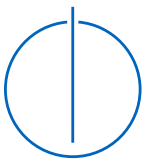
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementing an Efficient Shuffle Operator for Streaming Database Systems

Jonas Ladner



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementing an Efficient Shuffle Operator for Streaming Database Systems

Implementierung eines effizienten Shuffle-Operators für Streaming-Datenbanksysteme

Author:	Jonas Ladner
Examiner:	Prof. Dr. Thomas Neumann
Supervisor:	Maximilian Rieger M.Sc.
Submission Date:	17.02.2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 17.02.2025 Jonas Ladner

Acknowledgments

Abstract

Modern streaming database systems rely on efficient data partitioning to achieve scalability and high performance across processing nodes. Partitioned data shuffling is a crucial operation, as it is used to prepare and distribute data for further processing on distributed systems.

This thesis purposes and evaluates various partitioning implementations by simulating real-world usage of the shuffle operator. The implementations process incoming tuple batches and partition them into output buckets, which are based on slotted pages and can be passed to subsequent operators. The evaluation of the implementations is based on their performance, scalability and memory consumption.

The results demonstrate that using a lock- and Software Managed Buffer (SMB)-based approach yields the best overall performance, offering both efficiency and ease of implementation. Notably, the proposed locking mechanism minimizes the duration of holding a lock, ensuring minimal contention and contributing to the approach's superior performance.

TODO: quantify results
"As a result we implement a ... that is ?X faster than ..."
At the end: "Our approach uses software managed buffers, locking, ..."

Contents

1 Introduction

1.1 Motivation

Growing demand for real-time data analysis and increasing data volume create significant challenges [1–3]. Distributed and parallel systems like stream processing engines address these issues by distributing tasks across worker nodes [1, 4]. Data shuffling prepares and distributes tuples among worker nodes [5]. As a core streaming component, the shuffle operator must achieve high throughput and low latency to support efficient downstream processing. This thesis addresses these challenges by focusing on the most efficient implementations of the shuffle operator.

1.2 Streaming processing engines

Streaming processing engines are designed to process data as soon as it arrives rather than relying on traditional pre-computed information and index structures [1]. Their core operations include partitioning and distributing incoming traffic across worker nodes. The distribution process is frequently based on a partitioning function. These partitions can then improve the performance of further operators by ensuring the data locality of interdependent tuples [3, 6]. Data locality within the worker node is crucial for maintaining performance and scalability in large-scale deployments.

1.3 Shuffle operator

The shuffle operator provides a partitioned distribution of tuples, enabling downstream operators to leverage the data locality of partitioned data blocks. For instance, the throughput of the join operator can significantly be improved when tuples assigned to the same hash bucket are shuffled to the same worker node [3]. Implementing a shuffle operator involves addressing challenges like memory consumption, latency, and scalability. This thesis proposes and evaluates different implementations of the shuffle operator, focusing on their efficiency and performance.

1.4 Slotted pages

Slotted pages are a common way to store variable-size tuples within fixed size memory blocks [7]. These fixed size memory blocks can then be either stored on disk or easily be sent to worker nodes.

Typically, the pages consist of three sections: metadata, slots and a variable-size data section. The metadata area contains information like an identifier for the page, what fields the tuples have and the amount of tuples on this page. This fixed-size metadata section is then followed by the slot section. A single slot contains the fixed-size properties, the variable-size length and its start offset in the variable-size data section. In contrast to the previous two sections, the variable-size data section grows from the end of the page towards the slot section of the page.

1.5 Problem setting

The key contribution of this thesis is the creation and evaluation of the most efficient, multithreaded implementations of the shuffle operator. In order to simulate the real-world usage of the shuffle operator in a streaming system, the following three-step shuffle-simulation is proposed:

1. Tuple generation: The tuples are generated in a batched manner using a pseudo-random generator. Each implementation requests the ad-hoc generation of tuples, which contain a 32-bit key field and an optional variable-size data field.

2. Data shuffle: The requested, random-generated tuples are then processed using the different implementations and stored within partition buckets. Each partition bucket consists of slotted pages, where the tuple of this partition are stored.

3. Storing tuples on slotted pages: The implementations range from thread-local to shared slotted-page write-out strategies. The implementations using a shared write-out policy, can then be further categorized into locking and lock-free approaches.

This simulation is close to the real world usage of the shuffle operator, as streaming systems work on incoming tuple batches that are not materialized like in traditional relational databases.

Further, we use slotted pages as a communication format between nodes. This is an efficient and widely used format to transport tuples within fixed size data chunks. In contrast, sending each tuple as soon as it is processed, creates significant network overhead and makes downstreaming processing less efficient.

While the implementations are optimized for the simulated process above, the underlying algorithms can be transferred to any streaming system, that forwards data using slotted pages.

2 Related work

2.1 Radix Partitioning

2.2 Partitioned Joins

2.3 Software Managed Buffers

3 Implementations

This section explains the implementations and how to implement them efficiently. As the shuffle operator-simulation is based on fixed-size tuples, the explanations of the following implementation, are based on fixed-size tuples as well.

3.1 Partitioning

We generate our tuples using an pseudo-random number generator. To distribute the tuples into partitions, we use the following function:

$$f(t) = t.\text{key } \% \text{ partitions} \tag{3.1}$$

As the modulo calculation is quite expensive, it we use the following partitioning function, when the count of partitions is a power of two:

$$f(t) = t.\text{key } \& \text{ (partitions - 1)} \tag{3.2}$$

3.2 Slotted Pages

Slotted pages store their information on fixed size memory blocks, that are split up in three sections: a fixed-size header, slots and a data section. We are using slotted pages with a total size of 5 MiB per page.

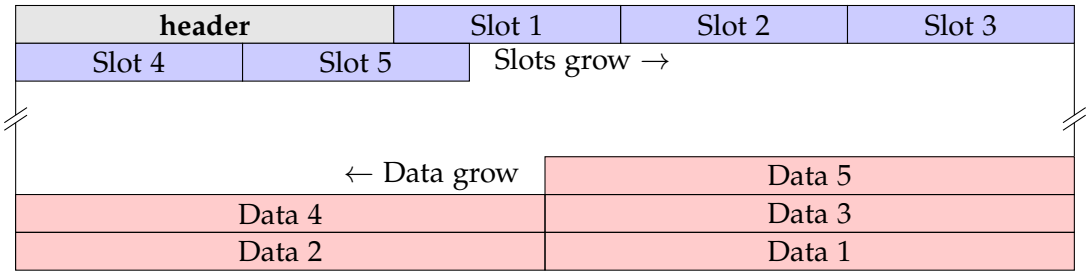


Figure 3.1: Slotted Page grow visualization

In our implementation, we only store the tuple count in the header. To construct our shuffle-simulation close to the real world usage, we split up each tuple. In the slot information, we store the 4-Byte key together with data offset and length information. The remainder of the tuple is then stored in the data section at the end of each page.

As we are using fixed-size tuples in our simulation, we only need to have the tuple slot index to be able to store the tuple on the page. In constrast, when dealing with variable size tuples, slot index and data offset are needed to store a tuple.

3.3 Slotted Page Managers

As some implementation share the same tuple write-out strategy, we propose the used write-out strategies here and reference them in the following explanations of the concrete implementations.

To further simplify the implementations, we intialize each partition with an empty slotted page. This significantly reduces the complexity of the page manager implementations.

3.3.1 Lock-based Page Manager

For each partition, we use a single lock and a vector for storing the slotted pages. As

Algorithm 1: Lock-based Page Manager insert_tuple Algorithm

input : `tuple`: The tuple to be inserted, `partition`: The target partition index
output: Tuple inserted into the appropriate slotted page of the specified partition.

```
1 function insert_tuple(tuple, partition)
2 Acquire lock on partition_locks[partition]
3 if pages[partition].back().add_tuple(tuple) then
4 |   // Tuple added successfully
5 else
6 |   add_page(partition)
7 |   pages[partition].back().add_tuple(tuple)
8 end
9 Release lock
```

can be seen in Algorithm 1, the lock-based insertion process is straightforward. The insertion on a given slotted page, can only fail if the page is full. This can easily be checked by reading the tuple count in the metadata section of the slotted page. If the

current page is full, we just allocate and append a new slotted page to the page vector of this partition.

Tuple insertion in batches

Similarly, we can further optimize the write-out by using tuple-batches. In Algorithm 2,

Algorithm 2: Lock-based Page Manager insert_tuple_batch Algorithm

input : `tuples`: The tuple-batch to be inserted
`partition`: The target partition index
output: Tuples inserted into one or more slotted pages of the specified partition.

```
1  function insert_tuple_batch(tuples, partition)
2  Acquire lock on partition_locks[partition]
3  for tuple : tuples do
4      if pages[partition].back().add_tuple(tuple) then
5          // Tuple added successfully
6      else
7          add_page(partition)
8          pages[partition].back().add_tuple(tuple)
9      end
10 end
11 Release lock
```

we reuse the tuple insertion logic from Algorithm 1 but acquire the partition lock only once for the entire insertion process. Since acquiring and releasing the lock is expensive, this optimization significantly improves performance in multi-threaded scenarios.

3.3.2 Lock-free Page Manager

As holding a lock of a partition denies a second thread to also write out tuples, we propose a lock-free implementation. In comparision to the lock-based variant, we now have to store our slotted pages in a pointer-stable data structure. This is necessary to ensure threads can work simultanious, while a thread adds a new slotted page. Furthermore, we have to edit the slotted page metadata using compare-and-exchange operations to avoid losing writes from other threads.

In order to gather the information, where we can write a tuple, we increment the tuple count using compare-and-exchange. This index then acts as our location, where our tuple is placed on the page. In Algorithm 3, we also add an condition to stop attempting to further increase the count of tuples on the page, if the maximum is

Algorithm 3: Lock-free Slotted Page increment_and_fetch_opt_write_info Algorithm

```
1 function increment_and_fetch_opt_write_info()
2 current_tuple_count = header->tuple_count.load();
3 while !header->tuple_count.compare_exchange_strong(current_tuple_count,
    current_tuple_count + 1) do
4 |   if current_tuple_count >= get_max_tuples(page_size) then
5 |   |   return std::nullopt
6 |   end
7 end
8 return {page_data.get(), page_size, current_tuple_count}
```

reached. This ensures that threads move to the next allocated page. Given the index, where the tuple is placed, we also append a pointer of the start of the page and the page size. This ensures we can write the page without requiring any further information.

Algorithm 4: Lock-free Page Manager insert_tuple Algorithm

```
  input :tuple: The tuple to be inserted, partition: The target partition index
  output:Tuple inserted into the appropriate slotted page of the specified partition.
1 function insert_tuple(tuple, partition)
2 wi = current_page[partition].load()->increment_and_fetch_opt_write_info()
3 while wi == std::nullopt do
4 |   wi = current_page[partition].load()->increment_and_fetch_opt_write_info()
5 end
6 if wi.tuple_index == LockFreeSlottedPage::get_max_tuples() - 1 then
7 |   add_page(partition)
8 end
9 LockFreeSlottedPage::add_tuple_using_index(wi, tuple)
```

Using the Algorithm 3, we retrieve the information to write the tuple in Algorithm 4. We read from an atomically-stored pointer to our current page, until we are assigned an index on a slotted page. If we are writing the last tuple on the page, we add a new page to this partition. This is necessary to create a unqiue condition, when a new page has to be allocated. With that write information at hand, we can write the tuple onto the slotted page.

3.3.3 Histogram-based Page Managers

The following page managers are based on histograms, which typically store tuple count and total tuple size per partition. Similar to Section 2.1, this information can then be used to assign assign memory areas on slotted pages.

Radix Page Manager

This page manager applies the concept of radix-partitioning on slotted pages. It uses an three step process:

1. Histogram retrieval: Each thread reads its assigned materialized tuple chunk and builds up an histogram. As we are using fixed-size tuples, it only stores the tuple count per partition. This histogram is then forwared to the Radix Page Manager, which collects the histogram of each thread and then moves on to step 2.

2. Page allocation: With all histograms at hand, the page manager sums up each histogram into a global histogram. This global histogram stores how many tuples each partition has to store. With that information we can allocate required pages for each partition. When all pages are ready to be used, step 3 begins.

3. Assignment of slotted page sub-chunks: Each thread uses its local histogram to request storage locations for its tuples. The page manager uses the pre-allocated pages to assign each thread one or more memory chunks. These memory chunks can then be used exclusivly by the thread to store each tuples. Only the tuple count in the metadata has to be atomicly updated once, to signal that this thread has finished its work on this page. Otherwise, this write-out process does not rely on any synchronisation after the memory chunks.

After these three steps are done, the page manager is finished. Each page can be send to its receiver, once the expected tuple count is reached. This can be done by the last thread to increment the tuple count.

Ad-hoc Radix ("Hybrid") Page Manager

The idea of the previous Radix Page Manager can be used to construct an approach, where each thread can hand in its histogram and receive memory chunks on slotted pages independently from other threads. This page manager merges the three steps of the Radix Page Manager into one.

A thread hands in its histogram and request the memory chunks, where the tuples can be written. The page manager reads the histogram and processes each partition

individually. For each partition, where a tuple has to be stored, the lock of the partition is aquired. If there is any space left on a partition, it is used up and assigned to this thread. When the current page is full, a new page is allocated and the page manager continues the assignment process. These exclusive memory locations are then used by the thread to store its tuples.

In comparision to the Radix Page Manager, this page manager does not pre-allocate all necessary pages. Instead it allocates the pages when needed. Furthermore, a thread can already receive its memory chunks, while other threads are still constructing their histograms. This allows this implementation to avoid materialization of all tuples.

3.3.4 Thread-Local Pages and Merge-based Page Manager

A further step into reducing the necessaty of synchronsation are thread-local slotted pages. This slotted page management scheme can be split into two phases:

1. Thread-local write-out: These slotted pages are exclusivly used by the owning thread and after the tuple processing is done, each thread hands in their pages. When the page manager has received all pages of each thread, the merging phase is started.

2. Page merging: The page manager splits up all partitions onto the available threads. Each thread is then responsible for merging the slotted pages for each partition in the assigned partition range. To minimize tuple movement, the slotted pages a sorted decreasingly by the tuple count. Then the pages fewer tuples are merged into the fuller pages. It can be the cases that the last page cannot be fully merged into any other slotted page. Then this page has to be reordered so that the slots and data section start at the section beginnings. This is necessary to avoid having an gap at the beginnng of the slot or data section.

After the merging phase, all tuples are stored on the least possible amount of slotted pages. During the thread-local processing, it is likely that more slotted pages than necessary are created. This can lead to an significant higher memory consumption than the previous approaches.

3.3.5 Implementation-independent Optimizations

We use the following optimizations to speed up our simulation of the shuffle operator.

Padded atomics and locks

All implementations, that use an array of partition locks, are affected by false sharing. False sharing is caused when an cache line stores two independent values and one

Central Processing Unit (CPU) core is modifying the first value. Then another CPU core wants to access the second value, which causes an cache miss.

This performance degration can be avoided by storing each individual partition lock aligned to the L1-cache line boundary. This significantly reduces the amount of L1-cache misses, as the partition locks are frequently accessed.

Minimal page-locking

When holding a partition lock, the tuple write-out onto a slotted page is an expensive operation. To reduce the lock-duration, we gather the necessary write-out information and prepare the next tuple insertion, and releasing the lock before the actual tuple write-out.

Two-step buffered slotted page write out

When writing out a batch of tuples onto a slotted page, we can process each tuple individually. First, we construct the slot for the tuple, and then we store the variable-size data in the data section at the end of page. Linux on an x86-64 machine, typically uses 4 KiB memory pages. As we use 5 MiB slotted pages, it is expected that the slot and data are on different memory pages.

To reduce the amount simultanious accessed pages, we store our batches in a slot phase and a data phase. As we are now writing to either the slot or data section, we switch between memory pages less often. This is more friendly to the CPU cache and the Translation Lookaside Buffer (TLB).

3.4 On-Demand Partitioning

3.4.1 Overview

On-demand Partitioning is the simplest algorithm to implement the shuffle operator. As soon as this implementation receives a batch of tuples, it processes each tuple individually. First, the hash-function is applied onto the tuple to gather information in which partition this tuple belongs. With this information, we can write out the tuple into the corresponding partition.

This implementation is compatible with the lock-based or lock-free Page Manager and the Thread-Local Pages and Merge-based Page Manager (see Section 3.3)

3.4.2 Software Managed Buffers-based Partitioning

This naive approach can be significantly improved by using Software Managed Buffers (SMBs). Instead of immediately writing out the tuple, it can be stored in a thread-locally allocated buffer. The tuples, that belong do a partition, are stored in a dedicated area within this buffer. As soon as this sub-region capacity is full, we write out the tuples to a slotted page. Similarly, if all incoming tuples have been processed, we write out all remaining tuples inside the buffer.

This temporal storage brings another benefit. We can now use the batch insertion features from the page managers. Batch insertions are more CPU-cache and TLB friendly, as we switch our modified memory locations less often.

3.5 Histogram-based Partitioning

3.5.1 Overview

Similar to Radix Partitioning (see Section 2.1), a histogram can be used to assign threads memory location to store the tuples. A histogram in the context of Radix Partitioning stores information like count of tuples and their total size per partition. This information can then be used to assign each thread exactly the memory block it needs.

In our fixed-size tuple case, we only store the count of tuples of each partition. As we have to construct the whole histogram, before the thread can store any tuples, we have to iterate over all tuples twice. The first phase is necessary to create the histogram and receive memory locations to store the tuples. The second phase then uses these memory locations to store the tuples.

Similar to On-Demand Partitioning (see Section 3.4), the proposed implementations below benefit from Software Managed Buffers (SMBs) to write out tuples in batches.

3.5.2 Radix Partitioning

This implementation naively implements Radix Partitioning on a stream of incoming tuple batches. First, we fully materialize all generated tuples. Then each thread receives a part of the materialized tuples. Just like in the traditional Radix Partitioning, we construct the histogram for the assigned tuples.

The Radix Page Manager (see Section 3.3.3) collects the histograms of all threads. Each thread then receives their own, exclusive memory locations to store their tuples. This allows us to store the tuples without any synchronisation. Only the tuple count in the metadata section of each slotted page has to be updated atomically. As soon

as a slotted page reaches its proposed tuple count, it can be send to next downstream operator.

3.5.3 Ad-hoc Radix ("Hybrid") Partitioning

As the full materialization of all generated tuples and the wait times until the global histogram is created, are quite expensive, we propose a more efficent solution. Instead of materializing all tuples, we process all incoming tuples morsel-driven. For a given morsel, we then create an histogram. The Ad-hoc Radix Page Manager (see Section 3.3.3) uses this morsel-histogram to greedily assign this thread exclusive memory location to store the tuples.

Just like in the streaming Radix-Partitioning case above, the tuples are stored on each slotted page without needing synchronisation. The tuple count in the metadata section also requires synchronsiation in form of an atomic tuple count. As the morsel size are smaller than the assigned part of the materialized tuples, this counter has to be increased more often.

3.6 Collaborative Morsel Processing

3.6.1 Overview

In the previous implementations, each thread can write out to each partition. We avoid invalid writes by using locks or atomic operation. Shared usage of locks or atomics causes contention and frequent cache invalidations.

To reduce contention, we assigning each thread a partition range. A thread will only write out a tuple, if the partitioning function maps it to a partition within this partitioning range. Nevertheless, we still have to partition all tuples to their correct partition. Because of that, we have to process each tuple in a way that it can be stored any partition. Thus, a tuple must be processed by a group of threads, where the union of their individual partition ranges covers all possible partitions.

3.6.2 Collaborative Morsel Processing with exclusive partition ranges

This implementation constructs the partition ranges so that it fufills two conditions:

1. Any partition range does not overlap with any other partition range.
2. The union of all partition ranges results in a complete coverage of all possible partitions.

With these two conditions in place, we avoid having to syncronize the tuple write-out process. But we have to process each tuple p times, where p is the number of partition ranges. As the incoming tuples only have to be read, they typically can reside in the L2 or L3 CPU caches.

Assuming uniform distribution across all partitions, the implementations efficiency decreases with increasing CPU core count. If we distribute the partitions into fair partition ranges, the likelyhood of a given tuple to be partitioned in a given partition range is:

$$\Pr[X = \text{Partition Range}_i] \approx \frac{1}{p}, \quad \forall i \tag{3.3}$$

As we want to increase the amount of partition ranges (p) with increasing core count, the likelyhood that a thread processes a tuple for its assigned partition range decreases.

3.6.3 Collaborative Morsel Processing using processing units

As we can see in the above Collaborative Morsel Processing (CMP) with exclusive partition ranges implementation, exclusive partition ranges theoretical performance does not scale well. This implementation aims to reduce this impact by using processing units. A processing unit is a partition of t threads, that split up the total partitions into t fair, exclusive partition ranges. Again, within a processing unit, no partition ranges overlap.

When we use multiple processing units, we have to synchronise the tuple write-out again. In constrast to previous implementations without exclusive partition ranges, each partition can be written by only one thread per processing unit. This significantly reduces the contention on the synchronisation mechanism of each partition.

Furthermore, each tuples has to be only processed by a single processing unit. Assuming we fairly split up our c CPU cores into pu processing units, then each processing unit will have around $pt \approx \frac{c}{pu}$ threads to process tuples. When we increase to count of CPU cores and processing units in a similar manner, then the amount of threads per processing unit remains constant. This means our likelyhood of processing a tuple, that belongs to into the partition range of a randomly-selected thread is:

$$\Pr[X = \text{Partition Range}_i] \approx \frac{1}{pt}, \quad \forall i \tag{3.4}$$

When we compare the approximation in 3.3 and 3.4, it is visible that as long as we keep the ratio between CPU cores and processing units constant, the processing efficieny remains constant. In contrast, when using Collaborative Morsel Processing with exclusive partition ranges the amount of exclusive partition ranges increases with the CPU core count. This then leads to a decrease of processing efficiency when increasing the CPU cores.

3.7 Thread-Local Pages and Merge-based Partitioning

3.8 Complexity Analysis

3.8.1 Time Complexity

To shuffle the tuples into partitions, we have to process every tuple. Thus, $\Omega(n)$ is the lowerbound for all shuffle implementations. As we process each tuple at most $c = \max(2, t)$, with t denoting the thread count, times and only apply $O(1)$ operations on each tuple, we can upperbound the implementations time complexity using this constant $O(c \cdot n) = O(n)$. Combining both lower- and upperbound, our proposed implementations run in $\Theta(n) = \Omega(n) \cap O(n)$.

Tuple Access Frequency

As the theoretical Time Complexity analysis above hides the constant $c = \max(2, t)$, we list the tuple access frequency per implementation:

- On-Demand Partitioning: Every tuple is only processed once and directly written to its final destination. When we do not use SMBs, we write each tuple directly onto the slotted page and only access it once. In comparision, when using SMBs we temporally store the tuple in the buffer, which increases the access count to two.

- Histogram-based Partitioning: To create a Histogram, we have to access the the tuple once. Furthermore, if the implementation requires full materialization, we have to materialize all tuples before processing it. This means that the tuple access frequency is either two or three, depending on if materialization is necessary.

- Collaborative Morsel Processing: In the simpler implementation with exlusive partition ranges, we have to process each tuple atleast p times, with p denoting the count of exlusive partition ranges. As we use SMBs, we access each tuple twice when processing it. Thus, the tuple access frequency is $2p$.

  When using processing units, we only have to process at most $\max(pt_i),\ \forall i$ times. pt_i is the total number of threads (and count of exlusive partition ranges) of processing unit i. Similarly, it follows that the tuple access frequency upper bounded by $2 * \max(pt_i),\ \forall i$ as we use SMBs as well.

- Thread-Local Pages and Merge-based Partitioning: In this implementation, we use SMBs as well. Similar to the On-Demand Partitioning, we have to at least process the tuples two times. But as we are using thread-local slotted pages, we

may have to merge a slotted page into another page. A tuple can only be merged into another page or moved to a different location on the page, onces. Thus, our tuple access frequency is three.

3.8.2 Space Complexity

Similary as in the Time Complexity analysis, we have to store each tuple at least once. Thus, we can set a lowerbound of $\Omega(n)$. Only, in the full materialization of Radix Partitioning, we create a copy of each tuple to store it on the page. Otherwise, we only move the tuples until it is stored in its final location on a slotted page. As the Software Managed Buffers are constant, we can upper bound it by a constant c_{SMB}. With this information, we can define the upperbound for the Space Complexity to $O(2 * n + c_{SMB}) = O(n)$. It now follows that the Space Complexity of each implementation is $\Theta(n) = \Omega(n) \cap O(n)$.

Memory Consumption

4 Evaluation

4.1 Experimental Setup

Hardware

Software

4.2 Tuple Generation

4.3 Tuple Write Benchmark

4.4 Shuffle Benchmark

4.4.1 Memory Consumption

4.4.2 Performance

4.5 Comparison with Stream Processing Systems

5 Conclusion

5.1 Conclusion

5.2 Future Work

Abbreviations

CPU Central Processing Unit

TLB Translation Lookaside Buffer

SMB Software Managed Buffer

CMP Collaborative Morsel Processing

List of Algorithms

List of Figures

List of Tables

Bibliography

[1] F. Gürcan and M. Berigel. "Real-Time Processing of Big Data Streams: Lifecycle, Tools, Tasks, and Challenges." In: *2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*. Oct. 2018, pp. 1–6. DOI: 10.1109/ISMSIT.2018.8567061.

[2] E. Zamanian, C. Binnig, and A. Salama. "Locality-aware Partitioning in Parallel Database Systems." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 17–30. ISBN: 9781450327589. DOI: 10.1145/2723372.2723718.

[3] S. Chu, M. Balazinska, and D. Suciu. "From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 63–78. ISBN: 9781450327589. DOI: 10.1145/2723372.2750545.

[4] G. Andrade, D. Griebler, R. Santos, and L. G. Fernandes. "A parallel programming assessment for stream processing applications on multi-core systems." In: *Computer Standards & Interfaces* 84 (2023), p. 103691. ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2022.103691.

[5] F. Liu, L. Yin, and S. Blanas. "Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems." In: *ACM Trans. Database Syst.* 44.4 (Dec. 2019). ISSN: 0362-5915. DOI: 10.1145/3360900.

[6] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. "Locality-sensitive operators for parallel main-memory database clusters." In: *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. Ed. by I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski. IEEE Computer Society, 2014, pp. 592–603. DOI: 10.1109/ICDE.2014.6816684.

[7] A. Ailamaki, D. J. DeWitt, and M. D. Hill. "Data page layouts for relational databases on deep memory hierarchies." In: *The VLDB Journal* 11.3 (Nov. 2002), pp. 198–215. ISSN: 1066-8888. DOI: 10.1007/s00778-002-0074-9.