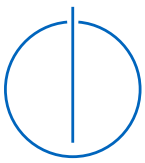
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

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

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Streaming processing engines . . . 1
1.3 Shuffle operator . . . 1
1.4 Slotted pages . . . 2
1.5 Problem setting . . . 2

2 Related work **4**
2.1 Radix Partitioning . . . 4
2.2 Partitioned Joins . . . 4
2.3 Software Managed Buffers . . . 4

3 Implementations **5**
3.1 Partitioning . . . 5
3.2 Slotted Pages . . . 5
3.3 Slotted Page Managers . . . 6
3.3.1 Lock-based Page Manager . . . 6
3.3.2 Lock-free Page Manager . . . 7
3.3.3 Histogram-based Page Managers . . . 9
3.3.4 Thread-Local Pages and Merge-based Page Manager . . . 10
3.3.5 Implementation-independent Optimizations . . . 10
3.4 On-Demand Partitioning . . . 11
3.4.1 Overview . . . 11
3.4.2 Software Managed Buffers-based Partitioning . . . 11
3.5 Histogram-based Partitioning . . . 12
3.5.1 Overview . . . 12
3.5.2 Radix Partitioning . . . 12
3.5.3 Ad-hoc Radix ("Hybrid") Partitioning . . . 13

3.6 Collaborative Morsel Processing . . . 13
3.6.1 Overview . . . 13
3.6.2 Collaborative Morsel Processing with exclusive partition ranges 13
3.6.3 Collaborative Morsel Processing using processing units . . . 14
3.7 Thread-Local Pages and Merge-based Partitioning . . . 15
3.8 Complexity Analysis . . . 15
3.8.1 Time Complexity . . . 15
3.8.2 Space Complexity . . . 16

4 Evaluation **17**
4.1 Benchmarking Setup . . . 17
4.2 Benchmarks . . . 17
4.2.1 Benchmark Overview . . . 17
4.2.2 Shuffle Operator Benchmark . . . 18
4.3 Comparison with Stream Processing Systems . . . 22

5 Conclusion **24**
5.1 Conclusion . . . 24
5.2 Future Work . . . 24

Abbreviations **25**

List of Algorithms **26**

List of Figures **27**

List of Tables **28**

Bibliography **29**

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

3.1 Partitioning

We generate our tuples using a 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, we use the following partitioning function when the number of partitions is a power of two:

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

Thus, we only need a substraction and a logical-and instruction to calculate in which partition a given tuple is placed. As the count of partitions remains constant during the shuffle operator execution, we can extract the constant partitions - 1. Now, we only need a single logical-and instruction to calculate the partition.

3.2 Slotted Pages

Slotted pages store their information on fixed-size memory blocks that are split up into 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.

In our implementation, we only store the tuple count in the header. We split each tuple to construct our shuffle simulation close to real-world usage. Each slot contains the 4-byte key together with data offset and length information. We store the remainder of the tuple in the data section at the end of each page.

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

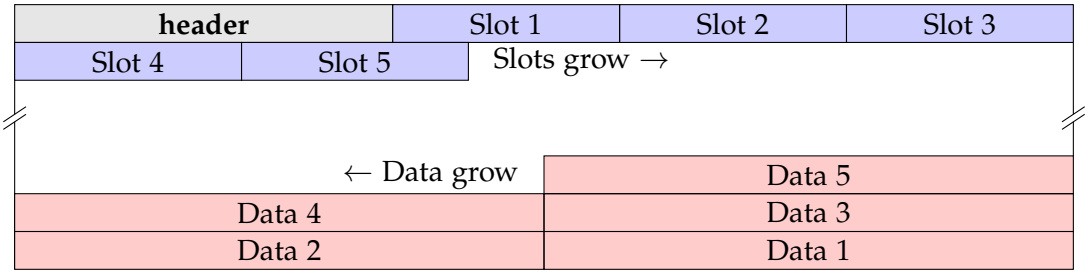


Figure 3.1: Slotted Page grow visualization

3.3 Slotted Page Managers

As some implementations 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.

We initialize each partition with an empty slotted page to simplify the implementations. Initializing each partition with a slotted page significantly reduces the complexity of the page manager implementations.

3.3.1 Lock-based Page Manager

We use a single lock and a vector for each partition to store the slotted pages. As can be

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
```

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. Compared to the lock-based variant, we must store our slotted pages in a pointer-stable data structure. A pointer-stable data structure is necessary to ensure threads can work simultaneously while a thread adds a new slotted page. Furthermore, we must 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 a compare-and-exchange operation. This previously stored index then acts as our location, where we store the tuple on the page. In Algorithm 3, we also add a condition to stop attempting to further increase the count of tuples on the

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 none
6 |   end
7 end
8 return {page_ptr, page_size, current_tuple_count}
```

page if it is full. This condition ensures that threads move to the newly allocated page. Given the index, where the tuple is placed, we also append a pointer to the start of the page and the page size. This information ensures we can write the tuple on 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 do
4 |   wi = current_page[partition].load()->increment_and_fetch_opt_write_info()
5 end
6 if wi.tuple_index == LockFreeSlottedPage::get_max_tuples(wi.page_size) - 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 receive an index on a slotted page. If we write the last tuple on the page, we add a new page to this partition. This last-tuple-check is necessary to create a unique condition when allocating a new page. With that information, we can write the tuple onto the slotted page.

3.3.3 Histogram-based Page Managers

The following page managers use histograms to keep track of the count of tuples per partition within a given set of tuples. Similar to Radix Partitioning (see Section 2.1), this information can be used to assign memory areas on slotted pages.

Radix Page Manager

This page manager applies the concept of Radix Partitioning on slotted pages. It uses a three-step process:

1. Histogram retrieval: Each thread reads its assigned materialized tuple chunk and builds up a histogram. As we are using fixed-size tuples, it only stores the tuple count per partition. Then, we forward this histogram 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 the 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. The thread can then use these memory chunks exclusively to store each tuple. 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 synchronization after the memory chunk distribution.

After these three steps, each page can be sent to its receiver once the expected tuple count is reached. The last thread to atomically-increment the tuple count in the metadata section can do this.

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 requests the memory chunks, where the tuples can be written. The page manager reads the histogram and processes each partition

individually. We acquire the partition lock for each partition where at least a single tuple has to be stored. If there is any space left on a slotted page, it is used up and assigned to this thread. When the current page is full, we allocate a new page, and the page manager continues the assignment process. Each thread uses these exclusive memory locations to store its tuples.

Compared 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 others still construct their histograms. This change allows this implementation to avoid the materialization of all tuples.

3.3.4 Thread-Local Pages and Merge-based Page Manager

A further step into reducing the necessity of synchronization is thread-local slotted pages. We split this slotted page management scheme into two phases:

1. Thread-local write-out: These slotted pages are exclusively used by the owning thread, and after the thread finishes its tuple processing, each thread hands in its pages. When the page manager receives all pages of each thread, the merging phase starts.

2. Page merging: The page manager splits 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 are sorted decreasingly by the tuple count. Then, we merge the pages with fewer tuples into the fuller pages. If 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 beginning of the section. This reordering is necessary to avoid having a gap at the beginning of the slot or data section.

After the merging phase, this implementation stores all tuples on the least possible number of slotted pages. During the thread-local processing, this implementation will likely create more slotted pages than necessary. This unnecessary allocation of pages can lead to a significantly 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 appears when a cache line stores two independent values and one Central

Processing Unit (CPU) core modifies the first value. Then, another CPU core wants to access the second value, which causes a cache miss.

We avoid this performance degradation by storing each partition lock aligned to the L1-cache line boundary. This alignment significantly reduces the number of L1 cache misses, as partition locks are frequently accessed.

Minimal page-locking

When holding a partition lock, the tuple write-out onto a slotted page is expensive. We gather the necessary write-out information to reduce the lock duration, prepare the subsequent tuple insertion, and release the lock before the actual tuple write-out.

Two-step buffered slotted page write out

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

We store our batches in slot and data phases to reduce the amount of simultaneously accessed pages. As we now write to the slot or data section, we switch between memory pages less often. This two-phase approach 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 most straightforward algorithm for implementing the shuffle operator. As soon as this implementation receives a batch of tuples, it processes each tuple individually. First, the hash function is applied to the tuple to gather information on the partition to which this tuple belongs. With this information, we can write 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, we store it in a thread-locally

allocated buffer. Each partition owns a dedicated area within this buffer. We write the tuples to a slotted page when this sub-region capacity is full. 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

Like Radix Partitioning (see Section 2.1), we use a histogram to assign threads memory location to store the tuples. A histogram in the context of Radix Partitioning stores information like the count of tuples and their total size per partition. We use this information to assign each thread the memory blocks 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, we have to iterate over all tuples twice before the thread can store any tuples. 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. Like in 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 its exclusive memory locations to store its tuples. We can store the tuples without synchronization as we assign exclusive memory locations. Only the tuple count in the metadata section of each slotted page has to be updated atomically. A slotted page can be sent to the next downstream operator when it reaches its proposed tuple count.

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 expensive, we propose a more efficient solution. Instead of materializing all tuples, we process all incoming tuples morsel-driven. For a given morsel, we then create a 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.

Like in the streaming Radix Partitioning case above, we store the tuples on slotted pages without synchronization. The tuple count in the metadata section also requires synchronization in the form of an atomic tuple count. As the morsel size is smaller than the materialized tuples' assigned part, we must increase the tuple counter more often than using streaming Radix Partitioning.

3.6 Collaborative Morsel Processing

3.6.1 Overview

In the previous implementations, each thread could write to each partition, and we avoided invalid writes by using locks or atomic operations. Shared usage of locks or atomics causes contention and frequent cache invalidations.

To reduce contention, we assign 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 so that it can end up in any partition. Thus, we must process each tuple by a group of threads, where the union of their 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 fulfills 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 synchronize the tuple write-out process. However, we must 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 implementation efficiency decreases with increasing CPU core count. If we distribute the partitions (p) into fair partition ranges, the likelihood of a given tuple being partitioned in a given partition range is:

$$\Pr[X = \text{partition range}_i] = \frac{|\text{partition range}_i|}{p}, \quad \forall i \tag{3.3}$$

As we want to increase the amount of partition ranges with increasing core count, the likelihood that a thread processes a tuple for its assigned partition range decreases. This is the case, as with increasing partition ranges the sizes of each 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 splits 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 synchronize the tuple write-out again. In contrast 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 synchronization mechanism of each partition.

Furthermore, we must process each tuple only 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 the count of CPU cores and processing units similarly, the amount of threads per processing unit remains constant. Thus, our likelihood of a randomly selected thread processing a tuple that belongs to its partition range is:

$$\Pr[X = \text{partition range}_i] \approx \frac{|\text{partition range}_i|}{pt}, \quad \forall i \tag{3.4}$$

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 efficiency per thread 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 increase in exclusive partition ranges decreases the processing efficiency when increasing the CPU cores.

3.7 Thread-Local Pages and Merge-based Partitioning

This implementation is very similar to the previous On-Demand implementation with Software Managed Buffers (see Section 3.4). Instead of using the On-Demand Page Manager, it heavily builds on the Thread-Local Pages and Merge-based Page Manager (see Section 3.3.4). Thus, this solution avoids sharing slotted pages between threads, which can only be achieved with significantly higher memory usage.

Each thread partitions the incoming tuples into thread-local slotted pages. With increasing tuple count and more partitions, these thread-local slotted pages are less likely to be fully used, leading to significant memory overhead. We hand these slotted pages to the shared Page Manager at the end of the tuple processing.

When the Page Manager receives all slotted pages, the slotted pages of each partition are sorted decreasingly by the number of tuples they store. Then, each thread is assigned an exclusive partition range and slotted pages for this partition range. All threads then merge the emptier pages into the fuller pages. All empty pages, which caused the additional memory overhead of this implementation, are then deallocated. Then, each thread returns the merged slotted pages to the Page Manager, which is the last part of the partitioning process.

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 lower bound 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 upper bound the implementations time complexity using this constant $O(c \cdot n) = O(n)$. Combining both lower- and upper bounds, 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 not using SMBs, we write each tuple directly onto the slotted page and only access it once. In comparison, 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 tuple once. Furthermore, if the implementation requires full materialization, we materialize all tuples before processing them. This means that the tuple access frequency is two or three, depending on whether materialization is necessary.
- Collaborative Morsel Processing: In the more straightforward implementation with exclusive partition ranges, we have to process each tuple at least pr times, with pr denoting the count of exclusive partition ranges. As we use SMBs, we access each tuple twice when processing it. Thus, the tuple access frequency is $2pr$.

  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 exclusive partition ranges) of processing unit i. Similarly, it follows that the tuple access frequency is 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 also use SMBs. Similar to the On-Demand Partitioning, we must at least store the tuples two times. However, 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 once. Thus, our tuple access frequency is three.

3.8.2 Space Complexity

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

4 Evaluation

In this chapter we explain our benchmarks and evaluate the implementations of the shuffle operator. Furthermore, we discuss best-case upper bounds to put our solutions performance in perspective.

4.1 Benchmarking Setup

The following benchmarks were executed on a 125 GiB (DDR4-2666) x86-64 machine with an Intel(R) Core(TM) i9-7900X CPU. The code was compiled using gcc version 13.3.0 under Ubuntu 24.04.1 LTS running the 6.5 linux kernel. We used the following gcc flags to compile the code: `-O2 -march=native -flto`

4.2 Benchmarks

We simulate the real world usage of the shuffle operator using a parallel tuple generation and shuffle, followed by a final tuple count check.

4.2.1 Benchmark Overview

Tuple Generation

This phase uses a Mersenne Twister pseudo-random 64-bit number generator, which is initialized using a true-random seed. This generator creates batches of tuples, using the total size of the requested tuple batch and without generating each tuple individually.

We allocate a new memory block for each tuple batch, which the implementions then process and store onto slotted pages. In the real world, the incoming tuples are often stored on slotted pages. When this is the case, each tuple batch is in a new memory location as well.

Each implementation uses the exact same tuple generation process, just the size of the generated batches varies.

Tuple Shuffle

In this step, we simulate the real world usage of the shuffle operator. The operator receives an input stream of tuple batches and creates parttioned output stream in form of slotted pages.

To keep the comparision between the implementations fair, each implementation must create an partitioned output stream with the following properties: For each partition, all slotted pages must be full except for the last one and each slotted page must store its tuples as single block starting from the the first slot. These two conditions assure that each implementation produces the same result, just the ordering of the tuples across the slotted pages can differ.

Final tuple count check

After the implementation has processed all tuples, we scan over all slotted pages and sum up the tuple count. This way we assure that all tuples are written in their final location.

4.2.2 Shuffle Operator Benchmark

The following benchmarks were executed on the above system. During the execution, the linux perf counters were recorded and these mesurements are the fundament of the following evaluation.

We use three different tuples sizes (4B, 16B, 100B) to simulate the shuffle operator usage. As we use first 4 Byte of a given tuples as our partitioning key, we also store those 4 Bytes in the slot. The remaining bytes are stored as variable-sized data and thus in the data section of the slotted page. In the 4 Byte case, this storage approach does only use the slot section, as there are no bytes besides the key to be stored. In contrast, both the 16B and 100B tuples have to be split into a 4 Byte key part stored in the slot, and the remaining 12B and 96B are stored in the data section.

Furthermore, we recorded two theoretical baselines for the shuffle implementations, which help to put the results into perspective. The first baseline simulates tuple writes onto unsyncronized slotted pages. In comparision, the second baselines uses shared and thus syncronized slotted pages.

To achieve an best-case upper bound performance, both baseline implementations use Software Managed Buffers (SMBs). The baselines always fully fill the SMB and then store it on the n partition. After that batched write out, the SMB is reset and the next tuples will be stored in the $n + 1$ partition. After a write to the last partition, we start from the first partition again. As our tuple generator creates uniformly-distributed

keys accross all partitions, these baselines act as best-case upper bounds, which can rarely be reached.

Performance with few Partitions

The following figures illustrate the processed tuples per second of each implementation as the thread count increases. As the benchmarking machine has only ten hardware threads, it is expected that beyond these ten hardware threads, only minimal performance improvements can be achieved. The theoretical baselines perform very similarly with a few threads, but as the thread count increases, the synchronization causes contention and thus the performance gain per thread decreases.

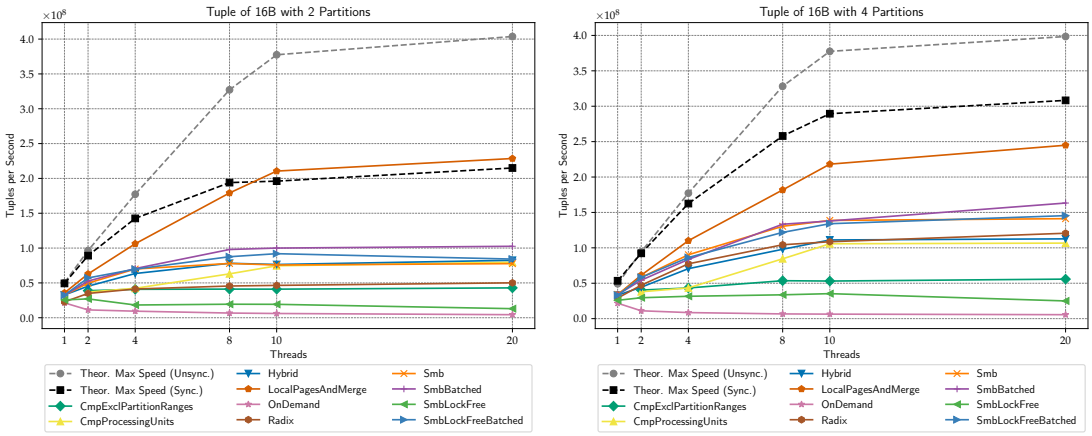


Figure 4.1: Benchmark Plots for Tuple of 16B with 2 and 4 Partitions

In figure 4.1, we can see that the LocalPagesAndMerge-implementation processes by far the most tuples per second. It even exceeds the synchronized best-case base line, as the LocalPagesAndMerge-approach avoids having to synchronize during write out at the cost of a small sort-and-merge phase at the end. In the four partition plot, we can already see that the performance of the LocalPagesAndMerge-implementation decreases in comparision to the other implementations. This is the case, as the thread-local pages causes a lot of heap-allocations and with a increasing number of partitions, the sort-and-merge phase cost increases as well.

After the LocalPagesAndMerge-implementation, there is a group of three Software Managed Buffer (SMB)-based implementations: Smb, SmbBatched and SmbLockFree-Batched. These three implementations struggle to scale with increasing thread count in the two partition case, as having to write all the tuples on two shared slotted pages causes a lot of contention. But it can already be seen that with a increasing number of

partition, these three implementation are able to scale with increasing thread count.

The next group contains the following three implementations: Collaborative Morsel Processing (CMP) with Processing Units, Hybrid and Radix. The leader of this group is the Hybrid implementation, that especially in the two partition case scales better than the other two implementations. The Hybrid implementation can process more tuples there, as it does not need to fully materialize all tuples. Furthermore, the slotted pages are allocated when needed and each thread can request memory locations without having to wait for other threads to hand in their histogram. The CMP with Processing Units implementation performs better with increasing threads, as each processing unit contains one thread, that only generates new tuples without partitioning them. All the other threads of this processing unit then can process these generated tuples.

The remaining CMP, OnDemand and SmbLockFree implementation do not scale with increasing threads. As explained in the Section 3.6.2, the CMP with exlusive partition ranges effiency decreases per thread with an increase in the number of threads. The OnDemand approach does not use a SMB, and has to immidiatly write each tuple to its final slotted page. As all slotted pages are shared, this causes a lot of contention. Similarly, the non-batched lock-free SMB implementation uses two busy-waiting loops, that aim to update the tuple-count of a slotted page using an compare-and-exchange operation. Thus, this approach has a significant higher number of branch-misses (1-3x), instructions (1-4x) and L1-cache-misses (1-2x) in comparision with the other SMB implementations.

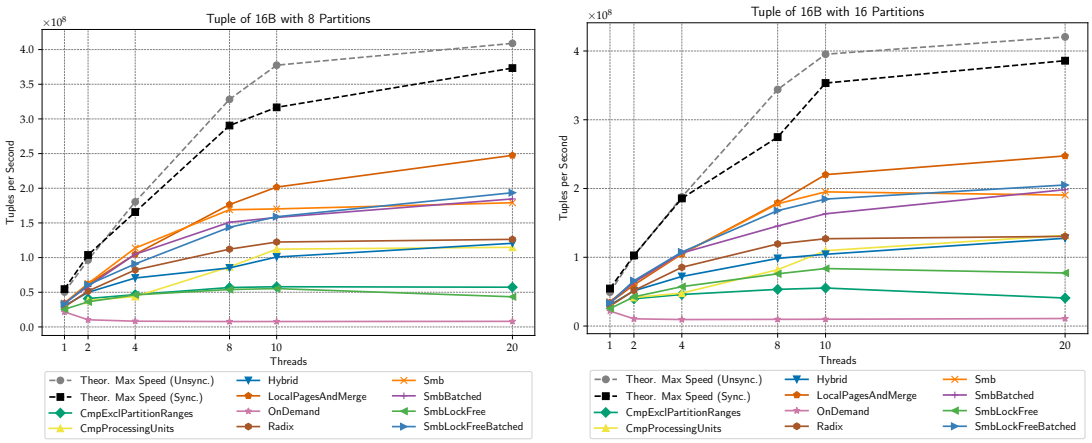


Figure 4.2: Benchmark Plots for Tuple of 16B with 8 and 16 Partitions

In Figure 4.2, that uses 8 and 16 partitions, we can see that the performance gap between the LocalPagesAndMerge- and the other implementations shrinks with increasing partitions. Also, the previous groups of implementations with similar performance

characteriscs remain.

The LocalPagesAndMerge implementation does not significantly increase its throughput, as with a higher number of partitions, there also must be more thread-local slotted pages per thread. This causes more heap allocations and the sort-and-merge phase is getting more expensive. Furthermore, the Last Level Cache (LLC)-cache-misses and Translation Lookaside Buffer (TLB)-misses increase as more memory location are accessed.

The previous group of SMB implementations (Smb, SmbBatched and SmbBatchedLockFree) can now process very similar amounts of tuples per second as the LocalPagesAndMerge approach. The synchronization contention decreases with increasing partitions, as the load on locks or atomics now is spread up between more slotted pages, and thus across more locks or atomics.

The next group of implementation contains the following four implementations: CmpProcessingUnits, Hybrid, Radix and SmbLockFree. The leading implementation is the Radix approach, which even performs better than the Hybrid implementation. In comparision, the Radix implementations has higher L1/LLC-cache-misses, but has less branch- and TLB-misses. This leads to a overall better CPU-usage with less time idling. Overall all implementations of this group have significant higher L1/LLC-cache misses than the previous group.

The last group of implementation, which do not scale well, contain yet again: CmpExclusivePartitionRanges, OnDemand and SmbLockFree. Similar to the previous Figure 4.1, increasing the number of partitions does not solve the decreasing efficiency per added thread issue of the CmpExclusivePartitionRanges implementation. Also, the OnDemand approach struggles even more to write out the tuples, as with increasing partitions, the amout of slotted pages also increases. This causes even more L1/LLC-cache-misses, as the Central Processing Unit (CPU) cannot hold all these memory location in caches. Like in the previous benchmark, the SmbLockFree approach still uses the two busy waiting loops to atomically get the write out information for a given tuple batch. With increasing number of partitions, these atomic tuple-count values less frequently requested by multiple threads. Thus, with increasing partitions, this implementation starts to improve its scaling.

Performance with many Partitions

c

d

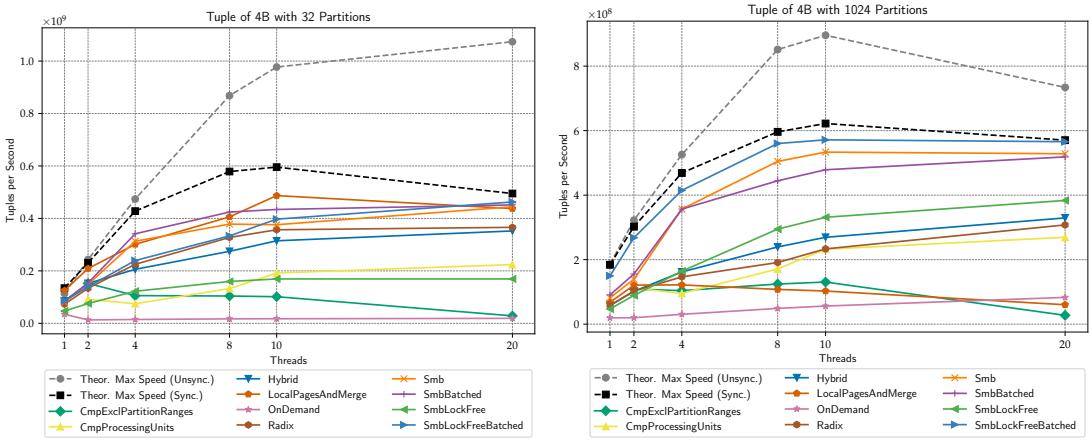


Figure 4.3: Benchmark Plots for Tuple of 4B with 32 and 1024 Partitions

Memory Consumption

4.3 Comparison with Stream Processing Systems

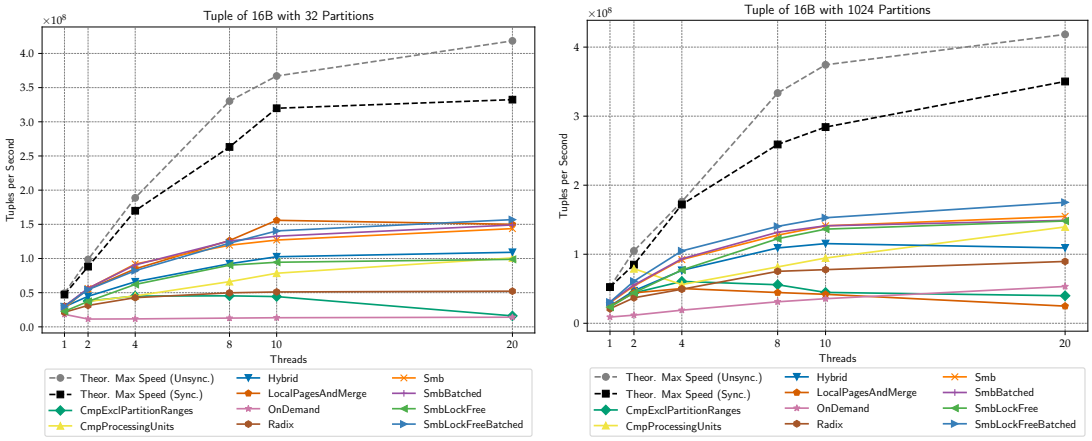


Figure 4.4: Benchmark Plots for Tuple of 16B with 32 and 1024 Partitions

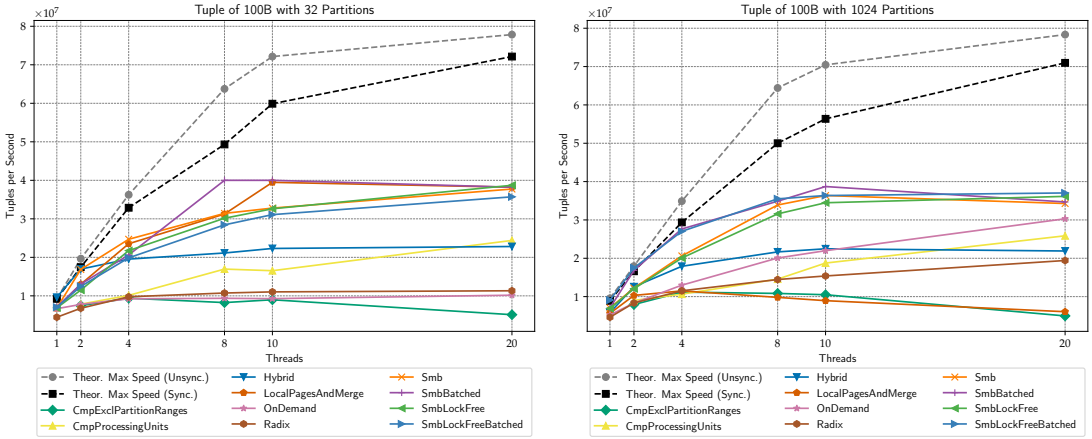


Figure 4.5: Benchmark Plots for Tuple of 100B with 32 and 1024 Partitions

5 Conclusion

5.1 Conclusion

5.2 Future Work

Abbreviations

CPU Central Processing Unit

TLB Translation Lookaside Buffer

LLC Last Level Cache

SMB Software Managed Buffer

CMP Collaborative Morsel Processing

List of Algorithms

1 Lock-based Page Manager insert_tuple Algorithm 6
2 Lock-based Page Manager insert_tuple_batch Algorithm 7
3 Lock-free Slotted Page increment_and_fetch_opt_write_info Algorithm . 8
4 Lock-free Page Manager insert_tuple Algorithm 8

List of Figures

3.1 Slotted Page grow visualization 6

4.1 Shuffle Benchmark Plots for Tuple of 16B with 2 and 4 Partitions 19
4.2 Shuffle Benchmark Plots for Tuple of 16B with 8 and 16 Partitions . . . 20
4.3 Shuffle Benchmark Plots for Tuple of 4B with 32 and 1024 Partitions . . 22
4.4 Shuffle Benchmark Plots for Tuple of 16B with 32 and 1024 Partitions . 23
4.5 Shuffle Benchmark Plots for Tuple of 100B with 32 and 1024 Partitions . 23

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.