Lock-free Data Structures for Data Stream Processing

Alexander Baumstark[1]

Abstract: The ever-growing amounts of data in the digital world require more and more computing power to meet the requirements. Especially in the area of social media, sensor data processing or Internet of Things, the data need to be handled on the fly during its creation. A common way to handle these data, in form of endless data streams, is the data stream processing technology. The key requirements for data stream processing are high throughput and low latency. These requirements can be accomplished with the parallelization of operators and multithreading. However, in order to realize a higher degree of parallelism, the efficient synchronization of threads is a necessity. This work examines the design principles of lock-free data structures and how this synchronization method can improve the performance of algorithms in data stream processing. For this purpose, lock-free data structures are implemented for the data stream processing engine Pipefabric and compared with current implementations. The result is an improvement for the tuple exchanging between threads and a significant improvement for the symmetric hash join algorithm based on lock-free hash maps.

Keywords: Concurrent Data Structures, Lock-Freedom, Stream Processing, Parallelism

1 Introduction

This work investigates the design principles of lock-free data structures and examines their potential use for data stream processing. The article from [SÇZ05] provide eight requirements for (real-time) data stream processing. Three of these requirements are directly dependent on the system architecture, the effectiveness of the used algorithms and the performance. Stream processing requires algorithms that produce constant progress as a minimum. Due to the fact that the data appears mostly in form of unbounded data streams and high costs for storage operations, the algorithms must be able to process data on the fly. Other requirements are stable and robust algorithms that are not prone to errors, because high availability is a must-have within data stream processing. Modern approaches take advantages of the multithreading paradigm to achieve this requirement, for instance, with concurrent operations on partitioned streams. The key component of such stream processing algorithms are concurrent data structures, for example, linked list data structures for the stream partitioning, where selected elements of the original stream are stored. In order to obtain consistency and a high degree of parallelism efficient synchronization methods are needed. Conventional techniques of thread synchronization make use of blocking

[1] TU Ilmenau, Databases and Information Systems Group, Helmholtzplatz 5, 98693 Ilmenau,
alexander.baumstark@tu-ilmenau.de

doi:10.18420/btw2019-ws-22

mechanisms like locks and mutual exclusions (*lock-based*). The major disadvantage of these methods is that they can suffer from problems like deadlocks, livelocks or priority inversion. Since the critical sections of shared resources cannot be executed in parallel by multiple threads, the possible degree of parallelism is decreased.

A different technique is thread synchronization without locks, called non-blocking synchronization. Basically, there are three classes of non-blocking methods: *obstruction-free*, *lock-free* and *wait-free*. The difference between these classes lies in the guarantee they provide for the progress. In short, lock-free synchronization guarantees that at least one thread makes progress, whereas wait-free makes sure that all threads do so. Obstruction-free synchronization is the weakest class and can only guarantee that an isolated thread makes progress. None of the mentioned problems of lock-based synchronization can occur with non-blocking implementations. This can lead to a higher degree of parallelism which may result in a performance gain. Certain modern database systems already use lock-free algorithms in order attract with their achieved performance ([Re], [Me17]). The goal of this work is to examine whether or not the benefits of lock-free synchronization are attainable in data stream processing. The primary research questions of this work ([Ba18]) can be summarized as follows: **(1)** What design principles exist for lock-free data structures? **(2)** For which data structures does a lock-free design exist? **(3)** How does lock-free synchronization affect the overall performance, especially for the use case data stream processing? Can this method fulfill the requirement of low latency and high throughput? The data stream processing engine Pipefabric[2] is used for benchmarks, in order to give an answer to the third research question.

To summarize, we make the following contributions:

1. We improved the tuple exchange algorithm in Pipefabric with lock-free synchronization.
2. We proposed a lock-free hashmap design that supports multiple elements with equivalent key, similar to the C++ unordered multimap structure.
3. We improved the scalability and performance of the symmetric hash join algorithm in Pipefabric.

2 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 [HP06]. These operations can be compared with transactions from database systems. Transactions follow the *ACID* property [HR83], which can be adapted to atomic operations. The ACID property guarantees that every

[2] `https://github.com/dbis-ilm/pipefabric`

operation must be uninterruptible (atomicity) and that every operation from a consistent state is followed by a consistent state too (consistency). Operations are executed concurrently but the effect of these is the same as if the operations would be executed sequentially (isolation). Each operation remains after it has committed (durability). Due to these properties, synchronization can be done without the use 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)[3] or *fetch-and-add*.
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 with the expected value, the new value will be stored in the memory location. If the compare-and-swap is successfully executed, it returns true, otherwise false. The failure of a CAS operation means that a thread changed the value in the interim, so the expected value does not match with the value of the memory location. A common technique is to execute the CAS operation (with refreshed expected values) within a loop until it is successful. [He91] has shown that the consensus number of the CAS operation is unbounded with the consequence that CAS can implement all other atomic operations.
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 operation causes unnecessary CPU time, which could be used by other threads to make progress. Consequently, the use of a correct back-off strategy can increase the performance of a lock-free data structure [Kh15]. An example is the *elimination back-off strategy* for a lock-free stack [HSY04]. It is based on the following observation. If a pop operation follows a push, the state of the stack does not change. Therefore, a pair of push and pop operations can meet at a different location to exchange data, without performing actions on the stack.

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 [DPS10]. 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 a true as a result. It is clear that this behavior, caused by the ABA problem, can lead to inconsistency and must be prevented. [MS96] described a efficient solution to the ABA problem with tagged pointers. After each successful CAS operation the tag of the pointer is incremented and each modification can be considered. Other approaches use reference counters described by [Va95] or hazard pointers [Mi04].

3 Lock-free Implementations

Stream processing operations rely on concurrent data structures. One of the research questions of this work is: For which data structures does a lock-free design exist at all? The

[3] An equivalent instruction (pair) for Load/Store architectures is load-linked/store-conditional (LL/SC).

answer is simple: There are no real restrictions. Several thread-safe lock-free data structure designs exist for almost all data structures. [Tr86] published a simple lock-free stack design. It is assumed that this is the first published non-blocking implementation. Another classical lock-free design that is implemented in a variety of libraries and systems, is known as the (multi-producer, multi-consumer) Michael and Scott queue [MS96]. Single-producer and single-consumer queues are widely implemented by multi-threading libraries, for example, Intel's Threading Building Blocks (*TBB*)[4], Facebook *Folly*[5] or by C++ *Boost* Libraries[6]. These implementations are based on lock-free ringbuffer data structures and achieve incredible fast execution performance.
Join operations in stream processing use hash maps to probe their entries against others to find a match. A disadvantage in conventional concurrent implementation is, that the whole hash map has to be locked to obtain consistency. Several lock-free designs exist for hash maps, like [FLD13] based on multi-level arrays or [Mi02] based on linked lists, only to name a few. [BP12] published a lock-free implementation of a B^+-tree which is an alternative to blocking lock-coupling techniques. The next section examines lock-free data structures in the use case of the data stream processing engine Pipefabric.

4 Use Case: Data Stream Processing

As already mentioned, the key requirements for data stream processing are high throughput and low latency. The goal of this section is to examine whether or not these requirements are more attainable with lock-free synchronization. Due to the fact that lock-free synchronization allows theoretically a higher degree of parallelism, it is expected that algorithms that rely on this technique achieve better performance results than their equivalent blocking approaches.

4.1 Pipefabric

Pipefabric is a data stream processing engine, developed by the Database and Information Systems Group at the TU Ilmenau. It is open source, written in C++, supports different network protocols like ZeroMQ, MQTT or AMQP and can get tuples from Apache Kafka servers or RabbitMQ. For multi-core machines there are several operations available in order to enhance the stream processing. A partition operator can split the data streams, so that each partitioned stream can be processed concurrently. Sub-stream can be merged into a single-stream again. The supported window operations are the tumbling and sliding window. Elements of data streams are represented in Pipefabric as a tuple data structure. These tuples and their components can be processed with several operations. Another component of Pipefabric is the topology, an interface for the data stream processing pipelines, similar to the implementation from Apache Spark.

[4] https://www.threadingbuildingblocks.org/
[5] https://github.com/facebook/folly
[6] https://www.boost.org/doc/libs/1_63_0/doc/html/lockfree.html

4.2 Benchmark System

For the benchmarks that are to be done in this section, an Intel Xeon Phi KNL 7210 processor is used with 64 cores and four threads for each core. The base frequency runs on 1.3 GHz and can boost up to 1.5 GHz (turbo). Each core owns an L1 cache of 32kB. This hardware setup allows to run a benchmark for high scalability and concurrency at the same time. The Intel compiler version 17.0.6 is used because it offers better results in benchmark scenarios, compared with gcc. Additionally, the code is compiled with the supported AVX-512 instruction set, but without further code optimization that would take full advantage of these instructions.

4.3 Tuple Exchanging

Sometimes data needs to be exchanged between two threads, for example, in exchanging information of the status or tuples for partitioning. An approach to realize this is to implement it with a buffer, with a single reader and writer. The underlying data structure of the current implementation is the STL queue, protected with locks and condition variables. In the first benchmark, the current queue data structure for tuple exchanging is compared with equivalent lock-free variants from the C++ Boost libraries, Intel's TBB and Facebook Folly.

For the first scenario, the producer and consumer has to process five million tuples/elements in order to simulate an unbounded situation. In the second scenario, the maximum size of the lock-free queues is reduced to the size 1024, in order to show results with realistic parameters. A naive waiting back-off strategy is used in case of a full or empty queue.

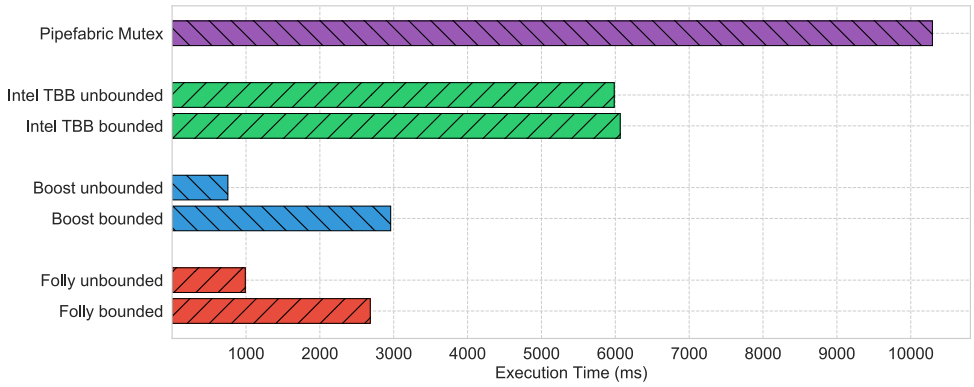

Figure 1: SPSC queue benchmark: execution time, unbounded and bounded

The results in Figure 1 show clearly that the non-blocking queues outperform the lock-based implementation from Pipefabric. Each thread in the blocking technique is executing its operation alternately in a way that no real parallel execution is possible and the amount of time that a thread waits for an unlock of the critical section is unused. The non-blocking implementations use fast atomic load and store instructions, the Boost and Facebook Folly implementations are even wait-free. Intel TBB's fine-grained lock queue is in the medium

range in this benchmark. A slow consumer, like in the bounded situation, can decrease the overall performance but is still faster than the blocking approach. It is recommended to implement the Boost queue for the tuple exchange, because of its speed and the reason that the Boost library is already used in Pipefabric. This benchmark shows clearly that lock-free queues can improve the tuple exchanging procedure significantly. The next benchmark examines in which way it is attainable in complex stream processing algorithms like the symmetric hash join.

4.4 Symmetric Hash Join

A commonly used join algorithm in data stream processing is the symmetric hash join, which is also available on relational database systems. The symmetric hash join algorithm for stream processing continuously generates results while tuples from the streams arrive. Figure 2 shows the idea of a symmetric hash join algorithm with two data streams sliced into windows and joined after the algorithm into a single stream.

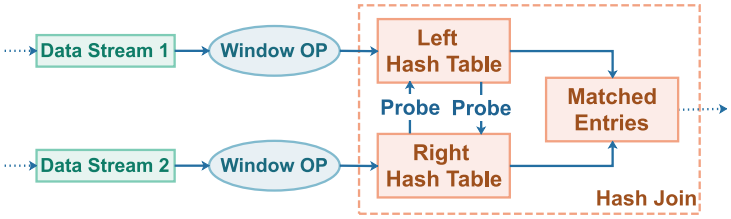

Figure 2: Symmetric hash join operation.

The symmetric hash join 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 hash map. In a next step the hash maps probe their entries against the others for a match. Entries with no match are removed from the hash map. The gained entries with a match are forwarded to the following operator as a single data stream.

For a symmetric hash join algorithm, two hash tables are mandatory for the left and right stream elements. Another requirement is that multiple stream elements are mapped to the same key. Therefore, the buckets of the hash map must be able to contain multiple elements, and the probing must also iterate through all available elements in the bucket. The algorithm is organized in three steps: *(1)* Insert the tuples in the corresponding hash table, or remove them if they are outdated, *(2)* probe for possible join partner in the other hash table and *(3)* the actual join of the tuples.
The implementation of the symmetric hash join algorithm in Pipefabric is based on the STL data structure unordered_multimap. This structure is an STL container, that contains key-value pairs, similar to the unordered_map structure but with the addition that elements can have equivalent keys. The internal structure of the multimap is a hash map which supports forward iterators with an average constant-time complexity. In order to guarantee thread-safety in a concurrent execution, each operation of the data structure is protected with a lock. The lock-free symmetric hash join is more challenging to realize. This relies on

the bucket structure, which allows that multiple values have the same hash key. A lock-free hash map with this bucket property needs combined data structures for the hash map and bucket structure. The following lock-free hash map and bucket implementation is based on the lock-free linked list structure by [Ha01].

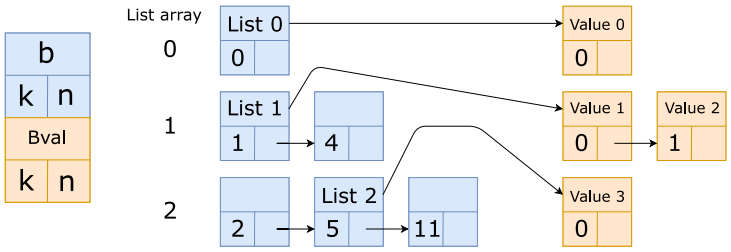

Figure 3: Lock-free hashmap supporting same key elements
A node in the hashmap list contains a key (k), pointer to the next node (n) and a pointer to the bucket as value (b). The bucket node contains the actual value (Bval).

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. The hash map itself is an array of n lists, where n is the size of the hash map with the hash function $h(x) = x \mod n$. In order 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. Additionally, the bucket structure is based on the same lock-free linked list structure 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 Figure 3).

Head and *Tail* pointers are used to iterate through all elements, by swinging to the *next* element of the node. The general behavior of this lock-free design (named Lock-free/Linked List in Figure 4) corresponds to the STL unordered multimap structure. Equivalent implementations based on lock-free skip lists (named Lock-free/Skip List in Figure 4) and a blocking implementation based on the unordered multimap from Intel TBB are used for reference in the benchmark.

In the benchmark of Figure 4, two tuple generators publish tuples into the left and right sliding window. The benchmark measures the execution time of the concurrent symmetric hash join (with up to 256 threads) with constant distributed 10.000 tuples in total. The buckets are preallocated in order to measure the relevant parts of the symmetric hash join.

The benchmark result given in Figure 4 shows clearly that these two approaches differ in performance. When applying the execution with two threads, the blocking hash map is slightly faster than the lock-free one. The reason for this lies in the implementation of the lock-free hash map. A lock-free insert operation generally needs more instructions than an equivalent lock-based implementation, because it operates within a loop with a CAS

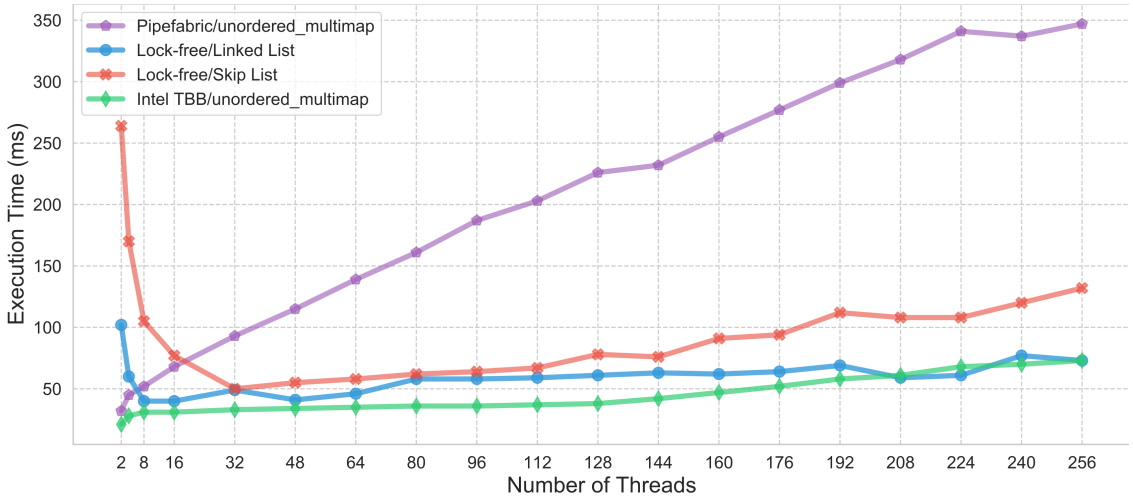

Figure 4: Symmetric Hash Join Benchmark: Execution Time

instruction. An insert operation of a lock-free data structure needs possibly more attempts to add an element into the lists, because CAS may fail, unlike the lock-based approach.

With the increasing number of threads the performance of the lock-based approach decreases drastically. This observation is based on the fact that only one thread can access a critical section. However, the lock-free implementation can guarantee that at least one thread makes progress resulting in a higher degree of parallelism and throughput. A benefit of the implementation based on singly linked lists is, that a constant number of stream elements is processed in an approximately constant time. Another observation of this benchmark is, that an optimized solution with fine-grained locks achieves the same or, in some situations, even better performance results than a lock-free implementation. Due to the reason that this design uses small critical sections, it achieves a similar degree of parallelism and throughput. 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.
The reason for the poor performance of the skip list-based structure lies in the probabilistic behavior: higher levels (express lanes) are created randomly. Consequently, the threads can not take full advantage of these in the worst case. At higher thread numbers, this implementation scales similar to the linked list structure, because the additional layers are created by multiple threads. This approach is not recommended for practical usage and just shown for reference, due to additional overhead for the higher layers.

5 Conclusion

The results of the benchmarks show 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 in order 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 that every thread can access the data structure simultaneously. This can boost the tuple exchanging process up to a a tenth, compared to the blocking variant, if 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. The stream processing can benefit from the higher degree of parallelism at the tuple exchanging, which lead to a higher throughput for stream operations, for instance, window operations or joins.

Another significant performance boost can be achieved with a lock-free symmetric hash join operation. The benchmark results have shown that the lock-free implementations are slower at lower thread numbers but faster and scale very well at higher thread numbers. Reasons for the results at lower thread numbers are the additional consistency checks before an actual stream processing operation takes place. The implemented lock-free data structures can also be used for other stream processing operations in order to improve the degree of parallelism, for example, the scale join or for the window operations. However, another observation is that optimized fine-grained locking methods achieve better results at lower thread numbers, due to small critical sections and consequently more parallelism. Hence, lock-free synchronization is not the so-called silver bullet in thread synchronization.

To summarize it all, lock-free designs can improve the performance of concurrent operations and deliver scalable and robust 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 and exceed the performance of blocking designs. This work has shown that lock-free implementation can fulfill the demands of data stream processing algorithms.

References

[Ba18] Baumstark, A.: Lock-free Data Structures for Data Stream Processing, Bachelor's Thesis, TU Ilmenau, Aug. 17, 2018.

[BP12] Braginsky, A.; Petrank, E.: A lock-free B+ tree. In: Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures - SPAA '12. ACM Press, 2012.

[DPS10] Dechev, D.; Pirkelbauer, P.; Stroustrup, B.: Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs. In: 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. IEEE, 2010.

[FLD13] Feldman, S.; LaBorde, P.; Dechev, D.: Concurrent multi-level arrays: Wait-free extensible hash maps. In: 2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). IEEE, July 2013.

[Ha01] Harris, T. L.: A Pragmatic Implementation of Non-blocking Linked-lists. In: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 300–314, 2001.

[He91] Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13/1, pp. 124–149, Jan. 1991.

[HP06] Hennessy, J. L.; Patterson, D. A.: Computer Architecture: A Quantitative Approach, 4th Edition. Morgan Kaufmann, 2006, ISBN: 0-12-370490-1.

[HR83] Haerder, T.; Reuter, A.: Principles of transaction-oriented database recovery. ACM Computing Surveys 15/4, pp. 287–317, Dec. 1983.

[HSY04] Hendler, D.; Shavit, N.; Yerushalmi, L.: A scalable lock-free stack algorithm. In: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures - SPAA '04. ACM Press, 2004.

[Kh15] Khiszinsky, M.: Lock-Free Data Structures. The Evolution of a Stack, Feb. 24, 2015, URL: https://kukuruku.co/post/lock-free-data-structures-the-evolution-of-a-stack/, visited on: 06/14/2018.

[Me17] MemSQL: How does MemSQL's in-memory lock-free storage engine work?, 2017, URL: https://docs.memsql.com/introduction/latest/memsql-faq/#how-does-memsql-s-in-memory-lock-free-storage-engine-work, visited on: 08/04/2018.

[Mi02] Michael, M. M.: 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. ACM Press, 2002.

[Mi04] Michael, M.: Hazard pointers: safe memory reclamation for lock-free objects. IEEE Transactions on Parallel and Distributed Systems Record 15/6, pp. 491–504, 2004.

[MS96] Michael, M. M.; Scott, M. L.: 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. ACM Press, 1996.

[Re] RethinkDB: How are concurrent queries handled?, URL: https://www.rethinkdb.com/docs/architecture/, visited on: 08/04/2018.

[SÇZ05] Stonebraker, M.; Çetintemel, U.; Zdonik, S.: The 8 requirements of real-time stream processing. ACM SIGMOD Record 34/4, pp. 42–47, Dec. 2005.

[Tr86] Treiber, R. K.: Systems programming: Coping with parallelism. IBM Research Center, 1986.

[Va95] Valois, J. D.: Lock-free linked lists using compare-and-swap. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing - PODC '95. ACM Press, 1995.