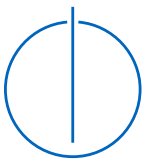
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementing an Efficient Shuffle Operator for Streaming Database Systems

Jonas Ladner



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

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

I would like to express my deepest gratitude to my supervisor, Maximilian Rieger, for his invaluable feedback and insightful ideas. I am also very thankful to Prof. Dr. Thomas Neumann and the database chair for their excellent lectures, which sparked my interest in exploring database internals.

I am truly grateful to my family for their constant support throughout my educational journey.

Finally, I would like to thank my friends who have been by my side during my time at TUM.

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

This chapter presents relevant algorithms and approaches, positioning them within the context of our problem domain.

2.1 Partitioned Joins

Partitioned joins are a widely used class of hash-based join algorithms [8–10], frequently employed in modern database systems [11–13].

A basic partitioned join algorithm consists of a partitioning and a join execution phase. In the first phase, the algorithm partitions the build and probe sides using a hash key derived from the join condition. The subsequent join execution phase uses the property that only tuples within the same partition must be compared, improving locality and reducing unnecessary checks. The final output consists of the union of each partition's results.

As the underlying hardware significantly impacts the performance of partitioned join algorithms, recent publications have proposed hardware-conscious implementations [9, 14–16]. In the following chapter (see Chapter 3), we incorporate hardware-conscious approaches from these papers and tailor them to the requirements of the streaming shuffle operator.

2.2 Radix Partitioning

Radix partitioning is a fundamental technique to efficiently partition a large dataset [14, 16–18].

Its naive, fixed-size tuple version uses three phases: histogram construction, offset calculation, and a write-out phase [14]. The first pass constructs a frequency map, the so-called histogram, that stores the number of tuples for each partition. Using this histogram, the algorithm computes a prefix-sum array to determine partition offsets. In the last phase, it allocates a memory block large enough to store all tuples. The histogram and the offset prefix-sum provide enough information to write each tuple to its final destination in the newly allocated block.

By grouping writes into a contiguous memory region, radix partitioning improves cache locality and reduces write amplification. Our approach extends this concept by applying histogram-based partitioning dynamically to streaming tuple workloads (see Section 3.5).

2.3 Software Managed Buffers

Database systems frequently use Software Managed Buffers in the size of cache lines to improve throughput and avoid Translation Lookaside Buffer (TLB) thrashing [19–21]. SMBs reduce TLB misses by buffering small batches of tuples before writing them out, reducing the number of memory page accesses. Prior work has shown that using cache-line-sized buffers significantly reduces TLB pressure and improves throughput [22].

In our shuffle operator, all implementations, except the naive OnDemand approach, leverage SMBs to optimize memory access and improve overall throughput.

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.

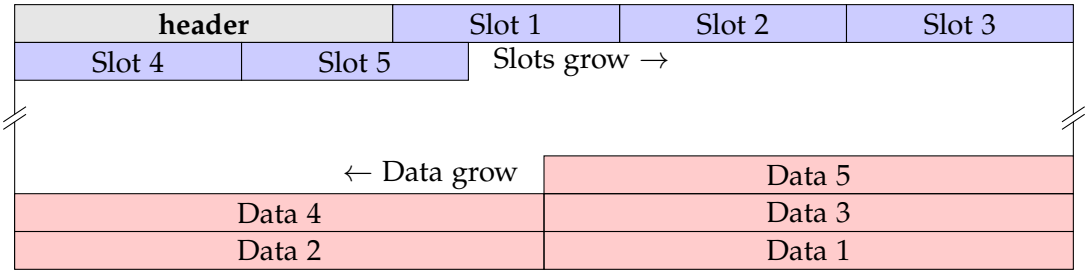


Figure 3.1: Slotted Page grow visualization

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 2.2), 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 2.2), 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 $O(c * n) = O(n)$. 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 $\Omega(n)$.

As the Software Managed Buffers (SMBs) sizes are constant, we can upper bound it by a constant:

$$c_{\text{SMB}} = (\text{Size of a single SMB}) * (\text{Thread count}) \tag{3.5}$$

Similarly, the LocalPagesAndMerge uses thread-local slotted pages. The sort-and-merge phase starts when we finish processing the incoming stream of tuples. All slotted pages of a partition that are not full have to be merged. During the merge process, some slotted pages can be merged into other slotted pages and deallocated. These deallocated slotted pages cause an unnecessary memory consumption that can be upper bounded by a constant:

$$c_{\text{LPAM}} \leq (\text{Partition count}) * (\text{Thread count}) * (\text{Size of a slotted page}) \tag{3.6}$$

This constant c_{LPAM} is independent of the input tuple size, but it can be several GiB big in real-world cases. When a distributed database system is used, the total size of

incoming tuples is significantly higher than the partition and thread count. Thus, we continue our space complexity analysis with the assumption: $n \gg c_{\text{LPAM}}$

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 we store them in their final location on a slotted page. With this information, we can define the upper bound for the Space Complexity to $O(2 * n + c_{SMB} + c_{\text{LPAM}}) = O(n)$. It follows that the Space Complexity of each implementation is within $\Theta(n) = \Omega(n) \cap O(n)$.

4 Evaluation

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

4.1 Benchmarking Setup

We executed the following benchmarks on a 125 GiB (DDR4-2666) x86-64 machine with an Intel(R) Core(TM) i9-7900X CPU. The machine runs Ubuntu 24.04.1 LTS with the Linux 6.5 kernel. The C++23 code was compiled using GCC 13.3.0 with the optimization flags `-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 initializes a Mersenne Twister pseudo-random 64-bit number generator with a true-random seed. The generator creates batches of tuples based on the total requested batch size without generating each tuple individually.

For each tuple batch, we allocate a new memory block, which the implementations then process and store onto slotted pages. Similarly, real-world systems send tuples via slotted pages, which reside in different memory locations in the receiver system.

Each implementation follows the same tuple generation process, varying only in the size of the generated batches.

Tuple Shuffle

n this step, we simulate the real-world usage of the shuffle operator. The operator processes an input stream of tuple batches and produces a partitioned output stream in the form of slotted pages.

To ensure a fair comparison between implementations, each implementation must generate a partitioned output stream with the following properties: For each partition, all slotted pages must be fully utilized except for the last one, and each slotted page must store its tuples as a contiguous block starting from the first slot. These conditions ensure that all implementations produce the same result, with potential differences only in ordering tuples across slotted pages.

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 ensure that all tuples are written in their final location.

4.2.2 Shuffle Operator Benchmark

We executed the following benchmarks on the system described in Section 4.1. During execution, we recorded Linux perf counters, and these measurements form the foundation of the following evaluation.

We use three tuple sizes (4B, 16B, 100B) to simulate shuffle operator usage. Since the first 4 bytes of each tuple serve as the partitioning key, we store these 4 bytes in the slot. The remaining bytes are stored as variable-sized data in the data section of the slotted page. In the 4-byte case, this storage approach uses only the slot section, as no additional data needs to be stored. In contrast, the 16B and 100B tuples split into a 4-byte key part stored in the slot and the remaining 12B and 96B, which reside in the data section.

Furthermore, we recorded two best-case references for the shuffle implementations to put the results into perspective. The first reference simulates tuple writes onto unsynchronized slotted pages, while the second uses shared and thus synchronized slotted pages. Both implementations leverage Software Managed Buffers (SMBs) to achieve a best-case upper bound performance. The best-case references always fully fill the SMB before storing it in the n partition. After this batched write-out, the SMB resets, and the next tuples are stored in the $n + 1$ partition. After writing to the last partition, the process restarts from the first partition. Since our tuple generator creates uniformly distributed keys across all partitions, these references represent best-case upper bounds that are rarely achievable.

In the following benchmarks, we use a constant SMB size per implementation, independent of the thread count. For example, if an implementation uses a 2 MiB SMB and 8 threads, each thread operates with 250;KiB of SMB:

$$\frac{\text{SMB Size}}{\text{Thread count}} = \frac{2;\text{MiB}}{8} = 250;\text{KiB} \tag{4.1}$$

Since SMB size varies between different thread configurations within the same implementation, small performance jumps occasionally appear in the benchmark plots. These jumps occur when the SMB size aligns with L1/L2 cache lines.

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 physical cores, it is expected that only minimal performance improvements can be achieved beyond these ten physical cores. The two best-case references 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.

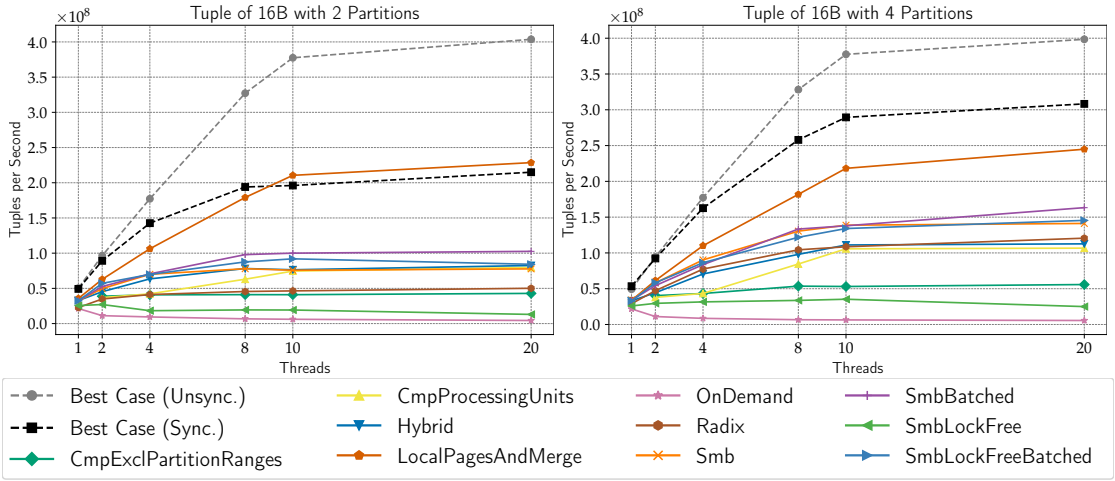


Figure 4.1: Benchmark Plots for Tuple of 16B with 2 and 4 Partitions

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 reference, as the LocalPagesAndMerge-approach avoids synchronizing during write-out at the cost of a small sort-and-merge phase at the end. In the four partition plots, we can already see that the performance of the LocalPagesAndMerge implementation decreases compared

to the other implementations. This is the case, as the thread-local pages cause a lot of heap allocations, and with an increasing number of partitions, the sort-and-merge phase cost also increases.

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. However, it can already be seen that with an increasing number of partitions, these three implementations can scale with increasing thread count.

The next group contains three implementations: Collaborative Morsel Processing (CMP) with Processing Units, Hybrid, and Radix. The leader of this group is the Hybrid implementation, which, 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 materialize them fully. Furthermore, the slotted pages are allocated when needed, and each thread can request memory locations without waiting 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. This processing unit's other threads can then process these generated tuples.

The remaining CMP, OnDemand, and SmbLockFree implementations do not scale with increasing threads. As explained in Section 3.6.2, the CMP with exclusive partition ranges efficiency decreases per thread with an increase in the number of threads. Furthermore, there are at most as many exclusive partition ranges as partitions. Those additional threads cannot be used when there are more threads than exclusive partition ranges. For example, when using only two partitions, there are only two exclusive partition ranges. The remaining threads are not used because we can only assign those two partition ranges to exactly two threads. The OnDemand approach does not use a SMB and has to immediately 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 a compare-and-exchange operation. Thus, this approach has a significantly higher number of branch misses (1-3x), instructions (1-4x), and L1-cache misses (1-2x) in comparison with the other SMB implementations.

In figure 4.2, which 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 characteristics remain.

The LocalPagesAndMerge implementation does not significantly increase its through-

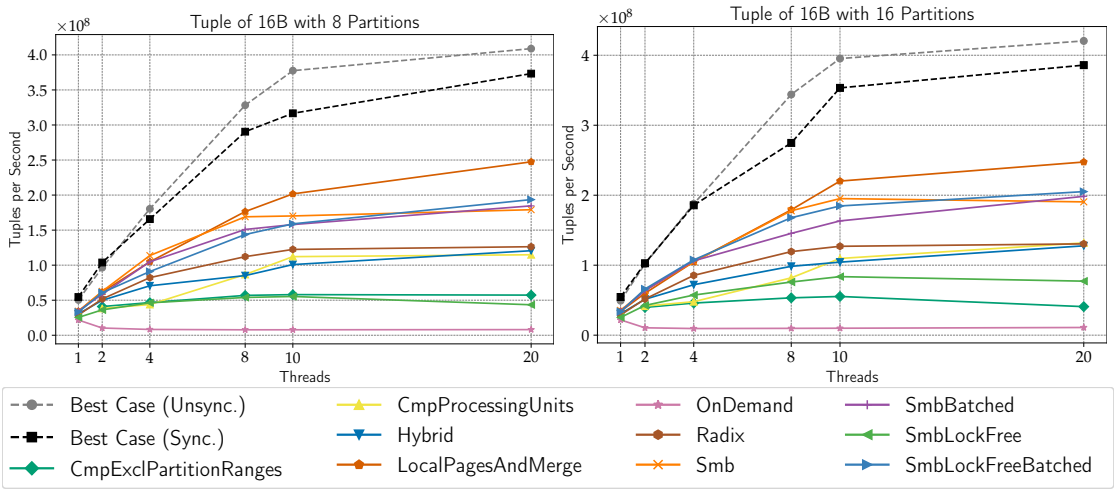


Figure 4.2: Benchmark Plots for Tuple of 16B with 8 and 16 Partitions

put, 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) misses and Translation Lookaside Buffer (TLB) misses increase as more memory locations 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 is now spread up between more slotted pages and thus across more locks or atomics.

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

The last group of implementations, which does not scale well, contains 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. Similarly, the number of logical cores is higher than the maximum number of partitions, which causes that a few threads do not get assigned a partition range und thus, cannot be

used. Also, the OnDemand approach struggles even more to write out the tuples, as the number of slotted pages increases with increasing partitions. This causes even more L1-cache-/LLC misses, as the Central Processing Unit (CPU) cannot hold all these memory locations in caches. Like 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 an increasing number of partitions, these atomic tuple-count values are less frequently requested by multiple threads. Thus, with increasing partitions, this implementation starts to improve its scaling.

Performance with many Partitions

After comparing the different implementations on 2 - 16 partitions, we now evaluate the performance on 32 and 1024 partitions. This high number of partitions creates new challenges to the problem, as memory consumption plays a big role in the performance. For example, when using 1024 partitions and our 5 MiB slotted page size, the initial memory consumption is already at 5 GiB. When thread-local slotted pages are used, this initialization of a slotted page per partition and thread already takes up 100 GiB on our 20 logical processors (10 physical cores + SMT) machine.

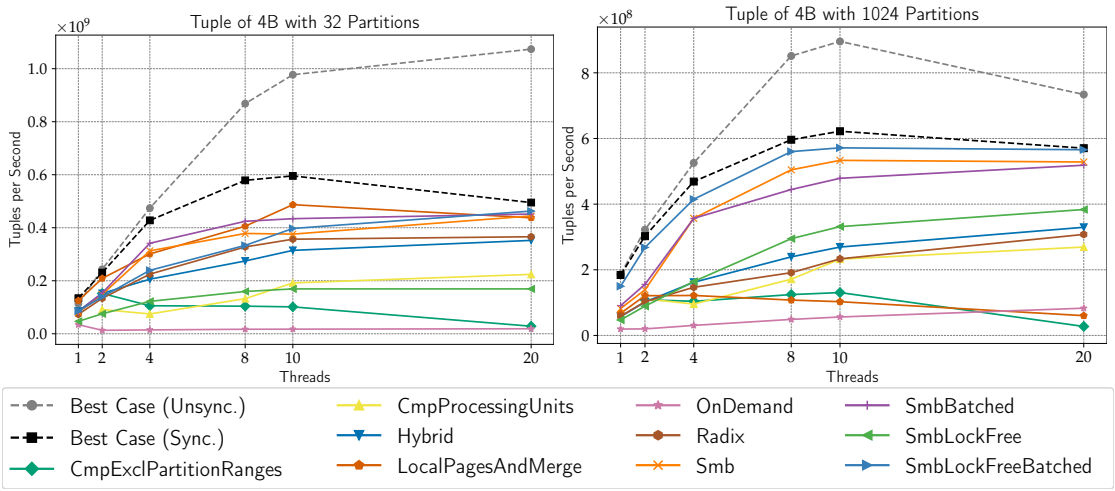


Figure 4.3: Benchmark Plots for Tuple of 4B with 32 and 1024 Partitions

In figure 4.3, we plot the performance of our implementations when using 4-byte tuples and 32 / 1024 partitions. In this 4-byte tuple case, we store the whole tuple in the slot section as the first 4 bytes construct the tuple key.

Like the few partition cases, the LocalPagesAndMerge-implementation processes the most tuples per second in the 32 partition case. In contrast, in the 1024 partition

case, we can see that the total amount of unnecessary heap allocations and non-shared memory accesses causes performance to degrade and lead to a negative scaling.

Like in the previous figures, most SMB-based implementations (Smb, SmbBatched, and SmbLockFreeBatched) scale very well while still struggling to keep up with the LocalPagesAndMerge in the 32 partition case. The two lock-based approaches outperform the lock-free implementation in the 32 partition case, as they have slightly fewer branch misses and an overall higher instructions per cycle (IPC). In contrast, the batched lock-free implementation processes more tuples in the 1024 partition case, as it has higher IPC and less LLC misses.

The next two best implementations in the 32 partition case are the Hybrid and Radix implementation. They can process slightly fewer tuples when using 32 partitions, whereas, with 1024 partitions, the gap between this and the previous group gets huge. When using the 32 partitions, the Radix approach performs better as it only has half as many branch misses than the Hybrid implementation. In contrast, the Hybrid implementations have 20% higher IPC in the 1024 partition case and thus perform better there.

The remaining four implementations (CmpExclusivePartitionRanges, CmpProcessingUnits, OnDemand, SmbLockFree) struggle to scale with increasing threads when using 32 partitions. The CmpExclusivePartitionRanges has yet again extremely high numbers of branch misses due to the partition range exclusive tuple processing. In contrast, the CmpProcessingUnits does scale with the number of threads in the 1024 partition case, but still up to 100x more branch misses than the best-performing implementations. The OnDemand approach does have slightly fewer branch misses than the previous two implementations. However, due to the single tuple write-out, it has up to 100x more TLB misses than the other implementations. Lastly, the lock-free non-batched SMB implementation also struggles to perform in the 32 partition case but does work surprisingly well with 1024 partitions. The branch and LLC misses are up to 3x more frequent than the other SMB implementations. The busy-waiting atomic accesses on the tuple-count variable of each slotted page cause the increase branch and LLC misses.

In figure 4.4, we use tuples with a size of 16 bytes instead of 4 bytes like in the previous figure. As we now use a tuple size greater than 4 bytes, we must also write the remaining bytes in the data section at the end of the slotted page.

Like the previous benchmark plots, the LocalPagesAndMerge implementation processes the most tuples per second in the 32 partition case. In contrast, when using 1024 partitions, this approach struggles to scale at all. Each added thread requires an initial allocation of 1024 slotted pages (5 GiB) and can then process tuples. In the end, these additional slotted pages must be sorted and merged, significantly increasing the cost of this final phase.

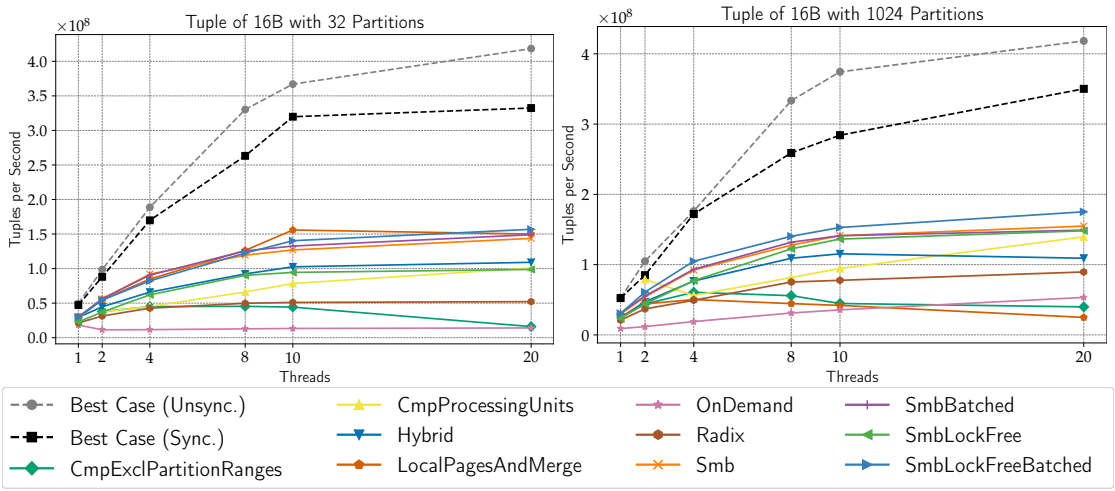


Figure 4.4: Benchmark Plots for Tuple of 16B with 32 and 1024 Partitions

The overall best-performing group is the four SMB implementations. Besides the slightly slower non-batched lock-free implementation, they can keep up with the LocalPagesAndMerge implementation in the 32 partition case. When using 1024 partitions, these four implementations outperform all other implementations by processing around 50% more tuples. The difference between the groups is that the lock-free non-batched variant has more branch misses, whereas the batched lock-free variant has slightly fewer LLC misses than the two lock-based approaches.

The next group of implementations is the following three that scale with increasing threads but not as well as the SMB variants: CmpProcessingUnits, Hybrid, and Radix. The CmpProcessingUnits approach has significantly higher cache misses due to its selective write-out, significantly slowing down this implementation. The Hybrid variant performs slightly worse than the SMB implementations due to slightly more LLC misses. Lastly, the Radix implementation struggles to scale in the 32 partition case but performs better when using 1024 partitions. It has around 50% more L1-cache-/LLC misses than the Hybrid approach, which slows down this approach.

In the non-scaling group, there are two implementations: CmpExclusivePartitionRanges and OnDemand. Like the previous benchmarks, the CmpExclusivePartitionRanges per thread efficiency decreases due to the more selective tuple write-out. The more selection write-out is caused by the increase of threads, which decreases the size of each exclusive partition range. Thus, the likelihood of processing a tuple that belongs to a certain partition range decreases. The OnDemand struggles to scale, as it does not use any form of buffering, which causes high contention on locks and more frequent accesses on slotted pages.

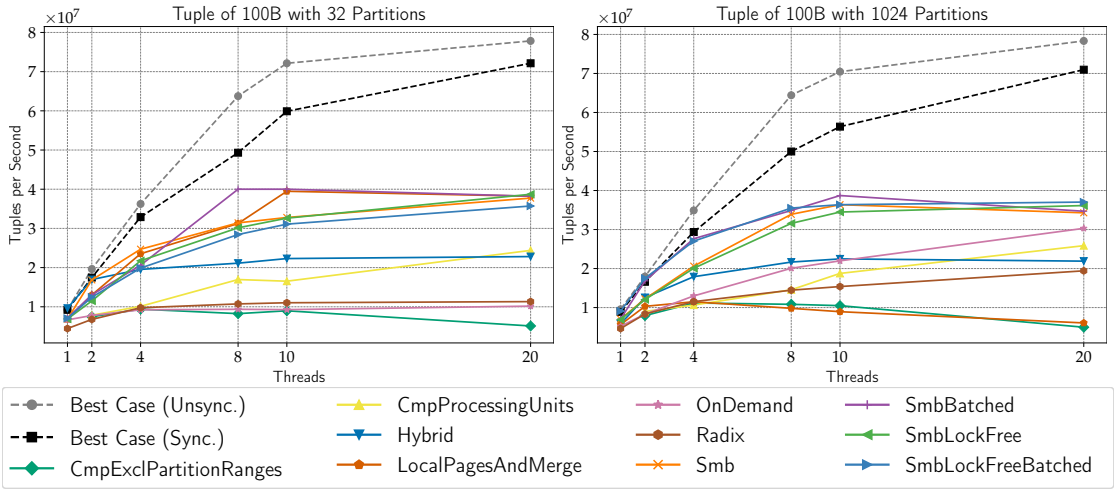


Figure 4.5: Benchmark Plots for Tuple of 100B with 32 and 1024 Partitions

In figure 4.5, we use 100-byte tuples instead of 16 bytes. Due to the bigger tuples, the cache lines and the SMBs can hold fewer tuples. This makes the write-out process more expensive and cache misses more likely.

Unlike previous benchmarks, the LocalPagesAndMerge approach does not process the most tuples in the 32 partition case. This time, the SmbBatched variant can outperform all other implementations. Nonetheless, the LocalPagesAndMerge scales still very well with increasing threads in the 32 partition case but has up to 2x more TLB misses than the SmbBatched variant. Like in the previous benchmarks, the LocalPages-AndMerge approach struggles to perform when using 1024 partitions. As each newly added thread requires the allocation of 1024 slotted pages (5 GiB), and the slotted page merge phase becomes more expensive, we observe that performance worsens as more threads are added.

The next group, which scales very well with increasing threads, are the four SMB implementations: Smb, SmbBatched, SmbLockFree, and SmbLockFreeBatched. Overall, all four implementations have similar perf measurements, just the SmbBatched has slightly fewer LLC misses and branch misses.

Hybrid and CmpProcessingUnits form the next group, as they scale with increasing threads in both partition cases, but are significantly less performant than the previous SMB group. The Hybrid approach causes 3x more LLC misses, whereas the CmpPro-cessingUnits has up to 20x more branch misses than the SmbBatched variant. These two significantly increased performance counters are causing a substantial performance decrease.

The next group consists again of two members: OnDemand and Radix. Both

approaches cannot scale with more threads in the 32 partition case, but perform resonable when using 1024 partitions. The OnDemand approach performs substantially better in the 1024 partition case, as the tuple write-out is more expensive in the 100-byte tuple case. This more expensive write-out and the fact that we are using 1024 partitions reduces the contention on the shared slotted page locks significantly. As we only hold locks during the fetching of the tuple index, the more expensive tuple write slows down the write enough so that the locks are rarely accessed simultaneously. The performance of the Radix implementation does increase due to a similar load-balancing effect. In the Radix implementation, every thread has to use its histogram to request write-out locations. We use a lock for each partition to make the implementation more scalable. We have to hold the lock during the allocation of a slotted page, as it will frequently be the case that the following thread wants to request a write-out location on this newly allocated page. By increasing the partitions, these locks are load-balanced as each thread starts at a different start partition and then requests a memory location for each partition.

Like in the previous benchmarks, the CmpExclusivePartitionRanges again cannot scale up, as the size of the exclusive partition ranges decreases with increasing thread count. This means that for a given thread, the likelihood that the next tuple must be placed in a partition of its assigned exclusive partition range decreases. This causes vast numbers of branch misses, and as all threads have to process all tuples, this implementation struggles to scale.

Memory Consumption

We used the heap profiler Massif within Valgrind 3.24 to measure the heap memory consumption of our implementations. In our memory benchmark, we partition 67.2 Mio. tuples of 16 bytes ($\approx$ 1 GiB) into 32 partitions. Since the tuple keys are distributed uniformly, the implementations store the tuples on ten 5 MiB slotted pages per partition. To highlight the impact of the benchmark parameters on the heap usage, we executed the benchmark using 40 threads.

In figure 4.6, we visualized the peak heap usage of each implementation. The partitioning of 67.2 Mio. tuples requires 10 slotted pages per partition. These 10 slotted pages per partition sum up to around 1.46 GiB of heap memory, which each implementation must use and acts as a lower bound in the plot. To better visualize the peak memory consumption of each implementation, we subtract the memory needed for the minimum number of slotted pages and show the additionally used heap memory above the bar of each implementation.

All implementations, except LocalPagesAndMerge and Radix, rely on a few small heap allocations (vectors, SMBs). The OnDemand implementation uses the least amount

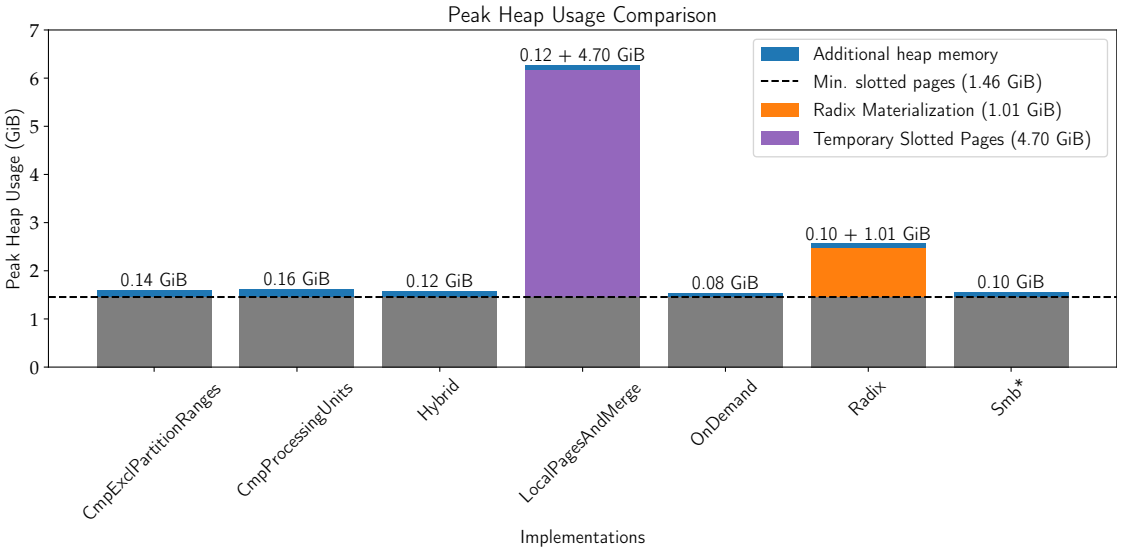


Figure 4.6: Peak Heap Usage when using 32 Partitions, 40 Threads and 67.2 Mio. 16B Tuples (1 GiB)

of heap, as it does not use any SMB. In the implementations that only use a few heap allocations, the two CMP approaches use the most heap. They both use SMBs, and their producer-consumer approach uses vectors.

The Radix implementation must fully materialize the 1 GiB of tuple data before writing tuples to the slotted pages. This materialization allocation directly scales with the total size of the incoming tuples. The Hybrid approach avoids this by only materializing small chunks and assigning them to threads.

The LocalPagesAndMerge approach uses the most heap memory due to the thread-local slotted pages. As explained in the space complexity analysis in 3.8.2, we want to emphasize that this additional heap memory consumption does not scale when increasing the incoming tuple count. Instead, it results from each partition's last, not fully used, thread-local page. These pages must be merged, so we pass the minimal number of slotted pages to the next operator. Since we can have at most one not fully used thread-local slotted page per partition and thread, the number of slotted pages can be upper bounded by a constant:

$$c_{\text{LPAM}} \leq (\text{Partition count}) * (\text{Thread count}) * (\text{Size of a slotted page}) \tag{4.2}$$

This constant c_{LPAM} must be considered when using the LocalPagesAndMerge approach, as it can be several GiB, as seen in figure 4.6.

4.3 Comparison with Stream Processing Systems

We compare our shuffle operator with keyBy operator of the batch and streaming processing engine Apache Flink[23]. This keyBy operator partitions an incoming data stream into disjoint partitions based on a key. We use the Java 19 Client and version 1.20 of Apache Flink in the following benchmark. Furthermore, we run the Apache Flink cluster on a single machine but let it use all available cores.

Like our implementations, we generate tuples on demand for a data stream. We apply the keyBy operator on this data stream by using the tuple key and our partition function (see Section 3.1). Then, we collect the count of records per partition.

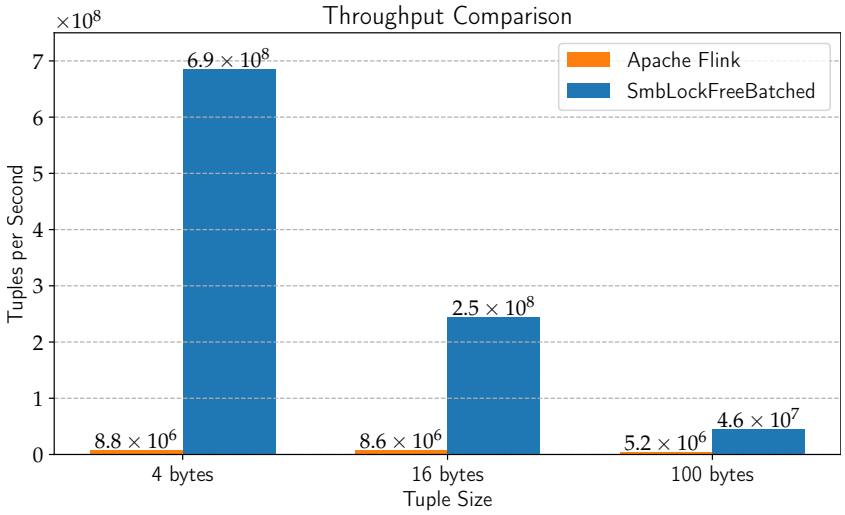


Figure 4.7: Tuples per Second Comparision when using 1024 Partitions

In figure 4.7, we compare the throughput of Apache Flink to our best-performing implementation, SmbLockFreeBatched. Across all tuple sizes, the SmbLockFreeBatched approach performs significantly faster and requires only one-third of memory in comparison with the Flink approach.

The results of this benchmark are unexpected. The Apache Flink benchmark simplifies to a frequency map. For each tuple, we apply the partitioning function and then increment the frequency of the corresponding partition by one. These two operations allow for heavy parallelization using a concurrent hash map. However, we must consider that Apache Flink is written in Java and Scala, whereas our implementation uses C++23 and compiles into optimized native machine code.

5 Conclusion

5.1 Conclusion

5.2 Future Work

Abbreviations

CPU Central Processing Unit

TLB Translation Lookaside Buffer

LLC Last Level Cache

IPC instructions per cycle

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: *SIGMOD Conference*. ACM, 2015, pp. 17–30.

[3] S. Chu, M. Balazinska, and D. Suciu. "From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System." In: *SIGMOD Conference*. ACM, 2015, pp. 63–78.

[4] G. Andrade, D. Griebler, R. P. dos Santos, and L. G. Fernandes. "A parallel programming assessment for stream processing applications on multi-core systems." In: *Comput. Stand. Interfaces* 84 (2023), p. 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 (2019), 17:1–17:45. 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: *VLDB J.* 11.3 (2002), pp. 198–215.

[8] M. Bandle, J. Giceva, and T. Neumann. "To Partition, or Not to Partition, That is the Join Question in a Real System." In: *SIGMOD Conference*. ACM, 2021, pp. 168–180.

[9] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware." In: *ICDE*. IEEE Computer Society, 2013, pp. 362–373.

[10] S. Schuh, X. Chen, and J. Dittrich. "An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory." In: *SIGMOD Conference*. ACM, 2016, pp. 1961–1976.

[11] T. Neumann and M. J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance." In: *CIDR*. www.cidrdb.org, 2020.

[12] A. Kemper and T. Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." In: *ICDE*. IEEE Computer Society, 2011, pp. 195–206.

[13] T. Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware." In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550.

[14] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich. "On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning." In: *Proc. VLDB Endow.* 8.9 (2015), pp. 934–937.

[15] O. Polychroniou and K. A. Ross. "A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort." In: *SIGMOD Conference*. ACM, 2014, pp. 755–766.

[16] Z. Zhang, H. Deshmukh, and J. M. Patel. "Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities." In: *CIDR*. www.cidrdb.org, 2019.

[17] P. A. Boncz, S. Manegold, and M. L. Kersten. "Database Architecture Optimized for the New Bottleneck: Memory Access." In: *VLDB*. Morgan Kaufmann, 1999, pp. 54–65.

[18] S. Lee, M. Jeon, D. Kim, and A. Sohn. "Partitioned Parallel Radix Sort." In: *J. Parallel Distributed Comput.* 62.4 (2002), pp. 656–668.

[19] W. Effelsberg and T. Härder. "Principles of Database Buffer Management." In: *ACM Trans. Database Syst.* 9.4 (1984), pp. 560–595.

[20] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. "Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited." In: *Proc. VLDB Endow.* 7.1 (2013), pp. 85–96.

[21] O. Polychroniou and K. A. Ross. "A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort." In: *SIGMOD Conference*. ACM, 2014, pp. 755–766.

[22] U. Drepper. *What Every Programmer Should Know About Memory*. Accessed: 2025-02-11. 2007.

[23] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine." In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38.