Datenbank Spektrum (2019) 19:209–218
https://doi.org/10.1007/s13222-019-00329-4

SCHWERPUNKTBEITRAG

Lock-free Data Structures for Data Stream Processing

A Closer Look

Alexander Baumstark[1] · Constantin Pohl[1]

Received: 30 May 2019 / Accepted: 4 October 2019 / Published online: 27 October 2019

Abstract
Processing data in real-time instead of storing and reading from tables has led to a specialization of DBMS into the so-called data stream processing paradigm. While high throughput and low latency are key requirements to keep up with varying stream behavior and to allow fast reaction to incoming events, there are many possibilities how to achieve them. In combination with modern hardware, like server CPUs with tens of cores, the parallelization of stream queries for multithreading and vectorization is a common schema. High degrees of parallelism, however, need efficient synchronization mechanisms to allow good scaling with threads for shared memory access.In this work, we identify the most time-consuming operations for stream processing exemplarily for our own stream processing engine PipeFabric. In addition, we present different design principles of lock-free data structures which are suited to overcome those bottlenecks. We will finally demonstrate how lock-freedom greatly improves performance for join processing and tuple exchange between operators under different workloads. Nevertheless, the efficient usage of lock-free data structures comes with additional efforts and pitfalls, which we also discuss in this paper.

Keywords Concurrent Data Structures · Lock-free · Stream Processing · Parallelism

1 Introduction

The data stream processing paradigm is in the ascendant today since more and more applications require online and real-time processing of their data. Examples are Internet-of-Things devices communicating with each other, the analysis of social media networks, or autonomous stock trading. Stream processing engines (SPEs) like Apache Flink [1], Apache Spark Streaming [24], or StreamBox [14] deal with those applications by providing operators as well as an interface to write queries. The efficient utilization of hardware during query execution is usually hidden from the user. An optimizing component scales out operations and rearranges operators for better performance. The parallelization of operators is mostly realized by multi-threading, where threads share their work to compute query results. Some data structures are inevitably shared between them, for example, a hash table of a join operation or a queue for window semantics. The usage of locks is a common way to deal with simultaneous access to data structures. But even for lightweight locking, where, e.g., a bucket is locked instead of the whole hash table, locks suffer from problems like deadlocks, livelocks, or priority inversion. This limits the degree of parallelism and thus, the scalability of a query [23].

An alternative to locks are lock-free synchronization mechanisms for threads. Lock-free synchronization guarantees that at least one thread makes progress, with a more fine-grained classification of algorithms into lock-free, wait-free, and obstruction-free synchronization. This paradigm shines in cases where thread contention is low, but even for higher contention, lock-free algorithms provide reliable performance at the expense of additional programming complexity. Modern DBMS already utilize lock-free algorithms, like MemSQL [13] or RethinkDB [18]. Available libraries of lock-free data structures also provide a broad range of algorithms and data structures with properties that fit perfectly to the area of stream processing.

✉ Alexander Baumstark
alexander.baumstark@tu-ilmenau.de

Constantin Pohl
constantin.pohl@tu-ilmenau.de

1 Databases and Information Systems Group, TU Ilmenau, Helmholtzplatz 5, 98693 Ilmenau, Germany

In this paper, we focus on the main bottleneck for parallel stream processing, namely joins of data streams and the tuple exchange between threads. In our SPE PipeFabric[1], this exchange is realized by a queue operation that acts as a buffer. The current implementation uses locks for synchronized accesses like reads and writes. Locks are also used for the stream join operation, namely the Symmetric Hash Join (SHJ), where the hash tables are locked for synchronization. The goal of this work is to examine whether or not the benefits of lock-free synchronization can be obtained for stream queries. Therefore, we implement and explore different lock-free data structures in PipeFabric, showing worst-case and average-case performance. Furthermore, this work proposes a lock-free data structure design, equivalent to the hashmap currently used for the SHJ algorithm. The resulting algorithm is compared with another lock-based implementation, using an optimized data structure for shared memory access. Since Pipefabric is based on state-of-the-art algorithms and data streaming concepts, these results can also be transferred to other SPEs.

To summarize, this work provides the following contributions:

1. An improved tuple exchange algorithm for PipeFabric with lock-free synchronization.
2. A lock-free hash map design that supports multiple elements having an equivalent key (called multimap).
3. An improvement of scalability and performance for the SHJ algorithm with lock-free synchronization.
4. An experimental evaluation and demonstration of the proposed structures and algorithms.

2 Data Stream Processing

Data streams are infinite flows of data in form of tuples. Storing them for later processing is often, therefore, no real option. A common way to deal with infinite streams are window operators that keep track of recently received tuples, to delete them when they get outdated after a while. There are different window semantics, like sliding windows or tumbling windows, and also eviction strategies, e.g., invalidating tuples by time or number. Aggregates are then calculated only on tuples inside of that window instead of the whole tuple sequence, saving memory and instructions. For the later experiments in this paper, we use a tuple-based sliding window. Stream queries are often realized as a directed dataflow graph, where operators are connected to their predecessor and successor. An operator then receives tuples from the previous operator, applies its function on it, and forwards the result to the next operator. Most of the SPEs use asynchronous pipelines, distributed over different threads, to execute operations. To pass tuples from one thread to another, explicit tuple exchange is needed. Common approaches use single-producer and single-consumer queues to realize this operation.

[1] Open-source, https://github.com/dbis-ilm/pipefabric.

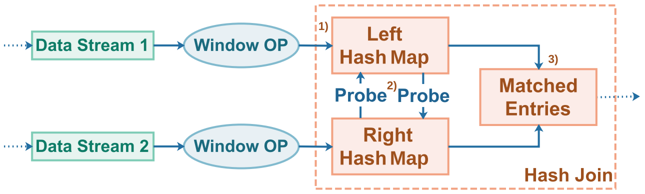

Fig. 1 SHJ algorithm

A commonly used join algorithm in data stream processing is the SHJ which is also available on relational database systems. The SHJ algorithm generates results continuously while tuples arrive from the input streams. Figure 1 sketches the algorithm with two data streams sliced into windows and joined, finally, into a single, consistent stream.

The SHJ algorithm processes two input streams denoted as left and right input. After each arrival of a tuple, either on the left or the right input, it is inserted in the corresponding table, realized as a hash map. In the next step, the currently inserted element is probed for a match against the elements from the other hash map. The matches are joined as new tuples and forwarded to the following operator after the SHJ. If a tuple becomes outdated due to the window semantics, it is deleted from the hash map accordingly. Two hash maps are mandatory in the SHJ for the left and right stream. A further requirement is that the data structure must support duplicate keys. Therefore, the buckets of the hash map must be able to hold multiple elements and the probing also has to iterate through all available elements in the bucket.

2.1 PipeFabric

Since this paper uses PipeFabric as SPE for our measurements, we give a short overview of its design decisions. It is an open-source SPE, written in C++ by the Database and Information Systems Group at the TU Ilmenau. It supports different network protocols like ZeroMQ, MQTT, or AMQP, and can get tuples from Apache Kafka servers or RabbitMQ. To support multi-core CPUs, a partition operator can split the data stream in such a way that each partitioned stream can be processed by individual threads concurrently. Partitions can then be merged into a single stream again. Elements of data streams are represented in PipeFabric as a tuple data structure with low-overhead pointers on them. These tuples can be processed by several operations, like selections and projections, joins, groupings, or aggregations. Another component of PipeFabric is the topology, an interface for the data stream processing pipelines, similar to the implementation from Apache Spark.

In this work, we focus on the queue and SHJ operator, since both have a high potential for performance advantages through lock-free data structures. Currently, the queue operation for tuple exchange between threads ensures thread-safety and consistency with locks on each access. The queue data structure follows the implementation of the queue from the C++ standard library (STL). The hash tables for the SHJ use the unordered multimap implementation from STL, which contains key-value pairs similar to unordered maps, but with the addition of duplicate keys. Internally, the structure of the multimap is a hash map, supporting iterator forwarding with average constant-time complexity. Thread-safety in a concurrent execution is guaranteed with mutual exclusions, where each operation on the data structure is protected with a lock.

2.2 Related Work for Stream Processing

The SHJ algorithm belongs to the first published join algorithms in stream processing [22]. A disadvantage in the conventional concurrent implementation is that the whole hash map or every single bucket has to be locked to obtain consistency. Synchronization of database operations under high contention by using a many-core CPU has gained research interest in recent years. Cheng et al. [2] investigated the state of the art hash join algorithms for main memory databases and their performance on the Xeon Phi Knights Landing (KNL) architecture. Makreshanski et al. [12] created a method for sharing joins for highly concurrent workloads and their performance impact when executing in main memory. In contrast, the present work is focussed on optimizing the concurrency and algorithmic aspects of stream processing algorithms, especially on the tuple exchange and SHJ. Main memory and instruction-level optimization is not part of this work. The paper from Zeuch et al. [25] looked into the current state-of-the-art of SPEs and the resulting performance on modern hardware. They evaluated also lock-free data structures for tuple exchanging, but with the goal of bottleneck identification in different SPEs. Contrarily, this paper is focussed on the SPE PipeFabric and adapts lock-free design principles not only to the tuple exchange algorithm but also to the SHJ algorithm.

3 Lock-free Data Structures

A data structure is a collection of data, providing efficient mechanisms to add, modify, read, or delete entries. The *linked list* is a linear collection of elements, organized as nodes, and is useful for storing data dynamically. Each node points to its successor and possibly to its predecessor, depending on the actual implementation (singly or doubly linked list). Additionally, a linked list manages pointers to the first and last node. A lookup for an element can be made by simply traversing through the *next* pointers. To insert a new element, a new node must be allocated. This new node can be either added to the end of the list (by adding it to the *next* pointer of the last element) or added to the beginning of the list (by adding the current first node to the *next* pointer of the new node). First-node and last-node pointers of the linked list must be refreshed. This scheme can also be applied to the delete operation. Additional pointers (also called express lanes) to other parts of the list can be implemented for fast access, leading to the concept of a skip list.

The *queue* data structure can be implemented like a linked list described previously, with the difference that elements are always added (pushed) to the end of the list. The pop operation is called to retrieve and delete an element, which returns the oldest element in the queue being removed afterward (first-in, first-out principle). Circular arrays (ring buffers) can also be used to implement queues. The successor of the last index element is the first element of the array. A pointer indicates the current position of the top element. The disadvantage of a circular buffer is the limitation in size, whereas the linked list can grow dynamically.

Another data structure next to queues used in this work is the *hash map*. The hash map is an array structure that maps keys to values. To enable fast access to its values, a hash function is used. The hash function computes the index element for a key to a specific bucket that contains a value. If duplicate keys are allowed, the structure is called a multimap, like the unordered multimap in the C++ STL. Each key is ideally mapped into a unique index element in the array. A collision occurs when the hash function computes the same index for multiple different keys. There exist several strategies for handling a hash collision. The used hash map implementation avoids collision by adding multiple keys with the same hash value to the same list.

3.1 Design Principles of Lock-free Data Structures

The conventional way to synchronize data structures is to use locks and mutual exclusions. Lock-free synchronization takes another approach and uses atomic operations, memory barriers, and fences to synchronize and guarantee consistency. Atomic operations are indivisible and uninterruptible instructions [8]. These operations can be compared with transactions from a DBMS. Transactions follow the *ACID* property [6], which can be adapted to atomic operations. With respect to this property, a thread can not observe another thread during an atomic operation, only the state before or after its execution. These instructions are the

```
 1  T* pop() {
 2    T* top_elem = top_;
 3    if(!top_elem)
 4      return null;
 5    while(true) {
 6      T* next = top_elem->next;
 7      if(CAS(top_, next, top_elem))
 8        return top_elem;
 9    }
10  }
```

Listing 1 A lock-free pop operation

points where the action of operation takes place, also called *linearization points*. An example is given in Listing 1 for a pop operation on a stack.

The actual pop operation takes places in line 6. Due to these properties, synchronization can be done without the usage of locks. There are two classes of atomic operations: The first is the class of atomic read and write operations. The other class is for complex atomic read-modify-write operations, like *compare-and-swap* (CAS) or *fetch-and-add*. An equivalent instruction (pair) to CAS for Load/Store architectures is load-linked/store-conditional (LL/SC). The consensus number of the CAS operation is unbounded with the consequence that CAS can implement all other atomic operations [9]. CAS takes three arguments: a memory location, the expected value of the memory location, and a new value. Only if the value of the memory location matches the expected value, the new value will be stored in that memory location. If CAS is successfully executed, it returns true, otherwise false. The failure of a CAS operation indicates that a thread changed the value in the interim, leading to a mismatch of the expected value compared to the value at the memory location. As a consequence, the CAS operation must be restarted to obtain progress and finish the task.

3.2 Common Pitfalls

A common technique is to execute the CAS operation (with refreshed expected values) within a loop until it is successful [20], as seen in Listing 1. Additional consistency checks before an atomic operation can reduce the number of failed operations. For instance, it is useful to check if the current object on which a CAS should be executed is valid (or null) before executing the CAS operation. If it is not valid, the operation can be restarted before the CAS is tried. Similar to the back-off strategies of network protocols that serve to limit the rate of retransmission, back-off strategies can be used to limit the rate of failed CAS operations. The reason for using a back-off strategy is that a high rate of successively failed CAS operations causes unnecessary CPU time, which could be used by other threads to achieve progress. This effect can be observed while executing lock-free algorithms with high numbers of threads. Consequently, the use of a suitable back-off strategy, for instance, a simple waiting strategy, can increase the performance [11].

Another problem in the context of CAS and lock-free synchronization is known as the *ABA problem*. It is defined as a false-positive execution of a CAS-based operation through an unobserved change of a memory location in the interim [3]. A CAS operation cannot consider a change from the value A to B and back to A. Therefore, the CAS operation falsely executes its swap and returns true. Consider a lock-free stack with pop operations implemented with CAS, as in Listing 1, and the initial values (starting with the top pointer) A, B, C. Thread 1 tries to pop an element from the stack and gets interrupted by thread 2 just before the execution of CAS (line 6). Thread number 2 pops the top element A and also the following top element B. After that, thread 2 pushes an element A back to the stack. The stack now contains the elements A and C. When thread 1 resumes its work, the CAS operation will be successful, despite the change in the interim. Applying the ABA problem to stream processing, this behavior can lead to inconsistency and wrong results. An efficient and simple solution is to use tagged pointers [17]. After each successful CAS operation, the tag of the pointer is incremented. Thus, each modification is visible and is tracked automatically. Other approaches use reference counters [20] or hazard pointers [16].

3.3 Related Work for Lock-free Data Structures

One of the lock-free data structures proposed at first is known as Treibers stack [19]. This design uses the CAS operation to implement concurrent stack operations and is the fundament for almost all subsequently proposed lock-free data structures. Another classical lock-free design implemented in a variety of libraries and systems is known as the (multi-producer, multi-consumer) Michael-and-Scott queue [17]. Single-producer and single-consumer queues are also widely implemented in multi-threading libraries, for example, by Threading Building Blocks from Intel (TBB)[2], Facebook Folly[3], or by C++ Boost libraries[4]. These implementations are based on lock-free ring buffer data structures and achieve fast execution performance. In the next section, they are used for the tuple exchange algorithm. Join operations in stream processing use hash maps to probe their entries against others to find a match. Several lock-free designs exist for hash maps, like the hash map by Feldman et al. [4] based on multi-level arrays or by

[2] https://www.threadingbuildingblocks.org/.

[3] https://github.com/facebook/folly.

[4] https://www.boost.org.

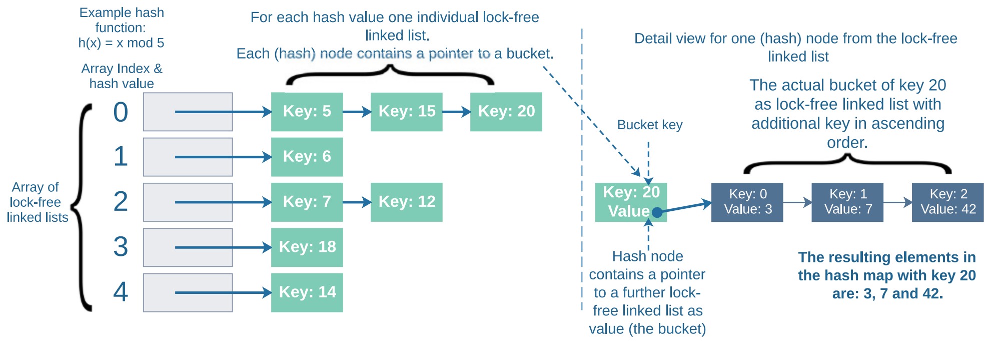
Example hash function: h(x) = x mod 5
Array Index & hash value
For each hash value one individual lock-free linked list.
Each (hash) node contains a pointer to a bucket.
Array of lock-free linked lists
0
1
2
3
4
Key: 5
Key: 15
Key: 20
Key: 6
Key: 7
Key: 12
Key: 18
Key: 14
Detail view for one (hash) node from the lock-free linked list
Bucket key
The actual bucket of key 20 as lock-free linked list with additional key in ascending order.
Key: 20 Value
Key: 0 Value: 3
Key: 1 Value: 7
Key: 2 Value: 42
Hash node contains a pointer to a further lock-free linked list as value (the bucket)
The resulting elements in the hash map with key 20 are: 3, 7 and 42.

Fig. 2 Structure of the proposed lock-free unordered multimap. The hash map consists of an array of lock-free linked lists. Elements with the same hash value are appended as node (called hash node) to the appropriate list. For instance, the keys 5, 15 and 20 are mapped to the same bucket. Each hash node contains a pointer to a further linked list (the actual bucket) that contains the elements with the same key as the hash node

Michael [15] based on linked lists. Some ideas from these lock-free hash maps flow in the design of the proposed approach of this paper.

4 Lock-free Data Stream Processing

This section describes our approach to add lock-free data structures to our SPE PipeFabric, with a focus on the queue and SHJ operator.

4.1 The Queue Operator

As already described, the queue acts as a buffer between threads running operators of a stream query. The publisher thread pushes the results of its subquery into the queue and notifies the subscriber, which takes them out of the queue for further computations. The switch from a lock-based queue to one of the lock-free implementations can be done by adapting the queue access interface correspondingly. For later experiments, we use lock-free queue implementations from Intel TBB, the Boost library, and Facebook Folly, as mentioned in previous Sect. 3.3.

4.2 The SHJ Operator

The lock-free SHJ provides more challenges to solve. This is based on the fact that the bucket structure allows duplicate keys. A lock-free hash map with this bucket property needs a combination of data structures for the hash map itself and its bucket structure. The following lock-free hash map and bucket implementations are based on the lock-free linked list structure from Harris [7]. A reason for using linked lists to implement a hash map is the avoidance of complex collision handling algorithms that reduces the throughput (for instance, due to multiple hashing).

Lock-free Hash Map. The hash map itself is an array of n lists, where n is the size of the hash map with the hash function $h(x) = x \bmod n$. Consistency and thread-safety of the array structure are guaranteed with pre-initialization of each field with the linked list structure.

Lock-free Linked Lists. Each node of the list consists of a key, a value, and a pointer to the *next* node. A key makes it possible to distinguish between entries located at the same index reference. To insert a new element, the *insert* operation computes the hash of the key to find the corresponding list within the array. Then, the new element is inserted into the bucket structure of the node with the corresponding key. *Find* hashes the key, iterates through the corresponding list, compares each key, and returns a pointer to the bucket if the key is found successfully.

The lock-free insert operation of the linked list works as follows. First, a new node is allocated with the key and value. Then, the operation searches for the particular appropriate predecessor and successor node (the new key is inserted between two other keys), which are not logically marked for deletion yet. A deletion with CAS can be done by connecting the predecessor node to the successor node, handled by the next pointer references. The marked node is now disconnected from the list. To find the predecessor, the search operation starts iterating from the head until a key is found that is greater than the searched key. The successor is the next node of the found predecessor. If the tail is reached, the new node can be inserted to the end of the list, executing a CAS operation on the next pointer of the last node (expecting the returned tail pointer of the search operation). Otherwise, the operations continue to check if the two found nodes are adjacent. The operation is tried again if they are not adjacent, which means that another thread inserted a new node in the meantime during this search. Otherwise, the left and right nodes are returned to the insert operation. Then, the new node is inserted between the returned nodes. When the CAS operation fails (meaning another thread inserted or deleted a node) the operation must be started again from the beginning. The bucket structure is based on the same lock-free linked list structure, but

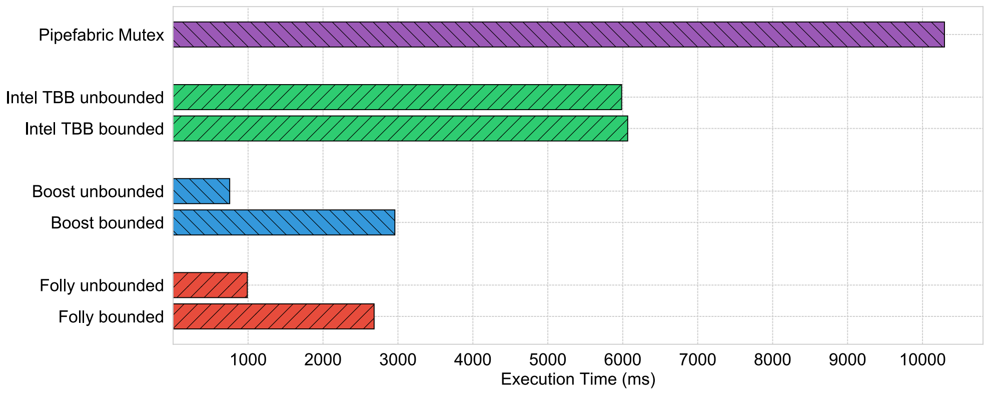

Fig. 3 Tuple exchange algorithm based on blocking and lock-free queues with maximum (unbounded) and fixed size of 1024 tuples (bounded)

with the addition of an atomic size counter that increments on each insertion with an atomic fetch-and-add operation. A new element is inserted into the list with the current value of the size counter as its key (see Fig. 2). The general behavior of this lock-free design (referred to as **LF-LL** for the experiments) corresponds to the STL unordered multimap structure. Equivalent implementations based on lock-free skip lists by Herlihy and Shavit [10] (named **LF-SL**) and a blocking implementation based on the unordered multimap from Intel TBB (named **FL-UM**) are used for a comparison in performance.

5 Experimental Evaluation

In this section, we evaluate our proposed approach with lock-free data structures. We will first specify the hardware and software used. Afterward, we describe our test cases to measure the tuple exchange mechanisms as well as the SHJ performance. We show the impact of different key distributions as well as time spent in different join phases, concluding with a discussion of our results.

5.1 Setup

For the measurements done in this section, an Intel Xeon Phi KNL 7210 processor is used. It has 64 cores with four threads per core due to hyperthreading, limiting threads running in parallel to 256. The base frequency of the KNL runs on 1.3 GHz and can be boosted up to 1.5 GHz (turbo). Each core owns an L1 cache of 32 kB and shares an L2 cache of 1 MB with one other core. This hardware setup allows running experiments for high scalability and concurrency at the same time. The Intel compiler version 17.0.6 is used because it offers better results in our scenario compared with the GNU compiler. Additionally, the code is compiled with the supported AVX-512 instruction set enabled (auto-vectorization), but without further code optimizations with vector intrinsics. The SPE is the previously described PipeFabric.

5.2 Tuple Exchange

In the first experiment, the current queue data structure in PipeFabric with locks for tuple exchange is compared to equivalent lock-free variants from the C++ Boost libraries, Intel TBB, and Facebook Folly. For the first scenario, the producer and consumer thread have to process and exchange five million tuples to simulate an unbounded situation.

Unbounded in this context means that the queue size is set to the maximum possible size. In the second scenario, the size of the lock-free queues is reduced to 1024 elements to show worst-case results for a slow consumer. A naïve waiting back-off strategy is used in case of a full or empty queue. The results in Fig. 3 show that the non-blocking queues outperform the current lock-based implementation from PipeFabric. Each thread in the blocking data structure is executing its operation alternately in a way that no real parallel execution is possible. The amount of time that a thread has to wait for an unlock of the critical section remains unused. The non-blocking implementations use fast atomic load and store instructions, the Boost and Facebook Folly implementations are even wait-free. The fine-grained lock queue from Intel TBB is in the medium performance range in this experiment. A slow consumer can decrease the overall performance but is still faster than the blocking approach. This experiment clearly shows that lock-free queues improve the tuple exchanging procedure significantly.

5.3 The SHJ

This experiment compares the blocking and non-blocking approaches for the SHJ. For this purpose, two tuple generators publish tuples into the left and right stream. An SHJ operation is performed with sliding windows on both streams. The experiment measures the execution time of

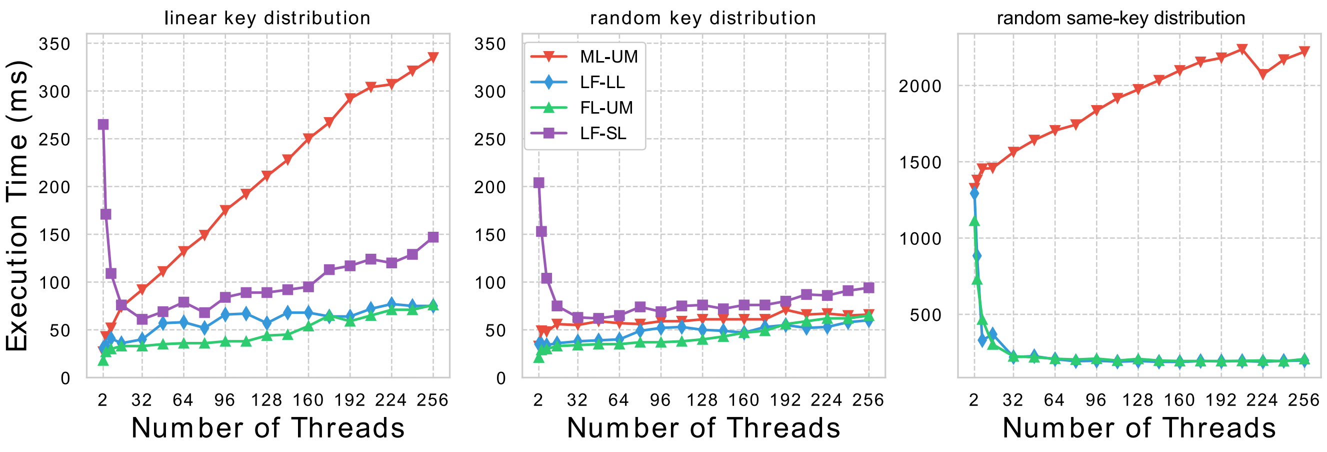

Fig. 4 SHJ experiment with different key distributions. The linear key distribution sends keys in an ascending order to the stream. The random key distribution includes 50% equal keys. The random same-key distribution includes 90% equal keys

the concurrent SHJ (with up to 256 threads) with uniformly distributed tuples. Additionally, this experiment is repeated with different key distributions to analyze the best case (linear same-keys), worst-case (randomized same-keys) behavior and a tradeoff between these two (randomized key). The experiments will be carried out with the following blocking and non-blocking data structures for the SHJ:

1. ML-UM: **M**utex **L**ocking based on the **U**nordered **M**ultimap of C++ STL.
2. LF-LL: **L**ock-**F**ree hash map based on **L**inked **L**ists
3. FL-UM: **F**ine-**G**rained locking based on Intel TBB **U**nordered **M**ultimap.
4. LF-SL: **L**ock-**F**ree hash map based on **S**kip **L**ists.

The results of the measurements found in Fig. 4 clearly show that all approaches differ in performance. When applying the execution with a lower thread number (up to 16 threads), the blocking multimap (ML-UM) is slightly faster than the lock-free structures, except for the random same-key distribution.

The reason for this lies in the implementation of the lock-free hash maps. A lock-free insert operation needs, in general, more instructions than an equivalent lock-based implementation, because it operates within a loop with a CAS. An insert operation needs possibly more attempts (perhaps multiple restarts of the whole operation, dropping the achieved progress) to add an element into the lists, because CAS may fail, unlike the lock-based approach. However, with the increasing number of threads, the execution time of the blocking approach (ML-UM) rises, whereas the execution time of the linked list based implementation (LF-LL) remains on a constant level. The LF-LL implementation can guarantee that at least one thread makes progress, resulting in a higher degree of parallelism and throughput. A performance drop can be observed in the random same-key distribution. In this key distribution, the hash maps must handle more conflicts than in other key distributions (growing linked lists, in case of the lock-free approaches). The execution time increases for the Pipefabric blocking approach, due to the reason that only one thread can handle a collision, whereas

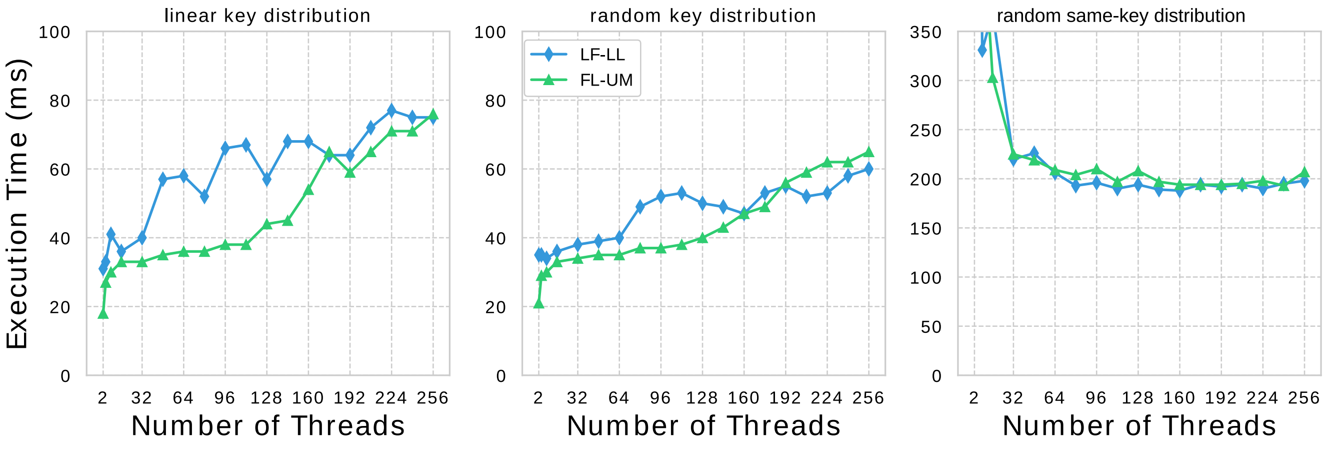

Fig. 5 In-depth experiment with the lock-free and fine-grained locking SHJ and different key distributions

the execution time drops with higher thread numbers for the lock-free and fine-grained implementation.

The reason for the poor performance of the skip-list-based structure (LF-SL) lies in the probabilistic behavior: higher levels (express lanes) are created randomly. Consequently, the threads can not take full advantage of these. At higher numbers of threads, this implementation scales similar to the linked list structure, because these additional layers are created by multiple threads. This approach is not recommended for practical usage due to additional overhead for higher layers and is just shown for comparison.

Another observation is that an optimized solution with fine-grained locks achieves the same or even better performance results than a lock-free implementation (see Fig. 5). Due to the reason that this design uses short critical sections, it achieves a similar degree of parallelism and throughput.

5.3.1 Fine-grained Locking

Comparing the lock-free and fine-grained locking approach in more depth in Fig. 5 shows that the fine-grained solution is slightly faster for the linear and random key distribution. For the random same-key distribution and for thread numbers above 192, both approaches show the same performance. The difference comes with the more non-deterministic execution of lock-free algorithms. In some cases, one execution needs more retries due to conflicts with other threads while executing an atomic operation. With a higher thread number, the waiting time for threads in the fine-grained solution also increases. Therefore, the lock-free approach gets slightly faster at a higher number of threads. It should be mentioned that the lock-free implementations are not further optimized. With additional lock-free techniques, back-off strategies, and other optimizations even better results are possible.

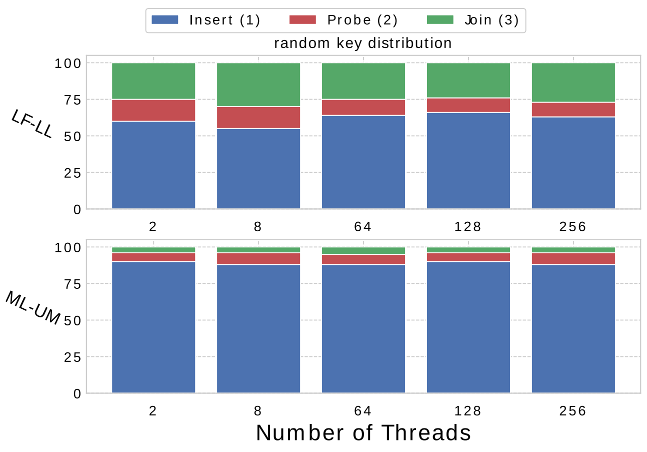

Fig. 6 Execution time proportions of the SHJ phases

5.3.2 Key distribution

The blocking approach is the slowest for the linear key distribution (keys in ascending order) and for the random same-key distribution (randomized keys, multiple occurrences of some keys). This time difference can be explained with the internal structure of the STL unordered multimap and the three phases of the SHJ algorithm. In the first phase, new elements are inserted into the multimap. With a blocking approach, the whole multimap is locked during the join procedure so that only one thread can execute its instructions and other threads are blocked. Internally the STL multimap uses an iterable unordered structure.

The second phase probes the entries of one multimap with the entries of the other. This is implemented by iterating over each item to find matches and gives a large part to the total execution time of the blocking approach because the threads can only execute its work in succession. Fig. 6 compares the execution time proportions of the SHJ phases for both approaches directly. The blocking approach needs more time for the insert and probe phase due to the blocking and slow iteration through the structure. In the lock-free approach (LF-LL), the insert and join phases need most of the time, while the probe phase has only a small fraction of the total execution time. This behavior can be observed for all key distributions. As already mentioned, lock-free algorithms are accompanied by a guarantee for progress for at least one thread. Therefore, inserts are executed with a higher degree of parallelism, which is observable in execution time and scalability.

5.3.3 Hash map size

As mentioned before, the lock-free design needs a pre-initialized array of linked lists for consistency and thread-safety. The array size is the actual size of the hash map. Using a smaller hash map size leads consequently to longer linked lists. The find operation (used in the probe phase of the SHJ) iterates through all nodes of the linked list to find the bucket list of the corresponding key. It is obvious that this parameter affects the performance. A downside of a large hash map is the memory usage due to the prerequisite of a pre-initialized array. This means that the size of the hash map is a trade-off between memory usage and performance.

Figure 7 shows the execution time of the SHJ with different hash map sizes using the LF-LL structure. In this experiment, the keys are generated randomly. The first observation is the jagged curve for the smallest hash map size (16). This can be explained by the (increased) non-deterministic behavior of lock-free algorithms and cache thrashing. Cache thrashing can decrease performance when accessing the data structures with high numbers of threads.

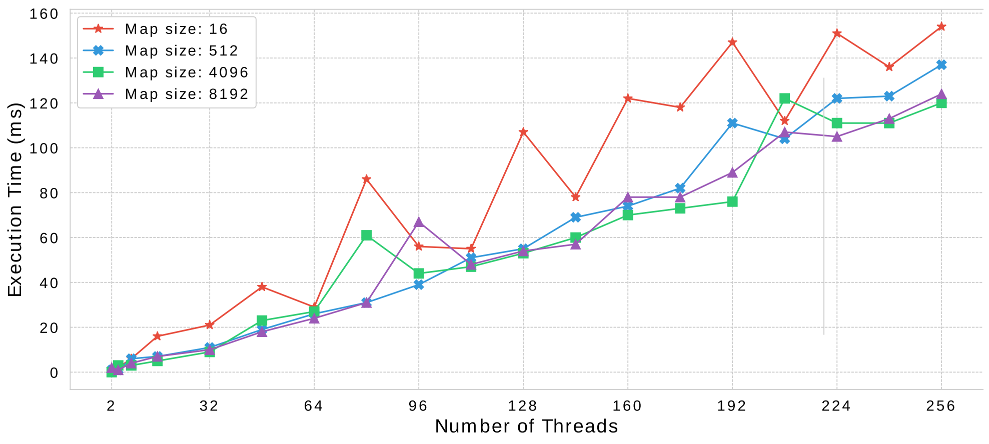

Fig. 7 SHJ with LF-UM. The LF-UM structure is pre-initialized with different sizes. The used hash function is $h(x) = x \bmod n$ where n is the map size

This effect can also be seen in Figs. 4 and 5 in the random same-key distribution, where the execution time drops for a certain thread number and rises for another (for a thread number between 2 and 32). In case of a small hash map size, the linked lists of each hash key grow fast. Consequently, more time is needed to iterate through the lists and probe the entries in the appropriate phase of the SHJ. For larger hash map sizes, the lists are small, so the find operation needs only to iterate through a few nodes. Therefore, a higher hash map size boosts the performance and reduces the latency, due to fewer iteration through the lists.

Comparing the memory footprints of the current PipeFabric implementation ML-UM to LF-LL shows that the lock-free approach creates a memory footprint up to 70% higher than the blocking approach. As already mentioned, the memory usage of the lock-free data structures can be further optimized. There exist several lock-free memory management strategies that can be applied to these structures to improve the memory usage, e.g., interval-based memory reclamation [21] or hazard pointers [16].

6 Conclusion

Comparing the execution time for each approach shows exactly the difference of lock-free and lock-based synchronization. Furthermore, the results of the shown experiments reveal that lock-free implementations can achieve similar results and, at higher thread numbers, even better results than the lock-based implementations. PipeFabric uses a blocking implementation of a concurrent queue to exchange tuples between threads. Every modification on that queue can only be executed by one thread at a time, which is the major disadvantage of blocking implementations. An equivalent lock-free implementation allows simultaneous thread accesses for the data structures. This can boost the tuple exchanging process up to ten times compared to the blocking variant, in cases where the consumer is as fast as the producer. In case of a slow consumer, where the queue is frequently full, the lock-free implementations are still significantly faster. Stream processing benefits from the higher degree of parallelism in tuple exchanging. It leads to higher throughput for stream operations and reduces the overall latency. Another significant performance boost can be achieved with a lock-free SHJ algorithm. Our results have shown that the lock-free implementations are slower at low thread numbers, but are faster and scale very well at high numbers of threads. Reasons for the results with fewer threads are the behavior of the CAS operation, which must be executed multiple times in a loop in order to guarantee consistency for concurrent executions. The ideas of the implemented lock-free data structures can also be used for other stream processing operations, what may be the aim of future work, for example, a lock-free scale join algorithm [5], or window operations.

Lock-free designs can improve the performance of concurrent operations and deliver scalable and robust results for algorithms, which are free from problems like deadlocks and priority inversion. Thanks to these properties, it is possible to achieve reliable latency and throughput in data stream processing by exceeding the performance of blocking designs. However, another observation is that optimized fine-grained locking methods achieve better results at lower thread numbers, due to short critical sections and, consequently, more parallelism. Hence, lock-free synchronization is not the so-called silver bullet in thread synchronization.

References

1. Carbone P et al (2017) State management in Apache Flink®: consistent stateful distributed stream processing. Proceedings VLDB Endowment 10(12):1718–1729. https://doi.org/10.14778/3137765.3137777
2. Cheng X et al (2017) A study of main-memory hash joins on many-core processor: a case with intel knights landing architecture. In: Proceedings of the 2017 ACM on conference on information and

knowledge management CIKM 2017, Singapore, 06.–10.11.2017, pp 657–666 https://doi.org/10.1145/3132847.3132916
3. Dechev D et al (2010) Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In: 13th IEEE international symposium on international symposium on object-oriented real-time distributed computing ISORC 2010, Carmona, Sevilla, 05.–06.05.2010, pp 185–192 https://doi.org/10.1109/ISORC.2010.10
4. Feldman SD et al (2013) Concurrent multi-level arrays: wait-free extensible hash maps. In: International conference on embedded computer systems: architectures, modeling and simulation IC-SAMOS 2013, Agios Konstantinos, 15.–18.07.2013. vol 2013, pp 155–163 https://doi.org/10.1109/SAMOS.2013.6621118
5. Gulisano V et al (2015) Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. In: IEEE International Conference on Big Data IEEE Big Data 2015, Santa Clara, 29.10.–01.11.2015. vol 2015, pp 144–153
6. Härder T, Reuter A (1983) Principles of transaction-oriented database recovery. ACM Comput Surv 15(4):287–317. https://doi.org/10.1145/289.291
7. Harris TL (2001) A pragmatic implementation of non-blocking linked-lists. In: Welch J (ed) Distributed computing DISC 2001, Lissabon, 03.–05.10.2001. Lecture Notes in Computer Science, vol 2180. Springer, Berlin, Heidelberg, pp 300–314 https://doi.org/10.1007/3-540-45414-4_21
8. Hennessy JL, Patterson DA (2006) Computer architecture: a quantitative approach, 4th edn.
9. Herlihy M (1991) Wait-free synchronization. Acm Trans Program Lang Syst 13(1):124–149. https://doi.org/10.1145/114005.102808
10. Herlihy M, Shavit N (2008) The art of multiprocessor programming. Morgan Kaufmann, Amsterdam
11. Khiszinsky M (2015) Lock-free data structures. The evolution of a stack. https://kukuruku.co/post/lock-free-data-structures-the-evolution-of-a-stack/. Accessed 14 June 2018
12. Makreshanski D et al (2018) Many-query join: efficient shared execution of relational joins on modern hardware. VLDB J 27(5):669–692. https://doi.org/10.1007/s00778-017-0475-4
13. Mem SQ (2017) How does MemSQL's in-memory lock-free storage engine work? https://docs.memsql.com/introduction/latest/memsql-faq/#how-does-memsql-s-in-memory-lock-free-storage-engine-work. Accessed 4 Aug 2018
14. Miao H et al (2017) Streambox: modern stream processing on a multicore machine. In: USENIX annual technical conference USENIX ATC 2017, Santa Clara, 12.–14.07.2017, pp 617–629
15. Michael MM (2002) High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the fourteenth annual ACM symposium on parallel algorithms and architectures SPAA '02, Winnipeg, 11.–13.08.2002, pp 73–82 https://doi.org/10.1145/564870.564881
16. Michael MM (2004) Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Trans Parallel Distrib Syst 15(6):491–504. https://doi.org/10.1109/TPDS.2004.8
17. Michael MM, Scott ML (1996) Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the fifteenth annual ACM symposium on principles of distributed computing PODC '96, Philadelphia, 23.–26.05.1996, pp 267–275 https://doi.org/10.1145/248052.248106
18. Rethink D How are concurrent queries handled? https://www.rethinkdb.com/docs/architecture/. Accessed 4 Aug 2018
19. Treiber RK (1986) Systems programming: coping with parallelism. IBM, San Jose (Research report)
20. Valois JD (1995) Lock-free linked lists using compare-and-swap. In: Proceedings of the fourteenth annual ACM symposium on principles of distributed computing PODC 1995, Ottawa, 20.–23.08.1995, pp 214–222 https://doi.org/10.1145/224964.224988
21. Wen H et al (2018) Interval-based memory reclamation. In: Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming PPoPP 2018, Vienna, 24.–28.02.2018, pp 1–13 https://doi.org/10.1145/3178487.3178488
22. Wilschut AN, Apers PMG (1993) Dataflow query execution in a parallel main-memory environment. Distrib Parallel Databases 1(1):103–128. https://doi.org/10.1007/BF01277522
23. Yu X et al (2014) Staring into the abyss: an evaluation of concurrency control with one thousand cores. Proceedings VLDB Endowment 8(3):209–220
24. Zaharia M et al (2012) Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: 4th USENIX workshop on hot topics in cloud computing HotCloud '12, Boston, 12.–13.06.2012
25. Zeuch S, Monte BD, Karimov J, Lutz C, Renz M, Traub J, Breß S, Rabl T, Markl V (2019) Analyzing efficient stream processing on modern hardware. Proceedings VLDB Endowment 12(5):516–530. https://doi.org/10.14778/3303753.3303758