hardware features, their approaches still do not fully exploit the existing opportunities for performance improvements. Furthermore, there is a gap for a high-level batch API of such hash tables for wider adoption of these high-performance data structures. In this paper, we present Vec-HT, a parallel, SIMD-vectorized, and prefetching-enabled hash table for fast batch processing. To allow developers to fully take advantage of its performance, we recommend a high-level batch API design. Our experimental results show the superiority and competitiveness of this approach in comparison with the alternative implementations and state-of-the-art for the data-intensive workloads of relational join processing, set operations, and sparse vector processing.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Computer systems organization → Single instruction, multiple data

Keywords and phrases Hash tables, Vectorization, Parallelization, Prefetching

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.27

Acknowledgements The authors would like to thank Huawei for their support of the distributed data management and processing laboratory at the University of Edinburgh.

1 Introduction

Hash tables are one of the most important data structures in programming. They are widely used in high-performance analytics workloads including database query processing, sparse linear algebra, graph processing, and computer networks. Besides the great efforts in algorithmic improvement of hash tables [10, 18, 20], the recent advances in modern processors, further motivated the research on high-performance hash tables that leverage the hardware characteristics including parallelization, prefetching, and vectorization.

Previous research [19] has shown that batch operations (e.g., batch lookups) on a hash table result in higher performance in comparison with ordinary scalar-parameter operations. This is because of the improved cache locality and the freedom given to the hash table designer for hand-tuning the code, which is not available when dealing with ordinary scalar-parameter operations over hash tables. Thus, hash tables with batch operations have gained more attention; both research [12, 14, 19, 23, 22, 32] and open-source projects [1, 3, 6] have proposed hash table designs with the support for batch lookups, insertions, and deletions.

Table 1 A summary of state-of-the-art batch lookups in hash tables. H: Horizontal Vectorization.

Approach	Parallelization	Prefetching	SIMD Vectorization
Hirola [3], R hashmap [6]	○	○	○
DPDK [1]	●	●	◐ (**H**)
Cuckoo++ [23]	○	●	◐ (**H**)
Polychroniou et al. [19]	●	○	●
Vec-HT (this paper)	●	●	●

The current literature on batch hash tables considers the following hardware features:

- Parallelization: The batch of inputs is divided into separate partitions that are processed in parallel [1, 19].
- Prefetching: The memory prefetching feature of processors is used to hide the latency of frequent memory accesses for a group of elements [1, 2, 9, 23].
- Vectorization: The Single Instruction, Multiple Data (SIMD) instructions of the processing unit are used for faster processing of a vector of inputs [1, 2, 9, 19, 23].

Although the existing approaches made noticeable efforts on improving the performance of batch hash tables, none of them fully exploits all of the three optimization dimensions mentioned above (cf. Section 2). The existing approaches for vectorization can be categorized into two classes (cf. Figure 1): (1) *Horizontal Vectorization*, where the SIMD instructions are used for the operations on a single input over multiple hash table entries [1, 9, 23], and (2) *Vertical Vectorization*, where the SIMD instructions are used for the operations on an input batch over single hash table entries [19]. The first approach is not inherently batch based; it can be applied to ordinary hash tables. By conducting intensive benchmarks, Polychroniou et al. [19] and Shankar et al. [30] have shown that vertical vectorization is faster than the horizontal approach. However, prefetching has only been considered for horizontal vectorization [1, 23], and the only existing implementation of the vertically vectorized approach [19] does not support prefetching.

This paper makes the following contributions:

- We present Vec-HT, the first batch hash table that is fully optimized in all three dimensions; Vec-HT is a multi-threaded, vertically-vectorized, and prefetching-enabled hash table that can be used for high-performance data analytics. We explain the architecture and the high-level batch API of Vec-HT in Section 3.
- Previous research has shown that in most high-performance use cases, optimizing the lookup performance is more important than the other operations (Section 2). Thus, the focus of this research is on batch lookups. We show the design decisions, the applicable optimizations, and the way we combine them for efficient batch lookups in Section 4.
- We consider several data analytics tasks that can benefit from batch hash tables, such as hash-join in query processing, set processing, and sparse vector operations (Section 5).
- We present the implementation challenges we faced (e.g., memory management and parallel iteration) and how we addressed them in Section 6.
- We experimentally evaluate (Section 7) the performance of our hash table on a set of micro benchmarks across different use case domains. Our results show that Vec-HT outperforms the state-of-the-art batch and non-batch hash tables.

2 Background and Related Work

In this section, we introduce the main concepts and techniques for building high-performance hash tables while summarizing the previous efforts in this area of research. Table 1 presents a summary of the state-of-the-art in batch hash tables.[1] To position the contributions of this paper, our approach is also appended to the table.

Batch Hash Tables. Batch Hash Tables accept a vector of keys or key/value pairs as their API arguments and do the operation on the inputs in batch. By taking batch inputs, the batch hash tables benefit from the cache locality and are also amenable to the use of Single Instruction, Multiple Data (SIMD), parallelism, and prefetching. The SIMD is a hardware feature that allows the simultaneous execution of an operation on a vector of values. On the other hand, prefetching is a hardware feature that allows the program to request future memory accesses in advance and asynchronous to the other computations. We will cover the more-detailed definitions of these two concepts later in this section.

Among the open-source projects [1, 3, 6], Hirola [3] presents a fast batch hash table written in C which is an alternative for `dict/set` in Python. Similarly, by considering the fact that most values in the R language are vectors and matrices, R-hashmap[6] presents a batch hash table for R, which is built over some existing ordinary hash tables in C. These two libraries are only using the cache-locality of the batch input as a way of performance improvement. As a more advanced open-source project, DPDK [1] offers a specific-purpose batch hash table for networking use cases. It is both SIMD-aware and prefetch-enabled.

There is also state-of-the-art research [9, 12, 14, 19, 22, 23, 32] on batch hash tables. Some of the approaches [12, 14, 32] are only using batch processing to benefit from the cache locality and prefetching while others [9, 19, 22, 23] use SIMD techniques on a vector of inputs. Similar to DPDK [1], Horton [9] and Cuckoo++[23] have focused on improving the performance of batch hash tables by applying SIMD and prefetching techniques to a specific type of SIMD-aware batch hash table designs called Bucketized Cuckoo Hash Tables (BCHTs). On the other hand, Polychroniou et al. [19] present a generic design for SIMD-aware batch hash tables and compare the performance of different design decisions by doing intensive benchmarks. To answer some of the open questions in vectorized hash tables, [30] conducted a survey on state-of-the-art and conducted micro-benchmarks to position each work with respect to the others.

SIMD-Aware Batch Hash Tables. To use SIMD features of a CPU in an operation (logical, arithmetic, memory, etc.), we first need to construct a vector of operands that fit the CPU register size. Then, the prepared register could concurrently process the data by using SIMD instructions. Modern CPUs offer even more advanced SIMD instructions such as selective load/store and scatter/gathers. Selective load/store makes the parallel optional load/store from/to a contiguous memory location possible by accepting a mask register. The gather/scatter operations provide the ability to load/write from/into different parts of the memory in parallel. At the time of writing this paper, the SIMD scatter is not widely adopted, and is only provided by specific hardware (e.g., Intel Xeon Phi).

[1] Although in the literature the terms batch and vectorized are used interchangeably, for the sake of consistency, in this paper, we use the term batch to refer to a collection of elements, and vectorized to refer to SIMD vectors.

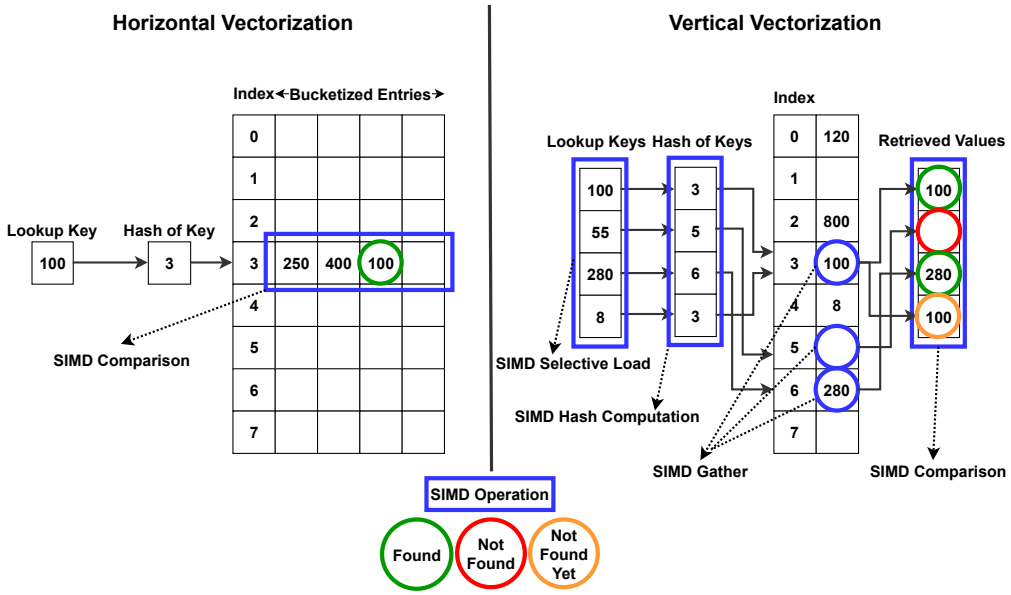

Figure 1 The comparison between horizontal and vertical vectorization.

As mentioned earlier, SIMD techniques can be used to improve the performance of batch hash tables. In high-level, the usage of SIMD in hash tables could be categorized in two: (1) Horizontal Vectorization and (2) Vertical Vectorization. A simplified visualization of these two approaches is depicted in Figure 1.

In horizontal vectorization, each cell of the hash-table entries array is bucketed into N inner cells. Then, while doing a lookup on the hash table and after computing the hash value, using the SIMD logical operations, the lookup algorithm can concurrently check the value of N bucket keys. This is much faster than having only one key per cell. By using this approach the hash table load factor can be improved without increasing the average lookup time. Regardless of the mentioned benefits of horizontal vectorization, it is wasteful if we expect to look up fewer than N buckets on average per probing key [19]. One of the famous open-addressing collision resolution algorithms in hash tables is Cuckoo Hashing [18]. A BCHT, as defined earlier, is actually the horizontally-vectorized version of cuckoo hashing. Many of the existing approaches on SIMD-aware batch hash tables [9, 22, 23] are in fact the improved version of BCHTs. Besides presenting a BCHT approach, Polychroniou et al. [19] also proposed and compared other horizontally-vectorized hash tables based on double hashing and linear probing hashing schemes.

Vertical vectorization [19] is a more generalizable but more complex approach to benefit from SIMD in batch hash tables. It is more generalizable because it does not change the inner structure of a hash table. However, it is more complex as it needs the collision-resolution algorithm to be translated into SIMD operations. Contrary to horizontal vectorization, in this approach, the input of a lookup operation must be a vector of probing keys. In each vectorized lookup, the vertical approach will pass a vector (of register size) of inputs through the lookup process and by using mask registers and advanced SIMD features (like SIMD permutations) probe those keys at the same time. When the status of a key lookup is determined (found/not-found), its related CPU-register lane will be assigned to the next key

in the batch of keys that are waiting to be processed. By conducting different experiments, Polychroniou et al. [19] and Shankar et al. [30] have shown that vertical vectorization yields higher performance than the horizontal approach. In vertical vectorization, since the hashing scheme must be translated into SIMD code, we need to use gathers and scatters to read/write from/to different entries of a hash table. As mentioned earlier, the scatter instruction is only available in limited types of processors hence the vertically-vectorized insertions can only be implemented on specific hardware.

The third category for SIMD-aware batch hash table design is a hybrid approach. However, the experiments in [30] show that the results of mixing the vectorized and horizontal vectorization approaches will not further improve the performance.

Besides the SIMD-aware vectorization methods discussed above, SIMD operations can also be used in the development of hash functions in any hash table [2]. Although this approach can improve the performance of hashing, it is orthogonal to the scope of this paper.

Prefetching-Enabled Hash Tables. Modern CPUs support hardware and software prefetching. Prefetching improves the performance of a program by amortizing the costs of memory access over time (in parallel to running computations).

Hardware prefetching is automatically enabled by the compiler and executed by the CPU when long and contiguous access to memory (e.g. iteration on a large vector) is requested by the program. The developer does not have much control over the hardware prefetching. On the other hand, in software prefetching, the developers can issue on-demand asynchronous prefetching commands to prefetch their future memory accesses.

In hash tables, regardless of the hashing scheme, accessing entries is based on the value of the computed hash for each provided key. This is an example of random access to a non-contiguous memory that can be improved by using the software prefetching. To have an effective prefetching in hash tables we need (1) a batch of operations and (2) a large hash table. The batch of operations provides enough computational tasks for the CPU while the prefetching instructions for future memory accesses are being processed. Also, if the hash table is not large enough, its content can entirely fit into the CPU cache. This cancels the benefits of prefetching and only puts its overheads on the CPU.

The effects of using software prefetching in hash tables have been studied in [11, 23, 32]. In the networking community, Scouarnec et al. [23] and Zhou et al. [32] have shown the effects of using prefetching on network-specific batch hash tables. They proposed different approaches and improvements in applying prefetching on BCHTs. In the database community, by proposing two generic techniques of using prefetching, Chen et al. [11] have shown their impact on the performance of relational hash joins. Among the off-the-shelf open-source hash tables, we found phmap [5] as the one that provides a `prefetch_hash` API in its interface. However, it delegates the responsibility of using this API (and designing a good prefetching strategy) to the developer who is not necessarily a system-level developer. There exist challenges in combining software prefetching with vectorization in the context of hash tables. We cover these challenges and their related design decisions in Section 4.3.

Parallel (Concurrent) Hash Tables. There is a long tail of research and open-source projects on parallel (i.e., multi-threaded) hash tables. The state-of-the-art systems [5, 7, 13, 15] have tried to enable concurrent insertion, lookup, and deletion on hash tables. These approaches can generally be divided into two categories: (1) the approaches that resolve the contentions using lock-based mechanisms, and (2) the lock-free hash tables that use atomic instructions, such as Compare-and-Swap (CAS) as their synchronization mechanism. Although these

parallel hash tables offer better performance in comparison with the sequential hash tables, they are not fully exploiting the advanced features of modern hardware such as SIMD awareness and prefetching. This is due to the lack of a batch API.

Although most of the batch hash tables offer batch insertions or deletions, previous research has shown that in most of the high-performance use cases (e.g. join processing in relational algebra, vector/tensor processing in linear algebra, and packet processing in computer networks), the amortized cost of insertions is negligible in comparison with the overall cost of highly frequent (or even endless and continuous) lookups [9]. Thus, in this paper, we only consider optimizing the lookup performance.

3 Architecture

In this section, we discuss the structure of Vec-HT and its high-level API.

3.1 Hash-Table Structure

The hash table consists of an array of `bucket` objects each of which contains a key (32 bits) and its related value (32 bits). As the hashing scheme, we use open addressing with linear probing (similar to [19]). We also use multiplicative hashing as our hash function.

Generally and without considering any optimization, to look up a key in the hash table, we first compute the hash of the key and then find its corresponding bucket in the array. If the key of that bucket is empty (the value of the empty key is defined during hash table initialization) we return the empty key which means "not found". Otherwise, we check if the key in the bucket equals the probing key or not. If it is, we return the value, otherwise, we continue checking the next buckets to find an equal key or an empty bucket.[2]

3.2 High-Level API

To make a batch hash table more accessible to developers, we expose an easy-to-understand API. The Vec-HT namespace consists of three classes. A batch hash table class (`lp_map`) that is currently designed by having the open-addressing linear-probing hashing scheme (Figure 2), a batch-iterator class (`iter_batch`) that defines the data type containing the result of batch lookup, which also supports parallel iterations (Figure 3), and the `bucket` class (cf. Secion 3.1) that is related to each entry of the hash table.

Batch Hash Table Class. The constructor of `lp_map` takes three arguments. The first one (`size`) is for setting the maximum number of elements that will be inserted into the hash table. The second parameter sets the group size for the internal prefetching. And the third parameter (`threads`), determines the number of threads (cores when the Hyper-Threading is disabled) to enable concurrent batch processing.

The methods exposed by the API of `lp_map` are categorized into two sets: non-batch and batch methods. The non-batch methods include `insert`, `find` that are similar to the standard hash table interfaces such as `std::unordered_map`. These methods give the developers the freedom of using Vec-HT without batch processing.

There are five batch-based methods. The method `insert_batch` inserts an array of keys and their related values into the hash table. The remaining methods are related to vectorized lookup which is the main focus of this work (cf. Section 2). The method `find_batch` accepts

[2] Currently, due to the restrictions imposed by SIMD-vectorization, Vec-HT only supports 32-bit integer keys (similar to [1, 19, 23]).

```
namespace vec_ht
{
    using K = uint32_t;
    using V = uint32_t;
    using P = uint32_t;
    // ---- Linear-Probing Batch Hash Table Class -----------
    class lp_map
    {
        // ...
    public:
        lp_map (size_t size, size_t group_size=64, size_t threads=1);
        // ---- Non-Batch APIs ------------------------------
        inline bool insert (const K& key, const V& value);
        inline bucket* find (const K& key);
        // ---- Batch APIs ----------------------------------
        inline size_t insert_batch (uint32_t* keys, uint32_t* values,
        size_t size);

        inline size_t find_batch (uint32_t* keys, size_t size,
        bool complement, iter_batch* res_it);

        inline size_t find_batch_apply (uint32_t* keys, size_t size,
        bool complement,
        std::function<void(K& key, V& value)>const& f);

        inline size_t zip (uint32_t* keys, uint32_t* payloads,
        size_t size, bool complement, iter_batch* res_it);

        inline size_t zip_apply (uint32_t* keys, uint32_t* payloads,
        size_t size, bool complement,
        std::function<void(K& key, V& value, P& payload)>const& f);
    };
}
```

Figure 2 High-level API of Vec-HT in C++.

three arguments: (1) an array of keys to look up, (2) the size of that array, (3) a boolean flag called `complement` that is used to request for the not-found elements instead of the successfully-found ones, and (4) an object of a Vec-HT-specific class called `iter_batch`. The `iter_batch` class is responsible for keeping the results of a vectorized lookup and making the (parallel) iterations over them possible. The other method in `lp_map` is `zip`. Although it is not a usual API for ordinary hash tables, we found it very useful in the case of batch hash tables. This method, similar to the `find_batch`, does a lookup for the provided array of keys. However, it takes one additional argument; `payloads` assigns one value to each key in the keys array. When the method `zip` is called, the result also contains the related payloads of the found keys. The `iter_batch` class can also keep the results of a `zip` API. We show how `zip` can be used in practice in Section 5.

The remaining useful APIs are `find_batch_apply` and `zip_apply`. These APIs do the same job as their related discussed APIs. However, a user can pass their customized lambda function to be applied on the tuples of key-values (or key-value-payloads) whenever a match is found. As a result, there is no need to pass a `iter_batch` object to these APIs since it is

```
namespace vec_ht
{
    // Container and Iterator Class for find_batch/zip Results
    class iter_batch
    {
    // ...
    public:
        iter_batch (size_t max_size, size_t threads,
        bool for_zip=false)

        K** get_keys();
        V** get_values();
        P** get_payloads();

        inline void foreach
        (std::function<void(K& key, V& value)> f);

        inline void foreach
        (std::function<void(K& key, V& value, P& payload)> f);

        inline void foreach_parallel
        (std::function<void(K& key, V& value)> f);

        inline void foreach_parallel
        (std::function<void(K& key, V& value, P& payload)> f);
    };
}
```

Figure 3 High-level API of batched iterator in C++.

the user's responsibility to handle the output. These two APIs can improve the performance of pipelined analytical tasks because they eliminate the need for the materialization of intermediate results (`iter_batch`). In other words, using these APIs, the user can fuse the batch lookups with the following operations in the pipeline. This is especially useful in the context of pipelined analytical query processing [25, 29, 17].

Batch Iterator Class. The `iter_batch` class can be constructed by passing (1) an upper bound on the size of the results, (2) the number of threads (it must be the same as the one in `lp_map`), and (3) a boolean that shows if we want to pass this object to a `zip` or a `find_batch` API. By receiving these parameters, the memory needed for the storage of parallel-processed results will be allocated. Then, the class is ready to be sent to the methods `find_batch` or `zip` in a *destination-passing style* [26, 31]. The destination-passing style improves the performance of computational workloads by bringing the memory-allocation overheads out of the performance-critical part of the workload. The `iter_batch` class has also two overloads of `foreach`. After the execution of `find_batch` or `zip`, their relevant `foreach` method can be used for a sequential iteration over the results. The `foreach` method takes a lambda function that will be applied to each of the stored results in the `iter_batch`. Similar to the `foreach` methods, `iter_batch` also offers `foreach_parallel` methods that use multiple threads to apply the provided lambda function on the stored results. In the `foreach_parallel`

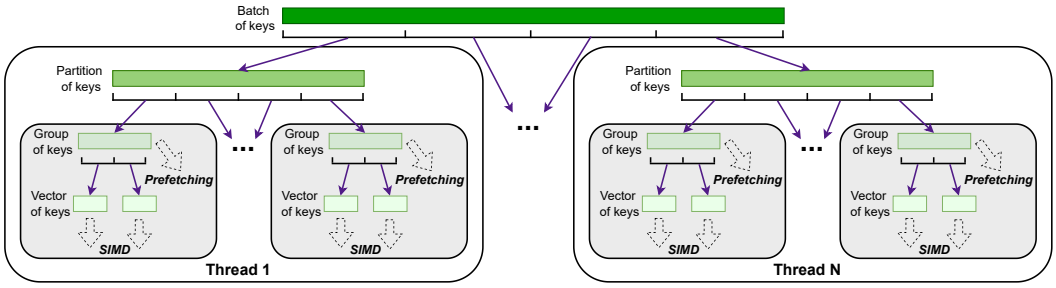

Figure 4 The architecture of batch lookup in Vec-HT.

```
template<typename FUNC_TYPE>
inline size_t parallel_dispatcher(uint32_t* keys,
                                  uint32_t* payloads,
                                  size_t size,
                                  bool complement,
                                  FUNC_TYPE func,
                                  iter_batch* res_iter)
```

Figure 5 The signature of `parallel_dispatcher`, a function used internally for parallelization.

methods, since they are internally implemented based on `tbb::parallel_for_each` [7], the developer can also use parallel containers such as `tbb::enumerable_thread_specific` or any other off-the-shelf parallel container to handle storing/aggregation of lambda outputs.

4 Design

In this section, we discuss the design decisions behind the optimizations in our approach and show how they relate to each other. Figure 4 shows the architecture of a batch lookup in Vec-HT. As it is shown in this Figure, the batch input is partitioned into smaller chunks on different levels and for different optimization purposes. In this section, these levels of input partitioning and the rationale behind them will be covered.

4.1 Parallel Processing

To make the most out of the multi-core processor, in case of a batch lookup, we partition the input batch of keys and assign each partition to a thread for parallel batch processing. When the `find_batch` or `zip` methods are called, they call the `parallel_dispatcher` method internally. This lower-level method is responsible for managing the threads needed for the computation and passing them the contextual information. The interface of `parallel_dispatcher` is shown in Figure 5.

The `parallel_dispatcher` method takes six arguments:

1. the keys that we want to look up in the hash table,
2. the associated payloads (set to `NULL` if the caller method is `find_batch`),
3. the size of the `keys`,
4. the `complement` flag (cf. Section 3.2),
5. the lambda function that is passed when the user calls `find_bath_apply` or `zip_apply`,
6. the `iter_batch` which is passed in case of calling `find_batch` or `zip`.

4.2 SIMD-Awareness

In this section, we explain the vertical-vectorization approach for batch lookups at a high level and refer the interested reader to Polychroniou et al. [19] for more details.

Suppose that a number of `W` keys can be stored in a CPU register. When the batch lookup starts, `W` keys of the input vector will be fetched into the `keys` register. To load the input keys, we use the selective load SIMD operation. This operation uses a mask to select which lanes of the target register must be filled with the new values and which of them must be set to zero. In the beginning, we define a register (`invalid`) with all lanes activated and pass it to the selective load as the mask. This means that we plan to read `W` new keys from the input. Then using the SIMD operators, the hash value of all the keys in `keys` is computed simultaneously and stored in the `hash` register. In Vec-HT, as we use a simple multiplicative hash function, the computation of hash values consists of logical and arithmetic SIMD operations such as vectorized multiplication and shift.

By having the hash values, we use the SIMD `gather` operation to retrieve the needed hash table entries. The `gather` operation reads multiple memory addresses (stored in a register) at the same time. Since the value of each computed hash shows the possible offset of each key in the hash table entries, we apply `gather` on the address of the hash-table entries array and the `hash` register. As a result of executing two `gather` operations, two registers for the retrieved keys (`tab_keys`) and values (`tab_vals`) will be created.

Next, based on the linear probing algorithm, we check the equality of key in `keys` and `tab_keys`. We do this check using SIMD logical operators. This check can have three different results for each lane: (1) the key is empty which means that the key is not found in the table (2) the keys are equal which means the key is found (3) the key is not empty or is not equal to the given input key and thus it needs further probing in the next rounds. For the not-found keys, we activate their relevant lanes in `invalid` register. For the found keys, we define and activate the relevant lanes in a new register called `output`. Finally, for the ones that need further probing, we create a new register called `offset`, initialize it with `0`, and increment its relevant lanes by `1`.

In this phase of the algorithm, we first add `out` to `invalid` and store the result in `invalid`. We do this since we are finished with both found and not-found keys and we want to fetch the new keys instead of them in the next round of lookups. Then, by using a static permutation table, we extract the permutation masks needed to align the active lanes of `invalid` and `output` to one side of the register. These permutation masks will be used in the SIMD `permute` operation, which changes the order of lanes in the register using a provided mask. First, we use the permutation mask of `out` on `out` itself and on `keys`. Now, we are ready to save the found keys to the target memory (reserved memory in `iter_batch`) using a selective store SIMD operation that its mask is the permutated `out`. The total number of output keys will also be updated at this stage. It is notable that the original vertical vectorization [19] uses a buffer to store the results temporarily and spills them to the output whenever the buffer is full.

After the work on the found keys is finished, we count the active lanes in `invalid` to know how many new keys are needed to be fetched. Then, we apply the `invalid` permutation mask on `keys`, `hash`, and `offset` to make them ready for the next run. By starting the next round of lookups, again the new keys will be fetched based on the updated `invalid` register. It is important to mention that this time all of the hashes are re-computed and the ones with inactive lanes in `invalid` will also be incremented by `offset` to point to the next entry in the hash table.

To return the value of found keys in the table and also their related payloads we need further considerations. For the hash table values, which are stored in `tab_val`, we can permute them using the `out` permutation mask and store them in their related memory in `batch_iter`. Similarly, the payloads can be selectively loaded exactly similar to the new key. Then they will be passed through the algorithm by similar permutations, and finally will be stored in the relevant output memory.

If the size of input keys is less than `W` or in the case of processing the last `W` keys, the algorithm switches to a normal scalar (non-SIMD) lookup in the hash table and stores the result into the `batch_iter`. This is because there are not enough keys to do a safe and efficient SIMD lookup.

4.3 Prefetching and Its Adaption Challenges

Vec-HT is a prefetching-enabled vectorized approach; we apply the prefetching on top of the parallel vertical vectorization. There is a large design space for combining prefetching and SIMD vectorization. We examined this design space through micro-benchmarking (cf. Section 7). The important design parameters for this combination are as follows:

- **Standard vs Group Prefetching:** putting the prefetching commands at the beginning of the main loop of the vertical vectorization is the standard solution for adding prefetching. We compare it with another approach (Group Prefetching) proposed by Chen et al. [11] in the context of databases.
- **Group Size for Group Prefetching:** Considering the group prefetching approach, the selection of the different group sizes might affect the performance of the system.
- **Optimistic vs Pessimistic Linear Probing:** Given that we use linear probing, there are two choices for prefetching for each key. Optimistic: we consider that the probe hits the correct location on the first try (or finds the location to be empty) and does not need to probe further; thus we only prefetch the hashed location. Pessimistic: we consider the case of not having a hit and thus prefetching the next location(s) as well.
- **Memoization of Computed Hashes:** We need the hash values in two places: (1) prefetching stage, and (2) vertical vectorization stage. We have the option of memoizing the hash value in the first stage and reusing/recomputing it in the second stage.
- **Buffering:** In vertical vectorization, we can write the output into an output buffer and if it is not carefully adapted to the prefetching design, it might result in performance overheads.

Figure 6 depicts a generic and high-level algorithm for combining prefetching with vertical vectorization (based on the assumption that we take the group-prefetching approach instead of standard prefetching, which is a take-away message of micro-benchmarks in Section 7). In this algorithm, by setting the `GROUP_SIZE` parameter, we can enable the grouping loop that partitions the input keys into parts of size `GROUP_SIZE` and then run the algorithm on these smaller batches. By having a group of keys as input, before starting the vertical vectorization, we define a loop over the group keys (prefetching loop). In each iteration of the prefetching loop, we first compute the hash of `W` elements using the SIMD approach mentioned before. By setting the `HASH_MEMOIZE` parameter to `true`, we can store the hash values and reuse them inside the vertical vectorization algorithm. After making a decision on memoization, we raise `W` software prefetch commands for the address of target entries in the hash table. Here we can do the prefetching also for the next bucket by setting the `OPTIMISTIC` parameter to `false`. After finishing the prefetching loop, all the related entries are prefetched. At this stage, the vertical vectorization algorithm will be executed for the current group and if the `OUT_BUFFER` parameter is enabled, the output buffering happens.

```
foreach group in array by GROUP_SIZE {
  // prefetching stage
  foreach vector in group {
    vector_h <- simd_hash(vector)
    if(HASH_MEMOIZE)
      mem_h += vector_h
    foreach h in vector_h {
      prefetch(buckets[h])
      if(!OPTIMISTIC)
        prefetch(buckets[h+1])
    }
  }
  // vertical vectorization stage using linear probing
  while(vec_elems not probed in group) {
    if(HASH_MEMOIZE)
      vector_h <- mem_h[vec_elems]
    else
      vector_h <- simd_hash(vec_elems)
    res <- vertical_vectorization(vec_elems, vector_h)
    if(OUT_BUFFER) {
      buffer += res
      if(buffer is full)
        flush(buffer)
    }
  }
  if(OUT_BUFFER) {
    flush(buffer)
  }
}
```

Figure 6 A generic algorithm showing the design space of combining prefetching with vertical vectorization.

5 Use Cases

In this section, we show the usability of our proposed batch table, by showing several high-performance data analytics use cases.

5.1 Relational Hash Join

First, the code for a join on two relations (S and R) is shown in Figure 7. We assume that the relations are stored in columnar format (i.e., struct of arrays) which is a popular design decision in high-performance query engines [17]. The code is executed using a prefetching group size of 64 on 4 threads. We keep these settings for all of the use cases covered in Section 5.

Build Phase. In the beginning, the batch hash table is initialized with the table size, group size, and the number of threads. The size is set to twice the number of the elements in the relation on the build side of the hash join (`S`). We do so to keep the fill ratio of the hash table less than or equal to 50%. Then, using the `insert_batch` method, all the key/value pairs from `S` are inserted into the hash table in a batch style.

```
// build phase
auto ht = vec_ht::lp_map(2*S_size, 64, 4);
ht.insert_batch(S_A, S_B, S_size);
// probe phase
auto res_it = vec_ht::iter_batch(R_Size/3, 4, true);
ht.zip(R_A, R_F, R_size, false, res_it);
// printing the output
res_it.foreach_parallel(
[](auto& key, auto& value, auto& payload){
    std::cout << "S_A/R_A: " << key << " | ";
    std::cout << "S_B: " << value << " | ";
    std::cout << "R_F: " << payload << std::endl;
});
```

Figure 7 Implementation of a hash join operator (on S and R relations) using Vec-HT.

```
auto ht = vec_ht::lp_map(2*S2_size, 64, 4);
for (int i=0; i<S2_size; i++) ht.insert(S2[i], 1);
auto res_it = vec_ht::iter_batch(S1_Size/5, 4, false);
ht.find_batch(S1, S1_size, true, res_it);
res_it.foreach_parallel([](auto& key, auto& value){
    std::cout << "Item: " << key << std::endl;
});
```

Figure 8 Implementation of a Set-Difference operation (S1\S2) using Vec-HT.

Probe Phase. Before running the batch lookups on the hash table, we first prepare the `iter_batch` for the results. This object is initialized by setting three parameters. The first one is an upper bound for the join result size. The more precise this estimation is, the less memory allocation time is spent during the batch lookups. The second parameter is the number of threads that must be equal to the one already passed to the hash table. The last parameter is a flag that shows if the `iter_batch` object will be used in a `zip` or `find_batch` API. Next, by having the `res_it` object, we can run our `zip` method to join the relations based on the `R_A` and `S_A` columns and by considering the `R_F` column as the payload.

After the `zip` execution is finished, we use the `foreach_parallel` method of `res_it` to iterate over the join results and print them. The desired functionality (printing) is passed to the `foreach_parallel` using a user-defined lambda function.

5.2 Set Operations

As our second use case, we show the implementation of a set difference operation (S1\S2) using our approach in Figure 8. The sets S1 and S2 are stored in two arrays. The code is almost the same as the one in the relational-join example. Its main difference is in using `find_batch` instead of `zip`. It is because there is no payload to be passed into the `zip` API. Furthermore, in this example, we see the usage of `complement` parameter as we need the elements of S1 that are not found in S2. To implement the set intersection we need to use the `zip` method, which is similar to the vector inner product, that is presented next.

```
auto ht = vec_ht::lp_map(2*V1_size, 64, 4);
ht.insert_batch(V1_idx, V1_val, V1_size);
auto res_it = vec_ht::iter_batch(V1_size, 4, true);
ht.zip(V2_idx, V2_val, V2_size, false, res_it);
uint32_t sum = 0;
res_it.foreach([](auto& key, auto& value, auto& payload){
    sum += value * payload;
});
```

Figure 9 Implementation of an Inner-Product operation (V1·V2) using Vec-HT.

5.3 Sparse Vector Operations

The last use case that we cover in this section is the inner product of two vectors (V1·V2). The related code is shown in Figure 9. In this implementation, we again use the `zip` method, since there are payloads on the `V2` side (values of `V2` for each index). After the `zip` execution is finished, this time we do a non-parallel iteration (`foreach` API) over `res_it` to prevent contentions on the `sum` shared memory. As mentioned in Section 3.2, a developer can easily use the concurrent containers (e.g. `tbb::enumerable_thread_specific<uint32_t>`) instead of `sum` here. However, in this case, we are interested in exhibiting the usage of non-parallel `foreach`.

6 Implementation

In this section, we give a more detailed explanation of the implementation behind Vec-HT.

Attributes of `lp_map`. The attributes of `lp_map` class are shown in Figure 10. All these attributes are initialized in the class constructor. The `size`, `threads_`, and `group_size_` attributes are set to the values that are passed by the constructor. The `hash_factor_` is set to a randomly generated number. The `empty_key_` attribute is set to the maximum possible value for `uint32_t` type. Lastly, we use an array of `bucket` structs (`entries_`) as the entries of our hash table. It has been shown that using an array of structs instead of a struct of arrays does not affect the performance of hash tables [21]. The memory of this array is allocated after the calculation of `size_` attribute.

Attributes of `iter_batch`. The attributes of the `iter_batch` class are shown in Figure 11. Here `max_size_` is passed by the constructor and shows an upper bound on the number of results for a `find_batch` or `zip` method call. The `threads_` and `for_zip` attributes are given by the constructor. The next three attributes are the storage for results of a `find_batch` or `zip` method call. In the constructor, we allocate arrays of arrays to these pointers; this will allocate memory of size `max_size` for each thread. In the case of a `find_batch` method call, we do not allocate and use the `payloads_` pointer. As the last attribute, we have `threads_res_size_` that will be extended to the size of `threads_`. Each element of this vector is used by a thread to store the size of the results for that thread. By using this attribute, the iterations over the results will be more efficient (cf. Figure 12).

Implementation of `foreach_parallel`. In Figure 12, the implementation details of `foreach_parallel` are presented. In this method, we create a range of integers from `0` to `thread_-1` and assign each number in the range to a thread. Then, by execution of a

```
namespace vec_ht
{
    class lp_map
    {
    private:
        size_t size_;
        size_t threads_;
        uint32_t group_size_;
        uint32_t hash_factor_;
        uint32_t empty_key_;
        bucket* entries_;

        template<typename FUNC_TYPE>
        inline size_t parallel_dispatcher(uint32_t* keys,
           uint32_t* payloads, size_t size, bool complement,
           FUNC_TYPE func, iter_batch* res_iter)

        template<typename FUNC_TYPE>
        inline size_t find_batch_inner(uint32_t* keys,
           uint32_t* payloads, size_t size, bool complement,
           FUNC_TYPE func, iter_batch* res_iter, size_t thread_id)
    public:
        // ...
    };
}
```

Figure 10 The internal of the `lp_map` class.

`tbb::parallel_for_each` and passing the prepared range to it, we run a lambda function on each thread with `thread_id` as its single argument. In the lambda function, using the `thread_id` argument, the `max_size_` attribute of `batch_iter` class, and the vector of result sizes for each thread (`threads_res_size_`), we compute the boundaries of the result vectors that are assigned to the current thread. By having those boundaries, we can finally apply the developer-provided lambda (`func`) on each triple of `key`, `value`, and `payload` in the results assigned to this thread.

Implementation of `zip`. The implementation of the `zip` method is presented in Figure 13. To bypass the overheads of dispatching in the parallel scenario, this method (and other performance-critical methods such as `find_batch`), checks the `threads_` attribute of the current `vec_ht`. If it detects a sequential setting, then calls the internal method that is responsible for the vertical vectorization and prefetching (`find_batch_inner`). Otherwise, the method calls the `parallel_dispatcher` (cf. Section 4) to partition and dispatch the work among the pre-determined number of threads. To call either of these two internal methods, the `zip` method provides them with the appropriate arguments or `null` types where required.

Implementation of `find_batch_inner`. Figure 14 shows a simplified implementation for `find_batch_inner`. This method is the most complex method in Vec-HT. It operates over a subset of batch input (the partition that is assigned to each thread) and is responsible to do the following tasks:

```
namespace vec_ht
{
    class iter_batch
    {
    private:
        size_t max_size_;
        size_t threads_;
        bool for_zip_;

        uint32_t** keys_;
        uint32_t** values_;
        uint32_t** payloads_;

        std::vector<size_t> threads_res_size_;
    public:
        // ...
    };
}
```

Figure 11 The internal of the `iter_batch` class.

```
inline void foreach_parallel
(std::function<void(K& key, V& value, P& payload)> f)
{
    auto range = std::vector<size_t>(threads_);
    for (size_t i=0; i<threads_; i++) range[i] = i;
    tbb::parallel_for_each(range, [&](size_t thread_id)
    {
        for (size_t j=0; j<threads_res_size_[thread_id]; j++)
            func(keys_[thread_id][j],
                 values_[thread_id][j],
                 payloads_[thread_id][j]);
    });
}
```

Figure 12 The implementation of `foreach_parallel` in `iter_batch`.

- To partition the input into group-sized batches.
- To compute and memoize the hashes for each group.
- To do the group prefetching for each group.
- To run the entire vertical vectorization algorithm for each group.
- To buffer the found keys and their related values and payloads.
- To store the results into the `iter_batch` or apply `func` over them.

We present a brief overview of the above-mentioned steps in Figure 14. The sections with high similarity to the code provided by Polychroniou [19] et al. are removed for the sake of brevity. We refer the interested reader to see those parts in the referenced work. Note that in Figure 14, the `vector_size` is a global constant (8) which is a function of the selected data-type size (32 bits) and the SIMD vector size (256 bits), computed as vector size divided by data-type size.

```
inline size_t zip (uint32_t* keys, uint32_t* payloads, size_t size,
bool complement, iter_batch* res_it)
{
    if (threads_ == 1)
        return find_batch_inner<no_func_type>(keys, payloads, size,
        complement, nullptr, res_it);
    else
        return parallel_dispatcher<no_func_type>(keys, payloads,
        size, complement, nullptr, res_it);
}
```

Figure 13 Implementation of `zip` in `lp_map`.

As the last topic in this section, to implement the complement behaviour in Vec-HT, we have slightly changed the original vertical-vectorization algorithm. In the case of a complement, we replace the keys with an *invalid* status with the keys with an *output* status. In other words, the found keys are considered *invalid* and the not-found keys are the *valid* ones that must be stored in the output.

7 Evaluation

In this section, we first present our experimental setup for the evaluation. Then, we show the performance of our approach in different use case scenarios and compare its performance with various competitors.

7.1 Experimental Setup

All experiments are done on a single machine equipped with 16GB of DDR4 RAM, and an Intel Core i5-10210U 1.6GHz with 4 cores and 256KB, 1MB, and 6MB of L1, L2, and L3 cache respectively. Hyper-threading is disabled for the experiments. We have used Ubuntu 20.04.3 as OS. Our C++ code is compiled with G++ 9.4.0 using the `-O3` flag. To enable SIMD operations, we use the `-march=core-avx2` flag. All of the experiments were executed with 5 warmup rounds followed by 5 timed iterations. Then, we took the average of the timed iterations.

Workloads. To run the experiments, we use three different workloads. For the micro-benchmarks and the join experiments, we use the random data generator from [19]. By focusing on the notion of inner joins in databases, it generates two random data sets as inner and outer relations. The elements of the inner data set are inserted into the hash table creating the build side of the join. The elements of the outer data set shape the probe side of the join. The data generator accepts arguments for `inner_size`, `outer_size`, and `selectivity` of the join.

The second workload is used for the set and sparse vector experiments. The set/vector generator receives `size`, `density`, and `maximum_value` as input parameters. For each set of size `N`, it generates `N`×`density` unique random numbers from the range of `[0, N)` as the value of items in the set. Similar to the sets, for the vectors, it generates unique random numbers but uses them as vector indexes. Then, using the `maximum_value` parameter, it generates random integer numbers in the range of `(0, maximum_value]` as vector values.

```
inline size_t find_batch_inner (uint32_t* keys, uint32_t* payloads,
   size_t size, bool complement,
                                FUNC_TYPE func, iter_batch* res_iter,
                                size_t tid)
{
    // Partitioning the input keys into group-sized batches
    size_t inner_batch_size = group_size_;
    for(size_t i=0; i<size; i+=group_size_)
    {
        if (size-i<group_size_)
          inner_batch_size = size%group_size;
        // Hash memoization and Group prefetching
        uint32_t hashs[inner_batch_size];
        for (size_t j=0; j<inner_batch_size; j+=vector_size)
        {
            // Hash computation for keys using SIMD operations
            // ...
            // Hash memoization and Group prefetching
            for (size_t k=0; k<vector_size; k++)
            {
                // Storing the hash in hashs[j+k]
                // ...
                // Prefetching the computed and stored hash
                _mm_prefetch(&entries_[hashs[j+k]], _MM_HINT_T0);
            }
        }
        // Execution of vert. vect. using memoized hash values
        // considering the "complement" flag (if enabled) ...

        // Buffering the matched keys, values (and payloads) ...

        // Flushing the buffer into the iter_batch container or
        // Applying the lambda function on the buffered tuples ...
    }
}
```

Figure 14 A simplified representation of the `find_batch_task` implementation in `lp_map`.

As the last workload, we use the well-known TPC-H [8] benchmark with a scaling factor (SF) of 1 (1 GB of data) for the evaluation of our approach in analytical queries. It is important to note that in all of the benchmarks, we keep the fill ratio of all alternative hash tables less than or equal to 50%.

Alternatives and Competitors. In the micro-benchmarks, to evaluate our proposed approach, we compare it with (1) a scalar implementation of Vec-HT without any optimization (2) a scalar + prefetching version (3) and the vertical-vectorization approach by Polychroniou et al. [19]. For all alternatives, we consider sequential and parallel versions. As mentioned in Section 6, we reuse the code from [19] as the base for our implementations. We do not add the comparison with the approaches such as DPDK [1], as it is previously shown [19] that the BCHT approach is slower than vertical vectorization which is the basis for Vec-HT.

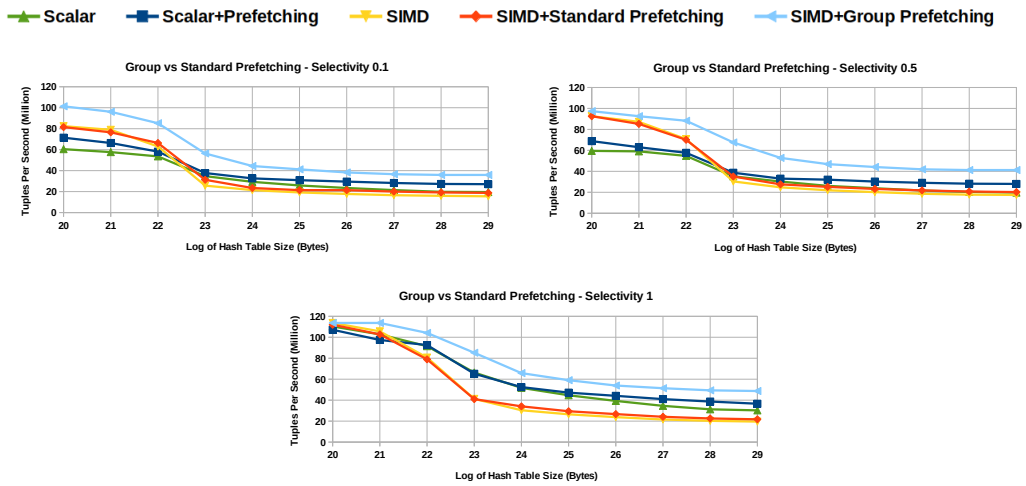

Figure 15 Comparing the performance of standard vs group prefetching for combining prefetching with vertical vectorization. The charts show the number of tuples processed per second (higher is better) for a simplified relational inner-join operation with a probe size of 1,500,000 over an increasing build-side size.

We also compare our implementations with state-of-the-art hash tables. TBB [7] is a well-known parallel computation framework. We use its `tbb::concurrent_unordered_map` as one of our competitors. Libcuckoo [13] is a research project on fast parallel hash tables and we use its open-source implementation `libcuckoo::cuckoohash_map`. As the last competitors, from the open-source high-performance hash-table project phmap [5], we use its sequential and parallel data structures `phmap::flat_hash_map` and `phmap::parallel_flat_hash_map`. The implementation of Vec-HT that we use in the benchmarks has a group size of 64, taking an optimistic approach, with enabled memoization and simple buffering inside each group.

7.2 Benchmarks

In this section, we first consider the design space of combining prefetching with vectorization and show the best implementation. In addition, we evaluate the effectiveness of our implementation in comparison with scalar, scalar + prefetching, and pure vertical vectorization in a holistic micro-benchmark for hash join processing. Then, we show its superiority over the existing hash table packages in different use cases. We consider benchmarks on set and vector kernels that are largely used in big data analytics frameworks such as query processors (e.g., BigTable, SparkSQL) and linear algebra frameworks (e.g., MLLib, SystemDS, distributed TensorFlow). We finally cover benchmarks on database query processing over a selected subset of TPC-H queries, the main benchmark for analytical queries.

Standard vs Group Prefetching Micro-Benchmark. The first micro-benchmark related to the design space (cf. Section 4.3) is shown in Figure 15. In these experiments, our focus is to show the performance difference between the standard prefetching and group prefetching approaches. The results show that even though the standard way of prefetching offers performance improvements compared to non-prefetched approaches, it cannot beat the performance and scalability of the group prefetching.

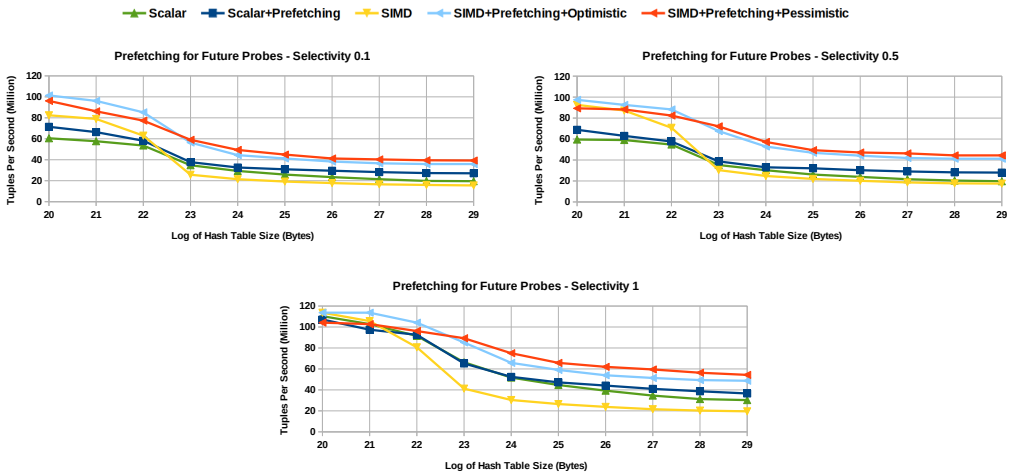

Figure 16 Comparing the performance of different alternatives by focusing on the optimistic and pessimistic approaches for prefetching. The workload is similar to Figure 15.

Optimistic vs Pessimistic Prefetching Micro-Benchmark. As mentioned in Section 4.3, by having a linear probing scheme in the hash table, for a given key, we can prefetch more than one bucket to improve the chance of a cache hit after an unsuccessful lookup. Although it seems to be an interesting strategy to take, the limited prefetching capability of CPUs, the variety in workload characteristics, and the parameters such as the fill ratio of the hash table can affect the benefits of this strategy. Figure 16 shows the performance results for prefetching with optimistic and pessimistic approaches. Both optimistic and pessimistic approaches perform better than the alternatives. However, for the smaller hash tables, the performance of the pessimistic approach is worse than the optimistic one and sometimes even worse than pure vertical vectorization. Thus, we decided to take the optimistic approach for our final implementation of Vec-HT.

Group-Size Micro-Benchmark. Figure 17 depicts the performance of group prefetching with different group sizes by also altering between the optimistic and pessimistic strategy. Overall, we see a performance improvement by increasing the group size, however, this improvement no longer holds after the group size of 64. The results show that selecting a group size of 64 with the optimistic strategy is a reasonable choice.

Hash Memoization Micro-Benchmark. Our last micro-benchmark investigates the effects of memoizing the hash values during the prefetching stage. Figure 18 presents the results of these experiments. In selectivities of 0.1 and 0.5, it is clear that the memorized version performs better than the non-memoized one. With a selectivity of 1, the non-memoized version does better (with a narrow improvement compared to memorized version) for larger hash tables, however, the memoized version is faster for smaller hash tables. Thus, we select the memoized version for Vec-HT.

Relational Inner Join for State-of-the-Art Hash Tables. In these experiments, using the first workload described in Section 7.1, we run a simplified relational inner-join operation using different hash table implementations to measure the performance of each approach.

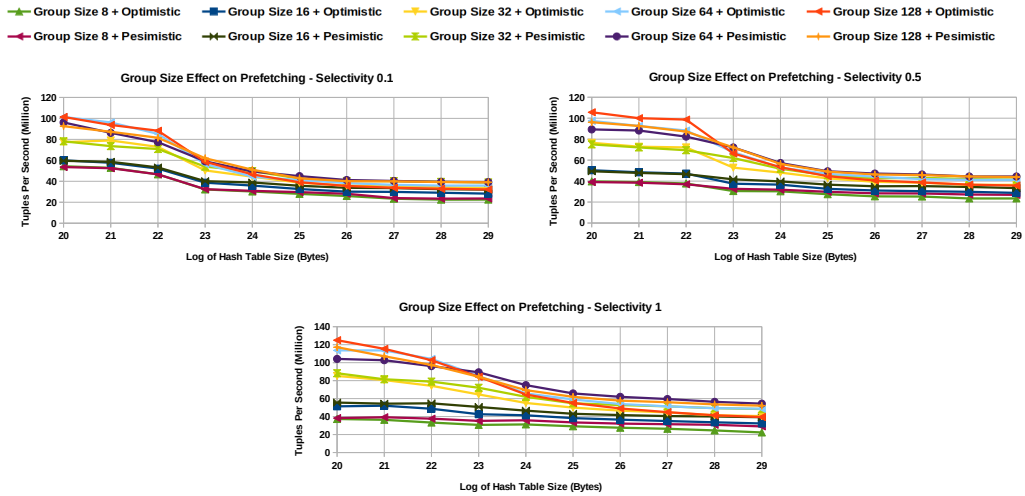

Figure 17 Comparing the performance of different combinations for group size and optimistic/pessimistic approaches for group prefetching. The workload is similar to Figure 15.

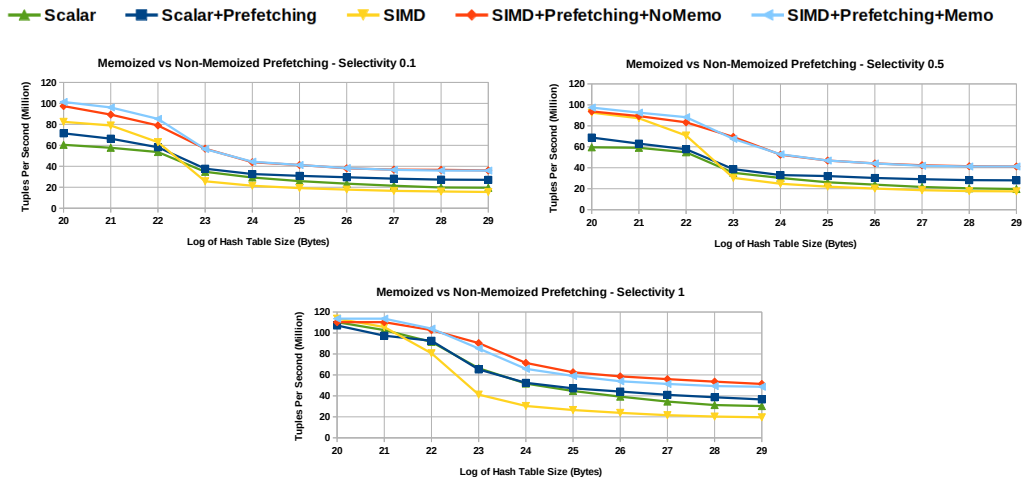

Figure 18 Comparing the performance of different alternatives by focusing on enable/disabled memoization for group prefetching. The workload is similar to Figure 15.

As it is shown in Figure 19, our approach `vec_ht::lp_map` outperforms other approaches irrespective of the build size, join selectivity or concurrency level. It is on average 5× faster on both 1- and 4-cores.

Set Operations Use Case. To show the performance of our approach in set operations, we run experiments on the set intersection and set difference. To run the operations, we iterate over the first set (S1) and look up the values in the other one (S2). Since the hash tables in our experiments (except `tbb::concurrent_unordered_map`) do not support parallel iterations, to have a more comprehensive benchmark, we keep S1 in the vector format and only make a hash table for S2. For the set-difference operation, we use an implementation similar to Section 5.2. Figure 20 depicts the results of our set experiments. In these experiments, using a fixed size for S1 and S2 and a fixed density for S1, we observe the changes in the run time of each competitor while increasing the density of S2.

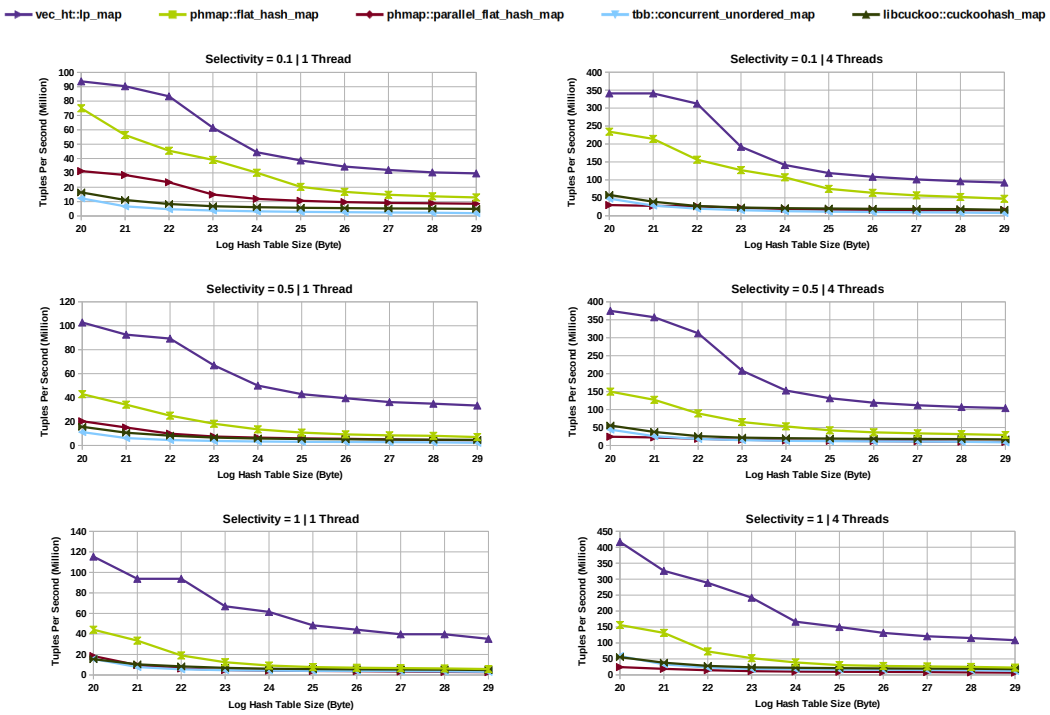

Figure 19 The number of tuples processed per second (higher is better) for a simplified relational inner-join operation with a probe size of 1,500,000 over an increasing build-side size. Charts on each side, represent the results for the join selectivities of 0.1, 0.5, and 1 on 1 or 4 threads.

In the set-difference experiments, our approach outperforms the others with an average of 11× speedup on 1 and 4 cores. Similarly, in set intersections, our approach performs better than the others excluding `phmap::flat_hash_map`. By excluding `phmap`, the proposed approach is on average 6× and 5× faster than the others on 1 and 4 cores, respectively. For the small S2 sizes, when the log of S2 density is less than or equal to -4, the hash table can still be fitted into the L3 cache, thus the benefit of using our prefetched approach is not promising and the overall performance is near to what `phmap::flat_hash_map` offers. However, after passing that size limit, `phmap::flat_hash_map` run time goes higher while our approach keeps its good performance thanks to software prefetching.

Vector Operations Use Case. In Figure 21, the results of running experiments on vector inner-product (cf. Section 5.3) and pair-wise multiplication are shown. Similar to the set operations, here we keep V1 in the vector format and embed V2 into a hash table. The experiment parameters are also set to the values that we had in the set experiments. In both vector operations, our approach is still faster than the alternatives. However, since the scenario of these two vector operations is very similar to the set intersection, we see similar behaviour in the performance of our approach versus `phmap::flat_hash_map`. We explained the reason behind this behaviour in the set experiments. The experiments on set and vector operations show that our approach is a great choice when we deal with large amounts of data; while the performance of other approaches degrades with increasing the hash table size, our approach maintains good performance even for heavier workloads.

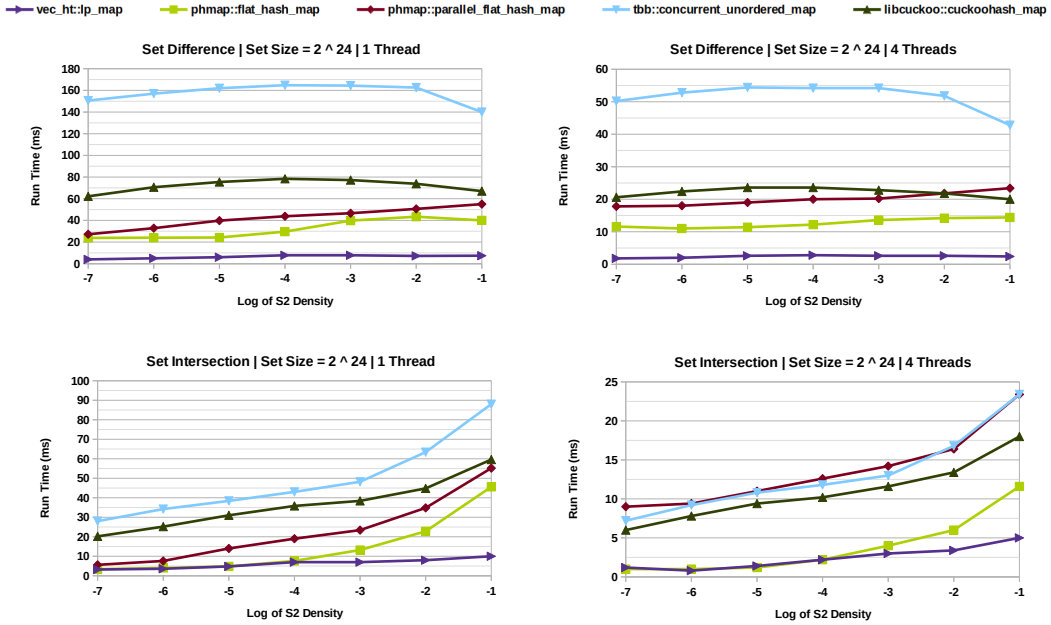

Figure 20 The run time (lower is better) for the set difference and intersection operations by varying the density of the second set on 1 and 4 threads. For both operands (S1 and S2), the size is 2^{24}. For S1, the density is set to 2^{-6}.

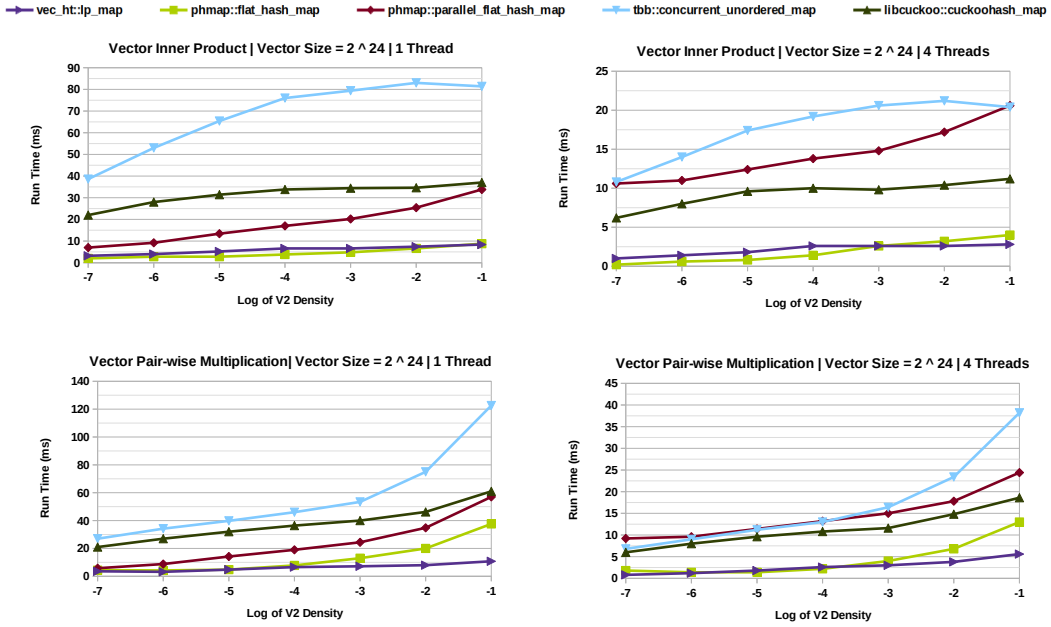

Figure 21 The run time (lower is better) for the vector inner-product and pair-wise multiplication operations by increasing the density of the second vector on 1 and 4 threads. For both operands (V1 and V2), the size is 2^{24}. For V1, the density is set to 2^{-6}.

Table 2 Modified TPC-H queries we used in the experiments.

Query	SQL Code
Q4	`select o_orderpriority, count(*) from orders where exists (select * from lineitem where l_orderkey=o_orderkey and l_commitdate<l_receiptdate) group by o_orderpriority`
Q8	`select o_year, sum(case when nation = 'BRAZIL' then volume else 0 end) / sum(volume) from (select extract(year from o_orderdate) as o_year, l_extendedprice * (1 - l_discount) as volume, n2.n_name as nation from part, supplier, lineitem, orders, customer, nation n1, nation n2, region where p_partkey = l_partkey and s_suppkey = l_suppkey and l_orderkey = o_orderkey and o_custkey = c_custkey and c_nationkey = n1.n_nationkey and n1.n_regionkey = r_regionkey and r_name = 'AMERICA' and s_nationkey = n2.n_nationkey and o_orderdate between date '1995-01-01' and date '1996-12-31') as all_nations group by o_year`
Q12	`select l_shipmode, sum(case when o_orderpriority = '1-URGENT' or o_orderpriority = '2-HIGH' then 1 else 0 end) , sum(case when o_orderpriority <> '1-URGENT' and o_orderpriority <> '2-HIGH' then 1 else 0 end) from orders, lineitem where o_orderkey = l_orderkey group by l_shipmode`

Analytical Queries Use Case. As the last set of experiments, we use the TPC-H benchmark and dataset. The queries we consider satisfy three criteria. (1) The join-build side of the query must result in a large hash table. (2) The join-probe part of the query must be a time-consuming part of it. (3) The build-side hash table can only have integers as keys and values. Based on these criteria we selected a modified version of Q4, Q8, and Q12 (cf. Table 2). Then, we implemented a manually fine-tuned version of them in C++ by taking the query plans of HyPer [4] and using the code generator of SDQL.py [24]. We used `phmap::flat_hash_map` as the competitor for `vec_ht::lp_map`, because it is the hash table of choice behind existing query processing systems that use open source hash tables [24, 28]. We alternate between these two hash tables only in the most time-consuming join operation in the query.

Table 3 shows the results of this benchmark. Our approach performs better than its alternative almost in all these queries. However, in the 4-core setup of Q4, it results in an 11% total performance degradation. In this case, the speedup of probing is still very high (1.33×) but the lack of parallel insertions in Vec-HT resulted in a faster hash table building by the competitor, which is a promising direction for the future.

Finally, it is worth mentioning that the speedups of using Vec-HT for the original Q8 (without modification on the build side to make the hash table larger) are 0.9× and 0.85× for 1- and 4-core respectively. This means that Vec-HT is a perfect choice whenever we are facing a large volume of data resulting in the creation of a large hash table.

Table 3 Performance improvements of using `vec_ht::lp_map` instead of `phmap::flat_hash_map` in the probes of the most time-consuming hash-join of TPC-H queries 4, 8, and 14.

	TPC-H Query					
	Q4		Q8		Q12	
	Total	Probe	Total	Probe	Total	Probe
1-Core Run Time `vec_ht` (ms)	246	102	354	321	260	236
1-Core Run Time `phmap` (ms)	487	261	512	465	718	533
1-Core Total Run Time Speedup	**1.98**×	**2.56**×	**1.44**×	**1.45**×	**2.76**×	**2.25**×
4-Core Run Time `vec_ht` (ms)	151	29	105	83	102	82
4-Core Run Time `phmap` (ms)	134	38	140	107	395	149
4-Core Total Run Time Speedup	**0.89**×	**1.33**×	**1.34**×	**1.29**×	**3.86**×	**1.82**×

8 Conclusion and Future Work

In this paper, we presented Vec-HT, a vectorized hash table that offers fast batch lookups backed by multi-threading, prefetching, and usage of SIMD-vectorization methods. We presented the design decisions for the structure, API, and the optimizations for high-performance batch hash table implementations. We showed the usefulness of our approach by implementing a handful of use cases using Vec-HT. Finally, by running a set of micro-benchmarks on various use case scenarios, we showed that our proposed design performs faster than the state-of-the-art approaches.

In the future, we aim to improve this approach by providing the support for complex keys and values and parallel iterations over such batch hash tables. It is also interesting to apply the current optimizations (especially vertical vectorization) to the other hashing schemes such as Cuckoo [18] and Robinhood [10] hashing. Another promising direction is to use the batch API as a wrapper for traditional hash tables and ordered dictionaries [27] to allow programmers to benefit from the batch processing offered by this API. Finally, one can integrate other SIMD query operators (e.g., selection [19, 16]) and use Vec-HT for a wider range of database analytical queries as well as sparse tensor processing.

References

1 Dpdk. `https://dpdk.org/`.
2 Highwayhash. `arXiv:1612.06257`.
3 Hirola. `https://github.com/bwoodsend/hirola/`.
4 Hyper. `https://hyper-db.de/`.
5 The parallel hashmap. `https://github.com/greg7mdp/parallel-hashmap`.
6 R hashmap. `https://github.com/nathan-russell/hashmap`.
7 Threading building blocks (tbb). `https://github.com/jckarter/tbb`.
8 TPC-H Benchmark . `https://www.tpc.org/tpch`.
9 Alex D Breslow, Dong Ping Zhang, Joseph L Greathouse, Nuwan Jayasena, and Dean M Tullsen. Horton tables: Fast hash tables for {In-Memory}{Data-Intensive} computing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 281–294, 2016.
10 Pedro Celis, Per-Ake Larson, and J Ian Munro. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 281–288. IEEE, 1985.
11 Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3):17–es, 2007.

12 Bin Fan, David G Andersen, and Michael Kaminsky. {MemC3}: Compact and concurrent {MemCache} with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

13 Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 1–14, 2014.

14 Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13, 2011.

15 Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general (?)! *ACM Transactions on Parallel Computing (TOPC)*, 5(4):1–32, 2019.

16 Prashanth Menon, Todd C Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11(1):1–13, 2017.

17 Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

18 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

19 Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508, 2015.

20 Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. *Journal of the ACM (JACM)*, 59(3):1–50, 2012.

21 Stefan Richter, Victor Alvarez, and Jens Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB*, 9(3):96–107, 2015.

22 Kenneth A Ross. Efficient hash probes on modern processors. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1297–1301. IEEE, 2007.

23 Nicolas Le Scouarnec. Cuckoo++ hash tables: High-performance hash tables for networking applications. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, pages 41–54, 2018.

24 Hesam Shahrokhi and Amir Shaikhha. Building a compiled query engine in python. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, CC 2023, pages 180–190, 2023.

25 Amir Shaikhha, Mohammad Dashti, and Christoph Koch. Push versus pull-based loop fusion in query engines. *Journal of Functional Programming*, 28:e10, 2018.

26 Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, pages 12–23, 2017.

27 Amir Shaikhha, Mahdi Ghorbani, and Hesam Shahrokhi. Hinted dictionaries: Efficient functional ordered sets and maps. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

28 Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–33, 2022. doi:10.1145/3527333.

29 Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1907–1922, 2016.

30 Dipti Shankar, Xiaoyi Lu, and Dhabaleswar K DK Panda. Simdht-bench: characterizing simd-aware hash table designs on emerging cpu architectures. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 178–188. IEEE, 2019.

31 Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R Newton. Local: a language for programs operating on serialized data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 48–62, 2019.

32 Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108, 2013.