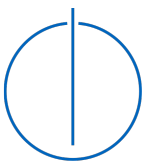
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Handling Skew in Morsel-Driven Hash-Joins

Parker Holland Timmins



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Handling Skew in Morsel-Driven Hash-Joins

Behandlung von Datenungleichverteilung in parallelen Hash-Joins

Author:	Parker Holland Timmins
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Philipp Fent, M.Sc.
Advisor:	Altan Birler, M.Sc.
Submission Date:	November 15, 2023

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

Munich, November 15, 2023 Parker Holland Timmins

Acknowledgments

Completing this thesis is only possible due to the support and guidance of a few individuals, without whom this achievement would not have been possible.

I am very grateful for the ideas and insight provided by my advisor, Altan Birler. These have been invaluable. Despite the short time we have known each other, I have learned much from Altan.

My thanks to my advisor, Philipp Fent, extend beyond this thesis. He has been a mentor in my education in database systems. I cannot thank him enough for the guidance and knowledge he has given me over the past year and a half. With my studies coming to an end, I will miss our Thursday morning meetings.

I quite literally could not have completed my master's degree without the help of my wife, Eloisa. Without her encouragement, I would not have thought to start my studies and certainly would not have finished them. It is her support that has enabled the opportunity I've had to study database systems in Munich.

Abstract

Hash Joins are the dominant join technique in modern database systems; they are highly efficient and perform well on a wide range of data. Unfortunately, their performance can decline in the presence of skew.

Hash joins often use separately chained collision lists, which can become excessively long when build-side keys are highly skewed. During the probe stage, iterating through long collision lists results in poor cache performance. If the probe relation is also skewed, repeated linked list iteration amplifies this behavior.

Morsel-driven parallelism is an efficient method to achieve high concurrency in modern relational database systems. The combination of Morsel-driven parallelism with hash joins using chained collision lists causes a second problem — low thread utilization. When the build relation is highly skewed, one morsel may produce far more results than another morsel. If the imbalance between morsels is high enough, the thread processing a long-running morsel will finish long after other threads. The query is delayed until the slow thread finishes and resources are utilized poorly.

The issues of poor cache performance and low thread utilization can drastically reduce the performance of hash joins on skewed data. To rectify these issues, we introduce two techniques: Node Compaction and Sub-morsel Stealing. Node Compaction gathers skewed tuples and copies them into dense arrays of tuples, improving cache utilization during probe. Sub-morsel Stealing allows one thread to help another thread join skewed morsels, improving thread utilization.

Using these techniques, we improve the execution time of queries on many skewed workloads. On a large subset of the Cardinality Estimation benchmark, these methods achieve a mean speedup of 49%, with a maximum speedup of over 87x. At the same time, we show that these techniques do not cause a significant reduction in performance on non-skewed workloads, such as TPC-H.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

2 Problem Setting **6**
2.1 Hash Joins . . . 6
2.1.1 Umbra's Hash Join . . . 6
2.2 Execution Model . . . 8
2.2.1 Interpreted / Pull-based . . . 8
2.2.2 Compiled / Push-based . . . 9
2.3 Parallelism . . . 10
2.3.1 Volcano-style Parallelism . . . 10
2.3.2 Morsel-Driven Parallelism . . . 11
2.4 The Problem with Skewed Data . . . 12
2.4.1 Poor Cache Performance . . . 12
2.4.2 Poor Thread Utilization . . . 13
2.5 Prevalence of Skewed Queries . . . 16
2.5.1 Uniform Data . . . 17
2.5.2 Key / Foreign Key Joins . . . 18
2.5.3 Foreign Key / Foreign Key Joins . . . 18
2.5.4 Graph Datasets . . . 19

3 Related Works **20**
3.1 Hash Joins . . . 20
3.2 Skew-Optimized Hash Joins . . . 20
3.3 Skew Recognition Techniques . . . 21
3.4 Linked List Optimization . . . 22

4 Improving Cache Performance **24**
4.1 Linked Lists for Collision Resolution . . . 24
4.1.1 Effects of Skew on Collision Lists . . . 24

4.2 Improving Cache Locality . . . 25
4.2.1 Naive Collision Array . . . 25
4.2.2 Array Nodes . . . 26
4.3 Array Node Construction . . . 27
4.3.1 Restricting Compaction to Skewed Keys . . . 28
4.3.2 Local Aggregation . . . 29
4.3.3 Insertion Algorithm Example . . . 32

5 Improving Thread Utilization **35**
5.1 Morsel Splitting . . . 35
5.1.1 Defining Sub-Morsels . . . 35
5.2 Scheduling . . . 39
5.2.1 Picking Sub-Morsels . . . 40
5.2.2 Running Stolen Sub-Morsels . . . 41
5.2.3 Probing Sub-Morsels . . . 41
5.3 Concurrency . . . 44

6 Selecting Join Algorithms **47**
6.1 Compile-time vs. Runtime . . . 48
6.2 Compile-time techniques . . . 48
6.2.1 Distinct Count . . . 49
6.2.2 Self Join Size . . . 49
6.2.3 Effectiveness of Compile-time Sketches . . . 50
6.3 Probabilistic Counting . . . 50

7 Evaluation **53**
7.1 System Setup . . . 53
7.1.1 System for Cardinality Estimation Benchmark . . . 53
7.1.2 System for other Benchmarks . . . 53
7.2 Benchmarks . . . 54
7.2.1 Zipfian Micro-benchmarks . . . 54
7.2.2 TPC-H & JCC Benchmarks . . . 56
7.2.3 Cardinality Estimation Benchmark . . . 60

8 Discussion **65**

9 Conclusion **68**

List of Figures **70**

List of Tables **72**

Bibliography **73**

1 Introduction

Relational database systems provide an interface for manipulating and analyzing arbitrary tabular data. A key feature of relational database systems, as described by Codd [8], is that the internal representation of data and means of manipulation is hidden from users.

In other words, when given arbitrary data and queries, a database system should do the right thing. A user should not have to think carefully about how to craft a query so that it runs efficiently — the database should figure this out. Of course, some queries have inherently high asymptotic complexities. A cross-join of two relations of cardinality n must produce n^2 output tuples; thus, its execution time must be quadratic. Nevertheless, a database system should attempt to run such a query within a reasonable time.

With this in mind, we look at a scenario where database systems frequently fail to produce reasonable execution times. When relations with highly skewed keys are joined with a hash join, queries can be slow. There is a great deal of literature on this topic [30, 23, 25]. Unfortunately, the existing literature does not solve the problem for many modern in-memory systems. Specifically, hash joins featuring chaining for collision resolution are susceptible to poor cache utilization in the presence of skewed data. Additionally, such hash joins can exhibit poor thread utilization when combined with morsel-driven parallelism.

We look at a few example joins to understand how these issues manifest. We start with a simple description of a hash join consisting of just two steps:

1. Build: Insert the tuples from one relation into a hash table by their join keys.
2. Probe: For each tuple in the second relation, output all matching tuples in the hash table.

We can see an illustration of a hash join in Figure 1.1. At the top left of this figure is the first relation, known as the build relation. Different colors represent the tuple keys. To the right of this, we see a hash table with the tuples of the build relation inserted by their key. This hash table uses separate chaining, storing values in a linked list accessible from the hash directory. All tuples with matching keys are accessible through the same linked list in the hash table.

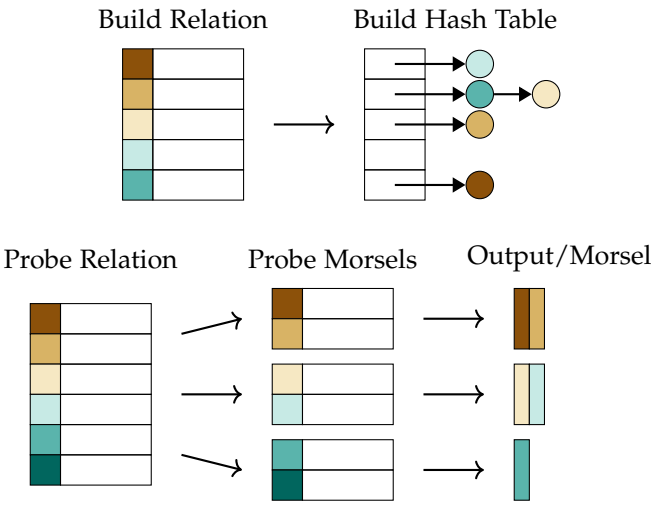


Figure 1.1: Illustration of a join between two relations with uniformly distributed keys. The output size is small and evenly distributed between morsels.

Below this, we see the second relation, known as the probe relation. Before performing the probe step of the above algorithm, the probe relation is split into units of work, known as morsels. A separate thread can process each morsel — a necessity as modern systems have many execution contexts. Processing each morsel involves looking up each tuple's key in the hash table. This requires hashing the key, accessing the index of the hash value in the hash directory, and iterating through the linked list of tuples at that index. Finally, the hash join outputs every matching pair of tuples from the build and probe relations. Pictured to the lower right are the number of output pairs per morsel. Each sliver represents a pair of build and probe tuples, with the color of the sliver being that of the pair's keys.

There are five output pairs for this join, which are reasonably evenly spread between the probe morsels — having 2, 2, and 1 outputs, respectively. Since the key values in the build and probe relations are unique, this is analogous to a join on primary keys. The result is a relatively small output, equal to the size of the smaller relation. Similar behavior would be seen if the keys were not unique but were uniformly distributed. There would be duplicate keys in this case, but the multiplicities would be low, resulting in a similar output size.

We now move on to Figure 1.2, where we join two highly skewed relations. We will see that the change in key frequency distribution drastically affects the output size and number of outputs per morsel.

This figure is similar to the previous one, with the difference that both relations are

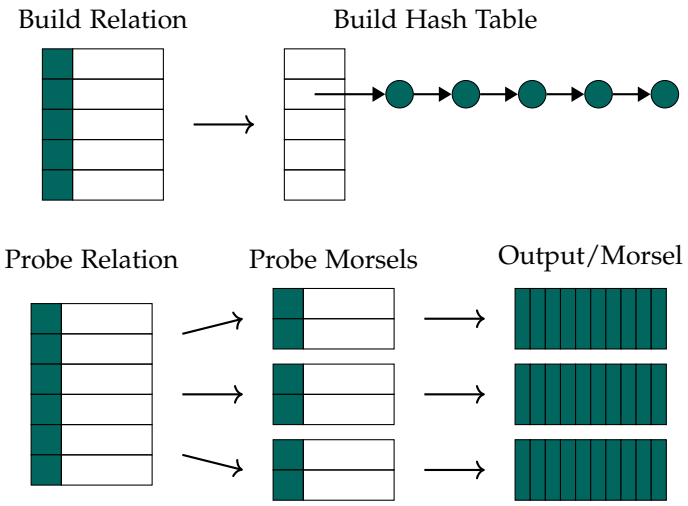


Figure 1.2: Illustration of a join between highly skewed build and probe relations. Due to the skew, the output size is large, though still evenly distributed between morsels.

now highly skewed, containing only green keys. In the lower right, we see that the skewed relations significantly increase the number of output tuples. Specifically, every tuple in the probe relation matches every tuple in the build relation, resulting in 30 output pairs.

Despite the same cardinality input relations, this join produces a much larger output than that of Figure 1.1. It is reasonable to assume that this query should take proportionally longer to complete. However, we should examine this assumption more critically. Consider the specific work being done during the probe stage of a hash join. For each tuple of the probe relation, the collision list at the appropriate offset in the directory is traversed. Each collision list node traversed requires a data access. Ignoring hash collisions, the number of accessed nodes equals the output size. Again, this may give the false impression that little improvement can be made since we cannot inspect fewer tuples than the output size.

However, all data accesses are not equal. There is a vast difference between accessing a tuple held in main memory versus one in the L1 cache. Unfortunately, linked lists have poor cache performance and poor space utilization. If the relations are large, many tuples read from hash table collision lists will be uncached, requiring main memory accesses. The large output size of skewed joins equates to a high number of main memory reads, resulting in high execution times. But what if tuples could be read from caches rather than from memory? This could result in a drastic performance

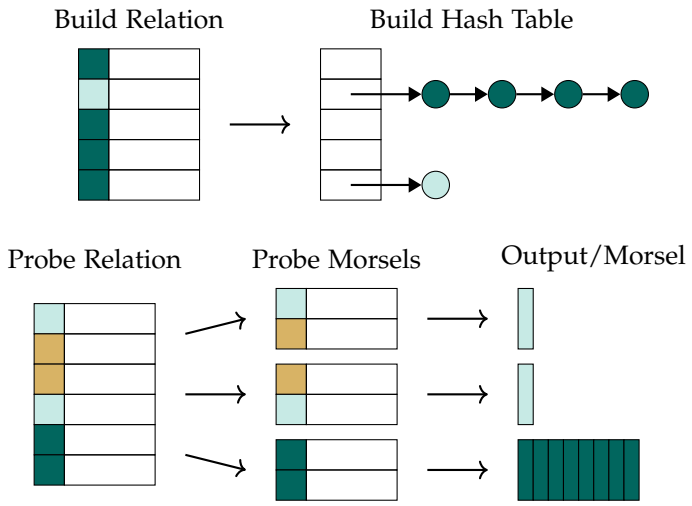


Figure 1.3: Illustration of a join between skewed build relation and low-skew probe relation. The output is not well distributed, with one morsel containing most of the results.

improvement. Thankfully, increasing the cache hit rate is relatively straightforward. By reorganizing tuples to be closer to each other in memory, we can significantly increase the amount of cached tuples, thus reducing expensive memory accesses. Showing how this tuple reorganization can be done to improve cache utility is the first contribution of this thesis.

Before moving on from the discussion of cache performance, it is worth taking a moment to consider why this was not a problem in the uniform case in Figure 1.1. In the skewed case, much of the time is spent probing the hash table rather than building the hash table. Whereas, in the uniform case, since the output is relatively small, less time will be spent probing the hash table. Thus, proportionally, more time will be spent building the hash table. While the probe stage dominates performance for skewed joins, the build and probe stages are of similar importance for uniform joins. Iterating collision lists when joining uniform relations does have poor cache performance. However, this is a worthwhile tradeoff, as separately chained hash tables have fast build performance. Throughout the rest of this thesis, we will see that there is usually a tradeoff between build and probe performance. The techniques we will use to improve cache performance will increase build times while drastically improving probe times for the right workload.

We now look at the second problem mentioned above: low thread utilization.

In Figure 1.3, we see a skewed relation joined with a more uniform relation. The

output size is 10, much smaller than the previous example, but the work is imbalanced in this case. Because the last probe morsel matches most of the build tuples while the other morsels each match one tuple, the last morsel requires more work. Since a separate thread is processing each morsel, this will result in one thread working for longer than the other two. The query can only be completed once the slowest worker thread is done. Though the system may be able to use the other threads to process a separate query in the meantime, they will likely sit idle instead. If the work could instead be shared more evenly between the threads, as in the other examples, all threads would finish sooner. A common technique for distributing work between threads is work-stealing, where under-worked threads may steal work to be done from overworked threads. Unfortunately, morsels are themselves the units of work that are stolen in morsel-driven systems. What is needed is a means to steal a "sub-morsel" of work from another thread. This is the second contribution of this thesis: we show how such Sub-morsel Stealing can be used during the probe stage of hash joins to reduce work imbalance between threads.

The remainder of this thesis describes our attempt at handling skew in morsel-driven hash joins. In Chapter 2, we provide the necessary background, describing hash-join and morsel-driven parallelism in detail. We then look at the previous work on this topic in Chapter 3. In Chapter 4, we describe Node Compaction, our method for improving cache utilization in skewed joins. Following this, we present Sub-morsel Stealing and show how it improves thread utilization in Chapter 5. In Chapter 6, we show how to identify skewed workloads. We then evaluate our techniques in Chapter 7. Finally, we discuss the results and reach conclusions in Chapter 8 and Chapter 9, respectively.

2 Problem Setting

In this section, we cover preliminary information needed to understand the optimizations proposed by this thesis. We start by describing hash joins in general and that of the Umbra database system in particular. Next, we look at Umbra's execution model, a pipelined, code-generating, push-based approach to processing an operator tree. This leads to a discussion of how query execution can be parallelized, focusing on how morsel-driven parallelism allows high thread utilization. Lastly, we will see how a system implementing the above features can fail to perform well on highly skewed data.

2.1 Hash Joins

Databases tend to support multiple join implementations, but arguably the most important is the hash join; they are efficient and perform well on many workloads [3]. There is significant literature on optimizing hash joins and hash tables [24, 4, 17]. Chapter 1 introduced a simple description of hash join operators. This section investigates a more realistic hash join, such as that used by a state-of-the-art database system. We will focus on the specific implementation used in the Umbra database system.

2.1.1 Umbra's Hash Join

The Umbra hash join has previously been described in detail [4, 17]. We will now cover the parts of this implementation relevant to this thesis.

Umbra's hash join uses a single global hash table. The table consists of an array of pointers known as the hash directory. It uses separate chaining rather than an open addressing scheme for collision resolution. Thus, rather than storing the tuples directly in the hash table, they are stored in linked lists accessible from the hash directory pointers. The hash table in the upper right of Figure 1.1 shows the general structure of a separate chaining hash table.

As opposed to the simple two-step hash join algorithm described in Chapter 1, Umbra uses three distinct steps, known respectively as materialize, build, and probe. We describe the steps in detail below.

Materialize Stage

The build relation is evaluated during the materialize stage, and all tuples are spooled into memory. Each tuple is padded with additional space to store a pointer. The pointer will be used during the build stage when inserting the tuple into the hash table. After processing the entire build side, we know the cardinality of the build relation. A hash directory can then be allocated; it is given a size slightly larger than the build cardinality to minimize hash collisions.

Build Stage

In the second stage, we iterate over the spooled tuples, inserting them in the hash table. This involves converting a tuple's hash value into a hash directory index and then inserting the tuple at the head of the collision list at the specified index. Insertion updates two pointers — setting the hash directory pointer to the newly inserted tuple, and setting the next pointer of the inserted tuple to the tuple at the current head of the collision list if one exists. Importantly, setting the pointer in the hash directory is done with an atomic exchange to guarantee that tuples inserted concurrently by other threads are not lost.

In addition to the pointer, each hash directory slot contains a 16-bit bloom filter of the keys held in the hash bucket. Any probe key which does not match the bloom filter can avoid traversing the collision list when looking for matches.

The materialize and build stages of Umbra's hash join can be combined if the size of the build side is known or can be estimated before materialization. This allows the hash table to be allocated before materialization, letting each tuple be spooled into memory and inserted into the hash table immediately. The benefit of this approach is that the data only needs to be traversed once.

Probe Stage

The final stage of the hash join is the probe stage. In this stage, the probe side of the join is processed, and for each tuple, we search for matches within the hash table. This involves finding the collision list associated with the index of the hash value and traversing the collision list. Since the list could contain tuples with colliding hash values, each tuple must be tested for equality with the probe key. Each matching pair of tuples is emitted to the next operator in the query tree.

The following section will discuss database system execution models, particularly Umbra's. With this additional context, the hash join algorithm's three stages will become more clear.

2.2 Execution Model

After a query is parsed into a tree of operators, the tree is optimized and evaluated. We ignore the optimization steps and focus on evaluation. Evaluation consists of producing the stream of tuples, the query's output.

There are several approaches to evaluation that can be organized across a number of axes — interpreted vs compiled, push vs pull-based, and tuple-at-at-time vs. block-based. Some of these attributes tend to be used together; for example, interpreted systems are usually pull-based. Nevertheless, these concerns are largely orthogonal, and there are systems that implement most of the possible combinations. We describe some standard evaluation models below.

2.2.1 Interpreted / Pull-based

The traditional approach to query execution is known as the iterator model or Volcano model [14]. This model is interpreted, pull-based, and tuple-at-a-time. In such a system, each operator implements a `next()` function, which emits a single tuple produced by the operator. Within the `next()` function, each operator calls the `next()` functions of its child operators in the query tree and consumes their tuples. Execution of the entire query is performed by repeatedly calling the `next()` method on the root of the operator tree. This approach is pull-based, as tuples are pulled from the bottom of the operator tree to the top.

The iterator model allows for elegant code but suffers from poor runtime performance. Specifically, iterator-based systems require many instructions per tuple produced [21]. A primary cause of this inefficiency is the polymorphism used to implement the iterator interface. Dynamic binding must be used to choose the specific `next()` method for a given operator. Worse, the resolution of the binding must happen for every tuple produced by every operator in the operator tree.

Block-Based Iterator Model

An optimization to the iterator model is to change from tuple-at-at-time processing to block-based processing. Such a system works similarly to the standard iterator model but passes chunks of tuples between operators rather than single tuples. This reduces the number of instructions per tuple, increasing performance but also code complexity.

Another downside of this approach is that the operators are no longer pipelined. In the standard iterator approach, each operator works in turn on a single tuple as it is pulled up through the tree. This allows the currently processed tuple to remain in the cache. Since the block-based model passed around a chunk of tuples, these tuples must

be materialized. Depending on the size of the block, this results in worse cache locality.

This leads us to compiled push-based systems which reduce evaluation overhead while keeping tuples pipelined.

2.2.2 Compiled / Push-based

In a compiled system, the query tree is used to generate machine code rather than being directly evaluated. For example, Umbra converts a query tree to LLVM code, which is compiled and executed. This compilation step removes the dynamic binding of the iterator model, reducing the number of instructions needed per tuple.

Evaluation in such a system amounts to running compiled functions with the base relations of a query as inputs. These functions directly loop over the tuples in the base relations. In this model, processing starts conceptually at the base relations and progresses to the top of the operator tree. Thus, this approach is classified as push-based rather than pull-based.

For example, Umbra's hash join algorithm described in Section 2.1.1 is conceptually push-based, as it describes iterating over base relations and pushing tuples up the operator tree. The build relation's tuples are pushed up into the join operator's hash table, and the probe tuples are pushed through the hash table to emit matches to the parent operator.

Pipelines

As mentioned previously, the iterator model benefits from being pipelined This means that a tuple can be passed between adjacent operators in the tree and used immediately. Such a design requires minimal copying and improves cache-locality.

The compiled approach reduces per-tuple overhead while still allowing effective pipelining. The query tree is compiled directly into loops over base relations. With each such loop, only the current tuple needs to be materialized, thus reducing copying and maintaining high cache locality. Of course, most queries will use multiple base relations and separate loops will be needed to iterate over each relation. In a compiled system, these loops are known as pipelines.

More generally, a pipeline is a contiguous piece of the operator tree that does not require tuples to be materialized. Each pipeline can be compiled into a single loop over a relation, with the work for every operator in the pipeline being a part of the loop body. This leads us to the notion of a pipeline-breaker. Sometimes, operators must materialize their inputs entirely. For example, the build relation of a join must be materialized entirely before proceeding with the probe side. We say that the hash join

operator is a pipeline-breaker for its build side, as the build side pipeline ends when its tuples are materialized in the hash join.

On the other hand, a hash join is not a pipeline breaker for its probe side. Tuples can be pushed from a probe side query through the hash join and then into the parent operators in the operator tree. Since the probe tuples are not materialized, they can stay in the cache.

In the following section, we look at the approaches database systems use to achieve high concurrency.

2.3 Parallelism

The above discussion has avoided parallelism, but this must be addressed as modern computers include dozens of discrete cores. Taking advantage of this parallelism is required in any system attempting to achieve high performance. The dominant technique to achieve parallelism in relational database systems is the use of Volcano-style parallelism [14].

2.3.1 Volcano-style Parallelism

The iterator execution model is also known as the Volcano model, and its method of achieving parallelism is Volcano-style parallelism. In a system with Volcano-style parallelism, operators are not multithreaded. Instead, a special exchange operator separates a tuple stream into multiple streams. Multiple, identical, single-threaded operators then process each such tuple stream. After the duplicated operators finish, the separated tuple streams are recombined by another exchange operator.

The primary benefit of such a system is its simplicity. Since each standard operator is single-threaded, a database system may be upgraded to handle Volcano-style parallelism with minimal changes to the relational operators. Most of the complicated multithread logic is encapsulated within the exchange operators.

Unfortunately, this simplicity of implementation comes at the cost of reduced performance; Volcano-style systems have two notable issues. First, as Volcano-style parallelism is often used in systems with interpreted, tuple-at-a-time execution, it shares the performance issues of such systems. The second issue with Volcano-style parallelism is that the level of parallelism for each operator is hard-coded within the operator tree. This is because there must be a duplicate operator node for each thread processing the operator. This reduces flexibility in the system, as threads cannot be reassigned between different queries or parts of queries. Additionally, since the work is evenly divided between duplicated operators, differences in processing speed can cause work imbalance. If one of the duplicated operators finishes faster than its siblings, the associated thread

will sit idle until all siblings have finished. This issue is particularly problematic in the presence of data skew.

These issues lead us to a more modern approach to multithreaded query execution — morsel-driven parallelism.

2.3.2 Morsel-Driven Parallelism

In a Morsel-driven system, the input tuples for each pipeline are broken up into chunks known as morsels; for example, each morsel might contain 10,000 tuples [17]. When the system is ready to execute a given pipeline, it decides how many hardware threads should be used. It then instantiates the threads and only releases them once the pipeline is completed. The number of threads used will be no more than the hardware thread count. Thus, there will be few context switches, and the threads can be considered hardware execution contexts. Each thread then loops, picking the next available morsel and pushing each tuple in the morsel through the pipeline. The threads run this loop until all morsels are exhausted.

Usually, threads consume morsels from a shared pool. For example, when processing a base relation, threads simply claim ranges of tuples within the relation. In other situations, threads steal morsels from each other. This is the case when materialized tuples are inserted in a hash table. It is preferable for the same thread that materialized a tuple to insert it in the hash table, as the tuple may be in the thread's cache. For this reason, after materialization, each thread picks morsels to insert out of the tuples that it materialized. However, if a thread finishes its morsels before others have finished theirs, it can steal morsels from the slower threads. Thus, morsel-driven systems are a form of work-stealing.

Morsel-driven systems have significant performance benefits. The use of work-stealing allows work to be evenly balanced between threads. This stops slow threads from stalling computations and allows the system as a whole to run faster. Additionally, by using fewer threads than hardware execution contexts, morsel-driven systems do not waste time context switching. They also allow for significant flexibility in thread allocation. Such a system could reallocate threads during a single pipeline execution, as any remaining threads can process the remaining morsels. This is an improvement over Volcano-style parallelism, which directly fixes the number of required threads in the operator, reducing flexibility.

Unfortunately, there is a cost to using morsel-driven parallelism — since operators are aware of parallelism, each operator must deal with parallelism explicitly. This adds significant code complexity, as many operations and data structures must be made thread-safe. For example, in the hash join operator, the hash table's insert function must be thread-safe. In Umbra, this is implemented by updating hash table pointers

with atomic compare and swap functions. Nevertheless, this is a worthwhile tradeoff, as morsel-driven systems can achieve high thread utilization.

Morsel Size

One aspect of morsel-driven systems worth discussing further is the morsel size. Since multiple threads call the morsel allocation code, it must be thread-safe, and frequent calls can lead to unwanted contention. Using larger morsels can result in less time being spent on the overhead of allocating morsels. But larger morsels can also cause work imbalance, leading to lower thread utilization. One solution is adaptive morsel sizing as described in [29]. This design starts with a small morsel and then increases the morsel size based on the time required to complete previous morsels. Though this method works well, we will see that it only solves some work imbalance issues caused by skewed data. The following section will describe the issues skewed relations can cause in hash joins, especially in morsel-driven systems.

2.4 The Problem with Skewed Data

Despite its dominance amongst join algorithms, hash joins are subject to poor performance on highly skewed relations. In particular, hash joins using separate chaining hash tables can result in poor cache utilization. Additionally, combining hash joins and morsel-driven parallelism can result in poor thread utilization. In this section, we look at both of these problems in detail.

2.4.1 Poor Cache Performance

There are some joins which, by their very nature, take a great deal of time. No optimization can make a cross-join on relations of size n have an asymptotic complexity smaller than $O(n^2)$, as the final output size has a cardinality of n^2. Unfortunately, very skewed relations can have similar behavior. Consider the inner join of a relation with only one unique key value with itself; the output is identical to that of a cross-join.

Assume we perform this join on a relation with $1,000,000$ tuples. The relation will first be inserted into the hash table. As only one key value exists, every tuple will be in a single linked list. Then, during the probe stage, every tuple in the relation will traverse this linked list.

We can compute the time it takes to complete the join described above, using an estimate of memory latency. A decent estimate for the latency of a memory reference is 100ns [10]. We ignore the additional time required to read data, focusing only on latency; we also assume the system is single-threaded. If the tuples are large, very

few may be cached. If we assume that no tuples are cached and every tuple must be accessed with a separate main memory reference, completing the join will take over 27 hours.

Of course, this is an extreme example; it is unrealistic that all tuple nodes are uncached and require a memory access. However, adjacent linked list nodes are likely not adjacent in memory. In this case, they cannot be read sequentially from memory and are likely not in the same cache line.

This leads to the obvious question: how can tuples in collision lists be read from the cache rather than from main memory? The answer is likewise obvious: the tuples of a collision list should be stored in an array rather than a linked list. This will achieve high cache utilization as adjacent tuples share the same cache line.

Cache misses on Zipf skewed data

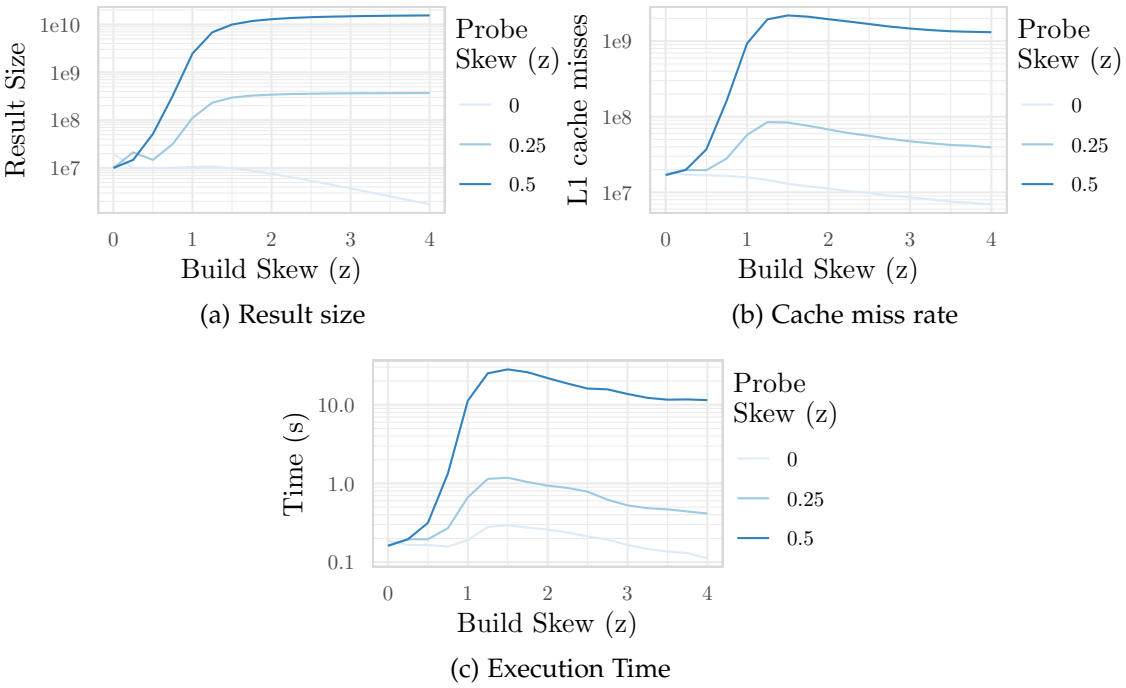
It is helpful to see actual data rather than the idealized example above. In Figure 2.1, we compare the execution time, cache miss rate, and result sizes of various joins. The keys of the build and probe relations are distributed with a Zipf distribution. The Zipf distribution is useful for simulating skewed real-world distributions, as described in Gray et al. [15]. We vary the build relation Zipf Z parameter between 0 and 4, while the probe relation Z takes values of 0, 0.25, and 0.5. A Z of 0 is equivalent to a uniform distribution. Both relations have 1e7 tuples, each consisting of two 64-bit integers. All three plots used log scales, as metric values become very large at high skew.

The plots show a strong relationship between the result size of a join and the L1 cache-miss rate. Furthermore, there is a correlation between these values and the total execution time of the query. This implies that the cache miss rate has a significant effect on the execution time of a query. Separating the cache miss rate from the result size of a join will lead to improvements in execution time. We will revisit these plots later in the thesis and show that the techniques provided do indeed change the relationship between these three metrics.

2.4.2 Poor Thread Utilization

In Figure 1.3, we saw how skewed data can cause an imbalance in work between threads in a hash join. We now look deeper at this problem, starting with a discussion of hash tables.

As previously mentioned, to access a value in a hash table, we convert the key to a hash value and then access the hash directory at the index specified by the hash value. Ideally, we would access the wanted item directly from this spot in the hash table. Unfortunately, this is not always possible. Since multiple keys can hash to the same



(a) Result size

(b) Cache miss rate

(c) Execution Time

Figure 2.1: Evaluation of baseline Umbra hash join on Zipf distributed build and probe relations.

offset in the hash directory, hash tables need a mechanism for accessing multiple items from the same offset. Open addressing hash tables handle this by storing colliding keys at other spots in the hash directory. A separate chaining hash table handles this by storing all values in a linked list accessible from the hash directory; values are not stored in the hash directory.

Traditionally, the keys of hash tables are unique — there is only one key-value pair for any given key in the hash table. However, the keys used to join database relations frequently have duplicates. For this reason, hash tables in hash joins allow the keys to be multi-sets. Moreover, the method used to handle hash collisions can be used to store duplicate keys. Thus, in a separate chaining hash table, key-value pairs with the same key will be stored in the same linked list. This works but can result in long collision lists. With a good hash function and a suitably sized hash directory, the chance of having many collisions in a slot is low. But, since keys in a relation can have an arbitrary number of duplicates, the length of the associated collision list can become arbitrarily long. This is not inherently problematic, but depending on the distribution of the probe relation, can cause an imbalance in work per morsel.

Morsel Work Imbalance

In Figure 1.2, we saw a join between two very skewed relations. Because the build relation has all identical keys, every tuple is stored in a single collision list in the hash table. Likewise, the probe keys were also all the same key. Thus, every probe key was required to traverse the single long collision list in the hash table. Since every probe key traverses the same collision list, processing every probe key requires the same amount of work. If we assume each morsel contains the same number of tuples, as in the example, each morsel will require the same amount of work to process. If the morsels are evenly distributed between threads, the amount of work per thread will be approximately equal. Thus, all threads will finish in a similar amount of time, and the query will have good thread utilization.

Now consider the situation in Figure 1.3. In this case, the build side is skewed, but the probe side is not. All the probe keys that match the tuples in the hash table are within a single probe morsel. Each tuple in this morsel will need to traverse the long collision chain. On the other hand, every other morsel will need to do minimal work. For each tuple in the other morsels, the appropriate slot in the hash directory will be found to be empty or containing a short collision list. Because most of the tuples are emitted by a single morsel, this morsel will take much longer to finish than the other morsels. The thread processing this morsel will take longer than other threads, and the query will only be completed once the slow morsel has been processed.

Effects of Probe Morsel Distribution

It is helpful to consider distributions of probe keys that can cause this work imbalance. As in the latter case above, to have a work imbalance, a small number of probe morsels must contain most or all of the high multiplicity build keys.

Consider a highly skewed build relation and a probe relation with unique keys. Assume a few keys account for the vast majority of build relation keys. If these keys appear in the probe relation and are in the same morsel, there will be an imbalance. The morsel containing these keys must traverse at least one long collision list, while most other morsels will find very few matches. On the other hand, if the few probe keys that will find many matches are spread throughout the probe morsels, there may not be an imbalance.

We can see an example of these situations in Figure 2.2b. On the left side of this figure, titled *Probe shuffled*, we plot execution time and thread utilization for the same benchmark shown in Figure 2.1. The right side of this figure, titled *Probe clustered*, shows queries with a slightly different probe relation. This relation is loaded into the database from a CSV sorted on the key value. Though the database makes no guarantee

of maintaining the order of the loaded CSV, in practice, tuples are clustered similarly in the base relation as they were in the associated CSV. Loading from a sorted CSV puts the keys that match the highly skewed build keys into a single morsel. On the left half of the figure, the CSV used for loading the probe relation was unordered. Thus, the few probe keys that emit the most matches are spread throughout the probe morsels.

Several parts of these plots are worth noting. For the clustered probe join, the execution time increases rapidly at a Z value of 0.5. A drop in thread utilization accompanies this. The rapid drop-off is due to two factors. At a Z value of 0.5, the number of duplicates in the build relation increases significantly, resulting in a few long collision lists. At this point, the probe keys that produce most matches now fit into a single morsel.

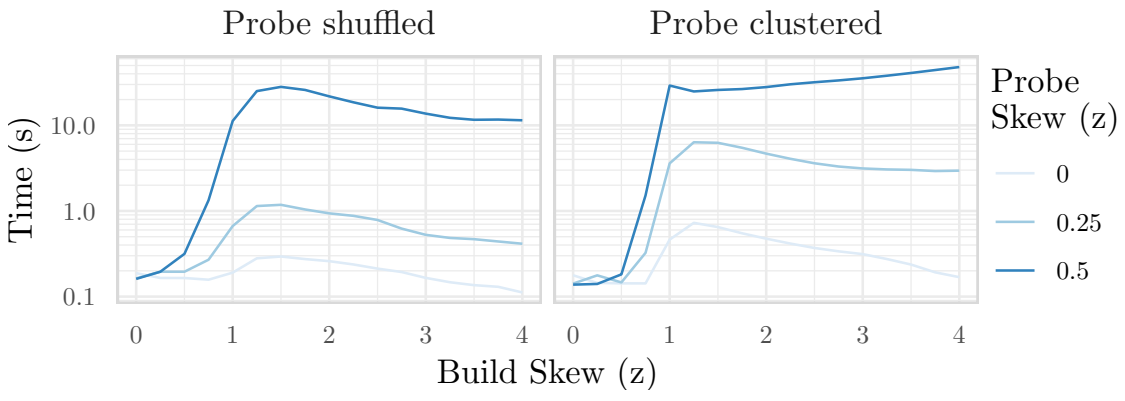
Adaptive Morsel Sizing

The Umbra database system uses adaptive morsel sizing [29]. This optimization helps mitigate some problems described above by keeping morsel sizes small when each morsel performs more work. Unfortunately, this does not solve the problem as a whole, as small morsels can still require an arbitrarily large amount of processing time. This is because the relation is split into morsels at the base of an execution pipeline and then processed by all code in the pipeline. Since the pipeline could contain joins, each tuple in the morsel could transform into many tuples higher in the pipeline.

The fundamental issue is that once a morsel is allocated to a thread, it cannot be subdivided. This is the primary contribution of this thesis — a means for morsel-driven systems to split morsels when they are found to be too large. Though we do not discuss adaptive morsel sizing in more detail, it is closely related to the work of this thesis. An interesting area for the future is to investigate the interaction between adaptive morsel sizing and the techniques of this thesis.

2.5 Prevalence of Skewed Queries

In Section 2.4, we showed experimentally that it is possible to design data and queries that cause the Umbra to exhibit the issues of high cache miss rate and low thread utilization. However, these data and queries are synthetic. It is reasonable to question whether there are more realistic queries that cause the same issues. This section shows queries on a well-known dataset that exhibit these issues.



(a) Execution time

(b) Thread Utilization as measured by kernel

Figure 2.2: Comparison of join with probe relation clustered by key versus randomly shuffled by key.

2.5.1 Uniform Data

The well-known database benchmark TPC-H uses a uniform data generation process. Due to uniformly distributed keys, skew does not pose a significant problem for queries on this dataset. The JCC benchmark is a skewed dataset based on the TPC-H benchmark [6]. It uses a skewed data generation process and adds correlated skew between columns. In addition to skewed data, the JCC benchmark includes queries that follow the template of TPC-H queries while using skewed attributes.

2.5.2 Key / Foreign Key Joins

There is another reason the TPC-H dataset and queries do not exhibit the issues described by this thesis. The issues with skewed data described in this thesis are more prevalent for build skew as this causes long collision chains. Database systems tend to choose the smaller of the two joined relations as the build side, so fewer tuples need to be materialized. A typical pattern for combining two relations involves joining a key in one relation with a foreign key in the other. Since keys must be unique, only the foreign key attribute has the potential to exhibit skew. Thus, in the join of a key and foreign key, to have a skewed build relation, the foreign key relation must be the smaller of the two relations. But this will never be true if every key value appears in the foreign key attribute. Even if this is not the case, the fact that a foreign key may appear multiple times means that the foreign key relation is likely the larger of the two. In the case of TPC-H, for every key/foreign key relationship, the relation containing the key is the smaller of the two.

It should be noted, of course, that relations joined in database systems are not necessarily base relations. They could be the results of arbitrary queries, for example, filtering predicates applied to a base relation. This provides a means for a key/foreign key join to occur where the foreign key appears in the smaller of the two relations and is thus on the build side of the join. Nevertheless, key/foreign key joins are less likely to exhibit build side skew than other join patterns.

2.5.3 Foreign Key / Foreign Key Joins

We turn now to joins between two foreign key columns. Because this allows duplicates on both sides of the join, this can result in inferior cache performance, as previously described. The duplicates in the build relation result in long collision lists and the duplicates in the probe relation cause these collision lists to be traversed repeatedly. But do such joins occur in practice?

We provide an example query in Listing 2.1, using the JCC benchmark dataset. It finds customers who purchased the same part as a given customer. Then, parts that a given customer has not purchased but that similar customers have can be recommended to the customer in question.

Unlike the key/foreign key joins described above, the equi-join key `partkey` is not a key in the joined relation. We ran this query three times on the baseline Umbra and a test Umbra with the methods developed in this thesis. The dataset was JCC at scale factor 1, and the system used is described in Section 7.1.1. The baseline version of Umbra completes this query in a minimum of 379 seconds and has 4.9e12 L1 cache misses for all three runs. Meanwhile, the version using the techniques described in this

thesis requires a minimum of 21 seconds, with a total of 5.5e10 L1 cache misses for all three runs. It is clear that the baseline version has poor cache performance on this query and that the methods in this thesis provide a way to mitigate this issue.

Listing 2.1: Query to find similar customers.

```
with purchases(custkey, partkey) as (
  select c_custkey, l_partkey
  from customer c, orders o, lineitem l
  where c_custkey = o_custkey
  and o_orderkey = l_orderkey
)
select count(*)
from purchases p1, purchases p2
where p1.custkey != p2.custkey
and p1.partkey = p2.partkey
```

2.5.4 Graph Datasets

It is worth noting that the above query joins the `purchases` relation with itself. Though sensible in the above query, this is perhaps not a common pattern. However, there is a domain where such queries are common — graph datasets.

In this domain, the data consists of graph vertices and edges. These can be modeled in a relational setting with an edge relation consisting of source and target columns. The vertices may not require a separate relation, existing implicitly as the source and target columns within the edge relation.

Since there is only the edge relation, all queries containing joins consist of joining the edge relation with itself. A simple example is a query that finds the neighbors of the neighbors of a given vertex. This consists of a join where the equality predicate matches the target of one edge relation with the source of another edge relation. We will see specific graph datasets and queries in Section 7.2.3.

In the next section, we review previous works related to the topics discussed in this thesis.

3 Related Works

This thesis touches on several topics, including hash join algorithms, skew handling in hash joins, skew recognition techniques, and linked list optimizations. Each of these areas has significant existing literature; we cover these related works in this section.

3.1 Hash Joins

The Grace hash join described how to perform a hash join on a multiprocess system when the build relation size exceeds memory [16]. It used two partition phases, once to assign tuples to processors and a second to select subsets of tuples from both relations, which are small enough to join in memory. The hybrid hash join optimized the Grace method by immediately building an in-memory hash table with the first bucket rather than first spooling it to disk as in the Grace hash join [26]. The partitioning technique used by both the Grace and Hybrid hash joins creates more partitions than there are processors. This allows multiple partitions to be distributed to a single processor, reducing the load imbalance caused by skewed keys. The same technique is used during the second partition phase to reduce the chance that skew causes an in-memory join to exceed the memory of a given processor.

3.2 Skew-Optimized Hash Joins

Walton et al. described a taxonomy of various types of skew that can affect a join algorithm [30]. This thesis only handles a subset of the types of skew described in the taxonomy, namely attribute value skew.

Dewitt et al. improved the existing methods by adapting them to handle skew explicitly [11]. This paper compared several algorithms but found the following to be the most performant. It first builds a histogram of the key distribution from a sample of the build and probe relations. If the histograms show neither relation is skewed, the algorithm uses a hybrid hash join. Otherwise, it uses a skew-specific algorithm, with the more skewed distribution as the build relation. The skew algorithm builds many partitions, using the histogram and range partitioning to build them approximately equally size. These partitions are then distributed to processors in a

round-robin fashion. Each processor then runs repeated in-memory hash joins on subsets of partitions that fit in memory.

Due to increases in memory size, the joins described above assume that build relation size is far larger than memory. In contrast, many modern join implementations are optimized for build relations that fit in memory. The join described in this thesis assumes that the build relation fits in memory. We now look at some related main memory joins, such as the radix-join, created by Manegold et al. [18]. It partitions similarly to the Grace and Hybrid hash join but with the purpose of improving cache locality rather than due to the size limitations of main memory. Our system does not radix partition but processes frequent keys thread-locally, improving cache locality during hash table construction.

Blanas et al. compare a radix partitioning join with a non-partitioning main memory join [5]. They observed that a non-partitioning join performed better than a partitioning join on skewed workloads. This was because skewed workloads cause an imbalance in partition sizes, resulting in some threads taking longer than other threads. Additionally, it was found that highly skewed workloads result in fewer cache misses. It should be noted that the tested workloads feature a primary key build relation and a skewed, Zipf-distributed probe relation. This differs from the workloads we focus on in this thesis — those with skewed build relations.

Leis et al. developed the Morsel-driven technique for achieving high parallelism in a main memory hash join [17]. This method schedules work in granular morsels, allowing work stealing of morsels between threads. They note that this results in balanced work between threads, mitigating some effects of skewed workloads. Though this is undoubtedly true, our work suggests that imbalance can still occur when build relations are highly skewed. We extend the ideas of morsel stealing to Sub-morsel Stealing, allowing skewed morsels to be split into smaller work units.

More recently, Rödiger et al. built Flow-Join, a method for handling skew in distributed hash joins [25]. The system finds heavy hitters with the Space-Saving algorithm, a technique for finding the approximate top-k [19]. It then broadcasts build tuples to other nodes to perform local hash joins. For the heavy-hitter build keys, it instead broadcasts the associated probe tuples, keeping the skewed build keys on the originating node. This leaves the skewed key spread across the system, allowing it to handle build keys which would overwhelm any single node.

3.3 Skew Recognition Techniques

A commonly occurring pattern in skew-optimized joins is the use of estimation techniques to identify the presence of skewed build keys. This thesis uses the HyperLogLog

sketch of Flajolet [12] and the AMS sketch of Alon et al. [1] to estimate the build key distinct count and second frequency moment. Umbra provides these sketches for all base relations. Based on these values, we attempt to decide before query runtime whether to use the standard hash-join or skew-optimized join. Our final test versions do not use this feature; we leave the application of this method as future work. Nevertheless, this idea is analogous to the histogram approach of Dewitt et al. [11], which is used to select between regular and skew-optimized algorithms.

The Space-Saving algorithm in Flow-Join is used for more granular decision-making, as it chooses how to handle specific skewed key values at runtime. Our system also does runtime per-key estimation by probabilistically counting the frequency of keys. Unlike Space-Saving, our approach does not attempt to find the heavy hitters but rather to find all keys above some frequency threshold. We take this approach as we believe that all keys of high multiplicity can benefit from improved cache utilization. Our probabilistic counting method is related to the Morris approximate counting technique, though we use less space while attaining less accuracy [20]. Finally, it is worth mentioning the Count-Min sketch [9]. We considered using this sketch to find the frequently occurring keys at materialization time. Unfortunately, because sketches cannot be resized and the number of tuples is unknown before materialization time, there is no way to instantiate a sketch and guarantee that the error rate will not exceed some bound.

3.4 Linked List Optimization

Previous works have identified issues with linked lists and provided methods to fix these problems. Shao et al. described the problems of inefficient space usage and long data dependency chains in linked lists [28]. They developed a technique known as unrolling, which stores multiple values per linked list node. The number of items stored per node is a constant in their design.

Bagwell introduced the VList, which allowed more efficient random access to linked lists [2]. In this design, multiple values are stored per node, but unlike unrolled lists, the nodes increase in size exponentially. The primary purpose of this node sizing is to amortize the cost of lookups by index, but they note that this scheme also improves cache performance.

Recent work uses similar ideas to optimize duplicate handling in Umbra's chaining hash join [27]. Their collision lists contain tuples with the same key within separate sub-lists. Within these sub-lists, each node can contain an array of multiple tuples. The first node for a given key is sized to fit within a single cache line. Subsequent nodes for the same key are increased in size exponentially. This results in a logarithmic decrease in the length of the linked list, significantly reducing cache misses. We use this method

for eager compaction. We discuss this method and how we modified it to our use case in Section 4.3.2.

In the next chapter, we describe the technique used to improve cache utilization in parallel hash joins.

4 Improving Cache Performance

The previous section discussed two issues that reduce the performance of hash joins: poor cache and thread utilization. We saw in Figure 2.1b that the L1 cache miss rate has a strong relationship with the total execution time of a join. A high cache miss rate is typical of a program with poor cache locality. Given the high join times in Figure 2.1c, we first address the issue of cache locality.

4.1 Linked Lists for Collision Resolution

As explained in Section 2.1, a separately chained hash table uses linked lists for collision resolution. Linked lists are known to have a potentially high cache miss rate [24]. With this in mind, why do we use a separately chained hash table rather than an open addressing one? Separately chained hash tables allow tuples to vary in size, unlike open addressing schemes, which require all tuples to be of equal sizes [17]. Perhaps more importantly, separately chained hash tables are fast to build since resolving a collision involves insertion at the front of the collision list. If build time accounts for a large proportion of the total join time, a fast build time can result in a fast total join time. Additionally, optimizations can mitigate the problems with linked lists. First, using a large hash directory reduces the probability of hash collisions. This decreases the average length of hash buckets, reducing the amount of linked list iteration and the associated cache misses. Second, Umbra uses a small bloom filter in the hash directory pointer, significantly reducing linked list iteration on probe misses [17]. These optimizations allow Umbra's chaining hash table to be highly performant.

4.1.1 Effects of Skew on Collision Lists

Surprisingly, high skew on the build side can sometimes improve chaining hash table performance. This can be seen in Figure 2.1c, where the join time decreases at very high skew. With higher skew, fewer hash directory slots must be accessed. As this reduces the size of the working set, cache utilization improves.

What, then, is the problem with skewed keys and chaining hash tables? Skew in both the build and probe relations have a multiplicative effect. Build side duplicates result in a hash table with long collision linked lists. Probe side duplicates cause these

linked lists to be iterated over repeatedly. This is the same effect seen in Figure 1.2 where build and probe side duplicates cause a large increase in output size and thus total work required.

This leads us to the goal of this section — to find a higher performance means of traversing collision lists. If there are probe side duplicates, we cannot decrease the need to repeatedly probe the hash table and potentially iterate across long collision lists. However, we can make the collision list iteration less costly. The following section describes a method to do this by compacting tuples into dense arrays and thus using caches more effectively.

4.2 Improving Cache Locality

When traversing a linked list, adjacent nodes are likely not in adjacent memory. Thus, subsequent nodes are frequently not cached. The solution to this problem is to place adjacent nodes into adjacent memory, that is to say, into an array.

4.2.1 Naive Collision Array

A naive solution to the problem of linked collision lists is to replace the linked lists with arrays. This could be achieved easily with the following algorithm. First, we materialize the tuples and build the hash table with regular collision lists. Then, we iterate over each collision list, replacing each with an array. This is done by first traversing a given list and recording its length. Next, we allocate enough memory to hold all the tuples and their hash values from a given collision list. Since there is no longer a null pointer to signify the end of the list, we allocate space for a length variable at the head of the array. After setting this length field, we re-traverse the list, copying tuples and hash values into the newly allocated memory. Finally, we replace the collision list pointer with a pointer to the array.

Consider how probing works in the above design. We start by going to the offset associated with the hash value in the hash directory. If a matching tuple exists, there will be a pointer to the collision array. At the head of the array is its length; this is used to bound iteration over the array. We now traverse the array, checking each hash value for matches and emitting matching tuples.

There are many workloads where this will perform better than the existing hash table implementation. The tuples and hash values can be read sequentially from memory. This will significantly improve cache performance, as subsequent adjacent tuples will likely be in the cache. Due to the sequential read pattern, adjacent tuples outside the current cache line can likely be prefetched into the cache.

Listing 4.1: Hash table node structures.

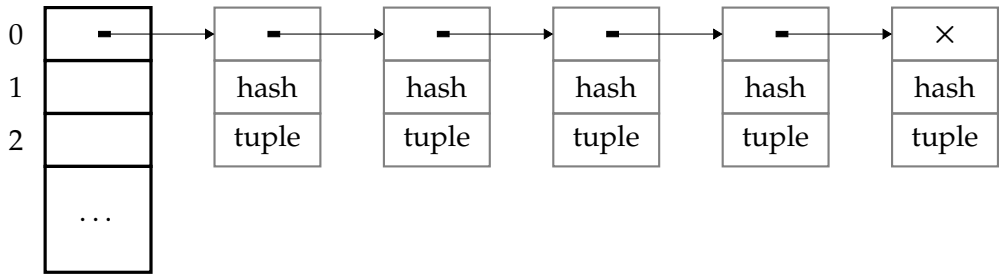
```
struct TupleNode {
  TupleNode* next;
  uint64_t hash;
  char tuple[TUPLE_SIZE];
}
struct TupleArrayNode {
  TupleArrayNode* next;
  uint16 length;
  uint64_t hash;
  char tuples[TUPLE_SIZE][length];
}
```

4.2.2 Array Nodes

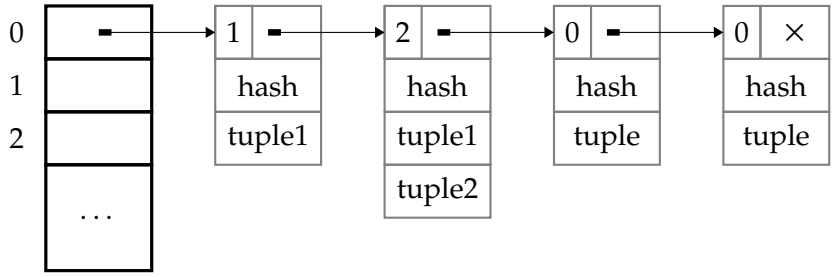
Though the above design will perform well on many skewed workloads, it has a few problems. First, it lacks a useful optimization. Each collision array contains many duplicated hash values. If every hash value in a given array were identical, it could be recorded once at the head of the array. This would save space, further improving cache utilization. However, due to hash collisions, there can be multiple hash values in a given collision array. An alternative is to use a separate array for tuples that share a hash value. These arrays can be linked together so they can be reached from their hash directory slot. This can be done by adding a pointer field before the length and hash fields and then linking these array nodes into a linked list. This is essentially the structure of the collision list nodes used in this thesis.

We explicitly define these nodes — TupleArrayNode in Listing 4.1. For comparison, we also define the structure of regular nodes. Each TupleArrayNode consists of a header and an array of tuples. The header contains a next pointer, a length field, and a hash value.

The actual implementation contains one difference from the above definition — the location of the length field. To save space, the length is stored in the upper 16 bits of the next pointer rather than using a separate field. This limits the length of tuple arrays to 2^{16}. Thus, in our design, a single array node will not necessarily contain all tuples in a given collision list that share a hash value. We will see in the next section that there are other reasons to relax this requirement. One notable feature of combining the length field into the next pointer is that TupleArrayNode and TupleNode headers have an identical memory layout. The length field will not be set in a standard TupleNode, but the next field, hash value, and first tuple will be located at the same offsets within the node. This is useful as it allows both types of nodes to be included in the same collision list. Standard and array-based collision list diagrams are provided in Figure 4.1a and



(a) Umbra's current collision list nodes, containing only a next pointer, the hash value, and the tuple.



(b) Collision list nodes containing a hash value, a next pointer, and an array of tuples. The top 16 bits of the next pointer contain the length of the tuple array. Non-array nodes have a length field value of 0 despite containing a single tuple.

Figure 4.1: Collision list node structure for different versions of the hash table.

Figure 4.1b, respectively. In the latter, it can be seen that the first nodes inserted in the list were standard nodes rather than array nodes. In the following section, we describe how these array nodes are constructed.

4.3 Array Node Construction

In Section 4.2.1, we described a naive solution to the issue of poor cache performance when hash joining skewed relations. Unfortunately, there is a problem with this algorithm. Namely, the time spent copying collision lists into compact arrays increases the time spent on the build stage of the join. On skewed workloads with highly skewed build and probe relations, the probe stage can take most of the execution time. In this case, spending more time building the hash table is a good tradeoff so that much less time can be spent probing it. However, if the workload is not very skewed, the build stage will likely take much of the execution time. In this case, probing a linked list will not take significantly longer than probing a collision array. Any time spent building

the collision arrays is wasted.

4.3.1 Restricting Compaction to Skewed Keys

The problem was eagerly compacting all collision lists into collision arrays. We can instead only build array nodes when there are many duplicates of a given key. For keys that only appear occasionally, it is not worth spending the time to copy the linked lists into more dense array nodes. This leads to a simple description of the insertion algorithm. If a key has yet to be seen many times, insert the tuple directly in the global hash table. If a key has been seen many times, aggregate it into compact array nodes. This algorithm is provided as pseudocode in Listing 4.2. After this algorithm has inserted all tuples, we add all compacted array nodes into the global hash table.

It remains to describe how to determine whether a key has been seen many times. When the hash directory is constructed, we can create a parallel array of counters. Every time we insert a key at an offset in the hash directory, we increment the counter at the same offset. If the counter is below a specified bound, we insert the tuple as usual; otherwise, we compact the tuple into an array node.

Since there are hash collisions, this will combine the counts of multiple keys, but this will be rare enough not to matter.

A more important concern is space usage. We must not slow down queries that do not have skewed keys. Incrementing and checking the counts needs to be very fast — the counters need to be cached. We can increase the chance that counters are cached by replacing them with smaller probabilistic counters. These probabilistic counters use a single bit per hash table slot. Rather than incrementing until the value exceeds some bound, for each tuple inserted in a given hash slot, we set the associated bit with a probability of $\frac{1}{duplicateBound}$. This can be seen in lines 10-11 of Listing 4.2. We discuss this issue in more detail in Chapter 6.

Listing 4.2: Algorithm to insert tuples in the global hash table or local skew table.

```
1  duplicateBound = 128
2  bool isSkewed[tableSize]
3  void* hashTable[tableSize]
4  threadlocal map<uint64_t, SkewTableEntry> localSkewTable
5
6  def insert(tupleNode):
7    if isSkewed[tupleNode.hash % tableSize]:
8      localSkewTable[tupleNode.hash].insert(tupleNode)
9    else:
10     if random(0, duplicateBound) == 0:
11       isSkewed[tupleNode.hash % tableSize] = 1
12     insert tupleNode at head of hashTable[tupleNode.hash % tableSize]
```

So far, we have yet to explain the process of compacting tuples into array nodes. We cover this in the next section.

4.3.2 Local Aggregation

During the insertion process, any hash values which is seen enough times will be determined to be skewed. These skewed tuples will be compacted by copying them into array nodes. We chose to do this compaction locally. This has a downside: duplicates processed by separate threads will not be compacted together. For example, consider a machine with 100 threads. If 100 duplicate keys arrive after the key has been determined to be skewed, each duplicate could be processed by a different thread. This would result in no improvement, and any time spent on the compaction algorithm would be wasted. Nevertheless, we did not find a means to perform compaction across multiple threads safely and efficiently.

The choice to compact locally drastically simplifies the compaction process. We need a mapping from the skewed hash values to the associated array nodes. Since we do not know the number of skewed hash values before insertion, we use a resizable hash table, which we call a skew table. Each thread maintains a separate skew table, which is an instance of the hopscotch map library [13]. The key in a skew table is the hash value of a tuple key, and the value is a SkewTableEntry. When a tuple is inserted in the skew table, we look up the associated SkewTableEntry and insert the tuple in the entry. This insertion can be seen in line 8 of Listing 4.2. We tested two methods of aggregation: lazy and eager compaction. These have different SkewTableEntry formats and insertion methods. We describe them both below.

Lazy Compaction

In lazy compaction, a SkewTableEntry consists of two lists: a list of regular tuple nodes that have not been compacted and a list of tuple array nodes that have been compacted. A definition of this SkewTableEntry is shown in Listing 4.3. When a tuple is inserted in the entry, we first check if the chain of single tuples is longer than some bound. If it is, we allocate an array node, copy the chain of single tuples into the array node, and insert it as the tail of the array node list. Then, we insert the new node into the single tuple list.

At the end of the hash table build stage, all the tuples in the skew table must be inserted into the global hash table. Any un-compacted single tuple chains must be compacted at this time. If a given hash value only has a short single tuple chain, we will skip compaction and insert the single tuple chain directly into the global table. This allows us to avoid compacting tuples, which are, in fact, not very skewed. We insert

the array nodes only after inserting any such short single tuple chains. This is done for two reasons. First, this is useful for Sub-morsel Stealing, as described in Chapter 5. By placing all single tuple nodes contiguously at the end of the chain, these can be treated as a single sub-morsel. Secondly, we improve branch prediction during the probe stage by not mixing single and array nodes.

The main benefit of the lazy compaction algorithm is its simplicity. Unfortunately, this simplicity comes at the cost of performance. This method must re-scan single tuple nodes when copying into array nodes. Thus, each compacted tuple is processed three times: once during materialization, once when appended to a single tuple chain, and once when copied into an array node. Ideally, we would only process each compacted tuple twice, once during materialization and once to compact into an array node. This can be achieved with the second compaction algorithm: eager compaction.

Listing 4.3: Lazy Compaction SkewTableEntry and insert function.

```
1  struct SkewTableEntry {
2    TupleNode* tupleHead;
3    TupleNode* tupleTail;
4    uint16_t numSingleTuples;
5    TupleArrayNode* chunkHead;
6    TupleArrayNode* chunkTail;
7  }
8
9  def insert(entry, tupleNode):
10   if entry.numSingleTuples == maxChunkLen:
11     newChunk = allocate(maxChunkSize)
12     newChunk.hash = tupleNode.hash
13     newChunk.length = maxChunkLen
14     insert newChunk at end of entry.chunkHead, entry.chunkTail list
15     clear entry.tupleHead, entry.tupleTail
16   insert tupleNode at end of entry.tupleHead, entry.tupleTail list
```

Eager Compaction

Lazy compaction has the problem that copied tuples must be traversed three times: once during materialization, once during initial hash table insertion, and once when copying into array nodes. We would prefer to immediately copy a tuple into an array node when inserted in a SkewTableEntry. Nevertheless, since we do not know how many tuples there are with the same key, it is difficult to pre-allocate an array node of the correct size. If we pick a constant size that is too small, cache performance will not be improved sufficiently. If we pick a constant size that is too large, some array spots may be unused, and we will waste space.

The solution is to create linked list nodes that increase in size exponentially. These were described by Bagwell [2] and implemented in recent work on the Umbra hash join [27]. In this latter work, all the nodes that share a given key value are placed within a sub-list of the collision list. The nodes contain multiple tuples and are allocated in exponentially increasing size.

We use the allocation scheme of [27] for eager copy compaction. Though our collision lists also contain sub-lists, ours are not guaranteed to contain tuples of a single key value. Their sub-lists are designed to allow skipping of tuples with different keys during the probe stage, while ours are used for Sub-morsel Stealing, as described in Chapter 5. Their method directly builds the exponentially sized array nodes on the global hash table. As previously described, we do this aggregation locally to avoid contention on the frequently updated nodes containing skewed keys.

We will now describe the compaction algorithm in more detail. The SkewTableEntry used in eager compaction can be seen in Listing 4.4. It only contains a list of array nodes; no single tuple nodes are stored. Upon insertion, we check if there is space in the chunkTail node. If there is no space, we must allocate a new node. We multiply the currentChunkLen variable by a constant to get the new chunk length and update currentChunkLen to this value. We used a growth factor of 1.5, but other values would likely have worked well. A new node is allocated of length currentChunkLen and inserted at the end of the array node list. Next, we update nextInChunk to 0, meaning the new tuple will be copied at an offset of 0 in the new chunk.

At this point, we know there is space for another tuple in chunkTail. We copy the new tuple into chunkTail at a tuple offset of nextInChunk, and increment nextInChunk.

Listing 4.4: Eager Compaction SkewTableEntry and insert function.

```
1  struct SkewTableEntry {
2    uint64_t currentChunkLen = 1;
3    uint64_t nextInChunk = 0;
4    TupleArrayNode* chunkHead;
5    TupleArrayNode* chunkTail;
6  }
7
8  def insert(entry, tupleNode):
9    if entry.currentChunkLen == entry.nextInChunk:
10     entry.chunkTail.length = entry.currentChunkLen
11     entry.currentChunkLen *= 1.5
12     newChunk = allocate(entry.currentChunkLen)
13     newChunk.hash = tupleNode.hash
14     insert newChunk at end of entry.chunkHead, entry.chunkTail list
15     entry.nextInChunk = 0
16   copy tupleNode.tuple to entry.chunkTail.tuples[entry.nextInChunk]
17   entry.nextInChunk++
```

Since the first array chunk is of length one, as an optimization, rather than allocating a new node, we can convert the inserted single tuple node into an array node by setting the length field from 0 to 1. Unfortunately, this optimization cannot be used with Sub-morsel Stealing, as will be described in Chapter 5.

As with lazy compaction, the compacted tuples must be added to the global hash table at the end of the build phase. Unlike lazy compaction, no copying is required as all tuples have already been copied to array nodes. In each SkewTableEntry, the final array nodes may not be full. In this case, the length of the final array node needs to be set to the correct value before inserting the array node list into the global hash table.

Unsurprisingly, this method performs better than lazy compaction. Both the *Compact* and *+Stealing* code versions used for evaluation in Chapter 7 use eager compaction due to the better performance of this technique. Nevertheless, we have included the lazy compaction method as its simplicity allows more flexibility when implementing Sub-morsel Stealing.

4.3.3 Insertion Algorithm Example

We finish this chapter with an example of the Node Compaction algorithm. The eager compaction method is used, though most examples would also apply were lazy compaction used instead. The description can be followed in Figure 4.2 starting at ①.

At ①, we have a new tuple to be inserted in the global hash table. We check the skewed bit set, finding a 0. This means that few duplicate hash values have already been inserted in the global hash table. Since the skew bit was not set, we insert the tuple directly in the global hash table at ②. At this point, we increment the associated bit with a probability of $\frac{1}{duplicateBound}$.

At ③, hash value 789 is found in the skewed bit set due to the previously inserted white tuple having set the skew bit. Rather than being inserted in the global hash table, it will be inserted in thread 1's local skew table. There is already an entry containing two array nodes for this hash value in the skew table; At ④, we see that the second node is not yet full. As the node is not full, the length must still be set in the node header. We now move to Figure 4.3, starting at ⑤. Here, we see that the tuple with hash value 789 has been copied into the array nodes. Since the node is full, the length field has been set to 2.

After all tuples have been inserted in either the global hash table or local skew tables, the skew tables need to be merged into the global hash table. At ⑥, the array node for hash value 789 in thread 2's skew table has been inserted in the global hash table. Finally, at ⑦, the same is done for thread 1's entries.

This ends the chapter on improving cache performance. The next chapter discusses how the proposed ideas can be extended to improve thread utilization.

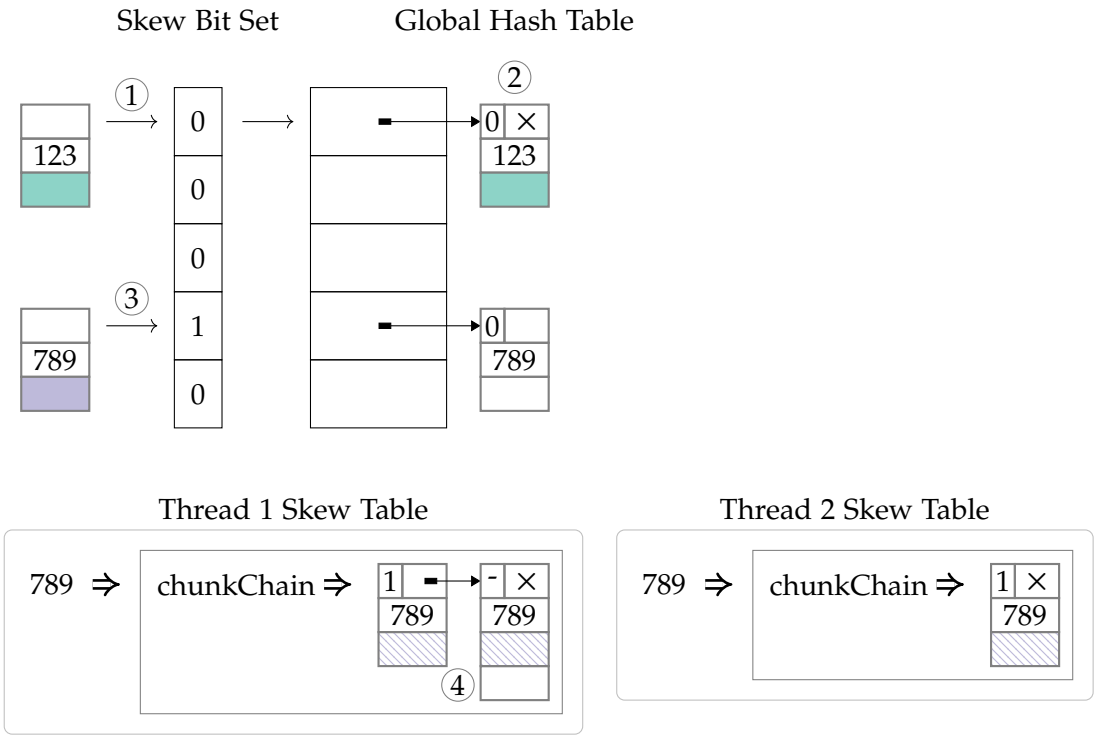


Figure 4.2: An example of the hash table insertion algorithm. The global hash table and the skew bit set are pictured above. Below, the thread-local skew table holds the tuples with potentially skewed hash keys. A description of the numbered steps is provided Section 4.3.3.

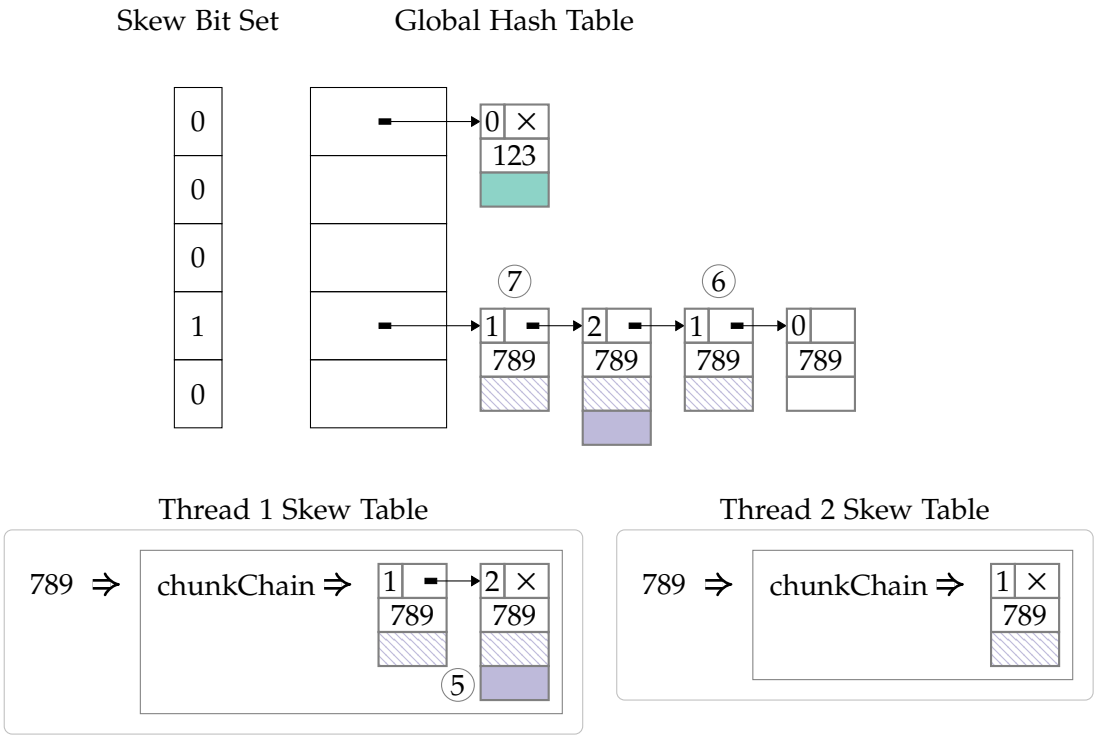


Figure 4.3: Insertion algorithm example continued

5 Improving Thread Utilization

As described in Section 2.4.2, a morsel consists of a set of tuples from a pipeline source, such as a table scan. A morsel is processed by executing it against the generated code for the pipeline. If the pipeline contains joins, a single tuple from the source can map to multiple tuples later in the pipeline. Thus, a single morsel can require an arbitrary amount of work. This can lead to an imbalance of work between threads, as one thread may process an expensive morsel while all others process cheap morsels.

This issue can be mitigated by reducing morsel size. However, this is a tradeoff as described in Section 2.4.2. Additionally, because a morsel can result in an arbitrary amount of work, any technique that requires a thread to complete all of a given morsel's work can result in imbalance. Thus, the solution is to let threads dynamically split the work of a morsel after that morsel has started processing. This chapter describes our technique for allowing threads to split morsels.

5.1 Morsel Splitting

In the Umbra database system, a morsel represents a piece of work to be processed. The specific definition of a given morsel depends on the code used to process it. Notably, the execution system does not know the binary representation of a morsel. When the execution system starts running a pipeline, it is provided with a `pickMorsel` function, which defines how to generate morsels. This function is specific to the type of data source at the base of the pipeline. For example, a table scan operator is a pipeline data source and provides such a function. The morsels this function returns consist of two integers defining a range within the table scan. Most morsel types have a similar definition, though this is not necessary. If we wish to split morsels into sub-morsels, we need a similar data type definition for sub-morsels.

5.1.1 Defining Sub-Morsels

We start with an example to motivate the following explanation. Assume we are hash joining two relations and have completed the materialization and build phases. During the probe stage, all threads but one have already completed their work and emitted all matches. The one remaining thread is still working through the one remaining morsel.

Let us assume this morsel is from a table scan and is defined by the range $[a, b)$. The remaining thread is currently probing with the tuple at index m, which is within this range. If another thread wishes to help this thread, it could offer to take some subset of $[a, b)$. Perhaps it takes the latter half of the remaining work: $[\frac{b-m}{2}, b)$. This is a reasonable way to split the morsel and has the benefit of representing a sub-morsel the same way as the parent morsel.

Nevertheless, we do not define sub-morsels in this manner. We have chosen to take a different approach for several reasons. First, because a single tuple in the data source can map to an arbitrary number of tuples higher in the pipeline, any technique that does not split the work of a single probe tuple can only partially solve the problem of work imbalance. Second, if the work required per tuple is relatively well distributed across the base data source, the adaptive morsel sizing described in Section 2.4.2 may solve the issue. Lastly, such a solution must be implemented separately for every data source.

Splitting Work of a Single Tuple

The fundamental issue with the above method is that a single tuple from the source can result in an arbitrary number of tuples higher in the pipeline. The point in execution when we can split the work for a single probe tuple is while iterating through the collision list in the hash table. What is needed is a way to split off a piece of a collision list and let another thread emit matches for this piece. To do this, we break collision lists into chunks, each containing many tuples. We leave the exact description of these chunks to the following section, as we tested multiple formats.

With collision lists separated into chunks, an idle thread can steal a single chunk of a collision list from the thread that initially started probing the list. The original thread must skip the stolen chunk to avoid emitting redundant matches. A given chunk consists of a set of build tuples, which a stealing thread will compare with the probing tuple. This thread must also have access to the probing tuple. Thus, in addition to stealing a collision list build chunk, the thread will copy a reference to the probe tuple. This necessitates an unfortunate aspect of the algorithm — the probe tuple must be materialized so that stealing threads can access it. Fortunately, only one probe tuple at a time needs to be materialized per thread and per join.

When stealing work, the probe tuple must be copied from a published sub-morsel into the stolen sub-morsel. This is because the initial thread may start probing with a new tuple while a helping thread is still emitting results for the previous probe tuple. Since the published probe tuple will be overwritten, the helping thread needs a copy of the previous probe tuple.

Listing 5.1: Type definition of a sub-morsel of work

```
1 struct SubMorsel {
2   void* buildChunk;
3   char* probeTuple = allocate(probeTupleSize);
4 }
```

Sub-Morsel Definition

So far, we have only loosely described a stealable sub-morsel; we now provide a definition in Listing 5.1.

The `probeTuple` field references a copy of the current probe tuple. The sub-morsel owns the associated memory and is reused for each probe tuple in turn. An alternative approach would be to use a shared reference. This would likely be a useful optimization but is left as future work.

The `buildChunk` field is a pointer to a subset of a collision chain from the build side. The value pointed to could be either a regular `TupleNode` or a `TupleArrayNode`; hence, it is defined as `void*`. We tested two ways of splitting collision lists into build chunks. Our first approach was using the `TupleArrayNode` as a build chunk, as defined in Listing 4.1. In this approach, subsequent build chunks are accessed via the `next` pointer. Thus, when a morsel is stolen, the published build chunk will be updated to the current buildChunk's `next` value.

This method works when using the lazy compaction method, as array nodes are always larger than some specified bound. Even so, this is inefficient, as the sub-morsel size is too small. We found 1024 to be a suitable array node length for efficient compaction. However, this is too small to compensate for the overhead of scheduling a sub-morsel. This issue is even more problematic when using eager compaction. In this method, array nodes start with a length of one and increase exponentially. This means that some build chunks will be as small as a single tuple; this is far too small to be worth scheduling as a sub-morsel.

An alternative is to add a second pointer to tuple nodes. Instead of referencing the subsequent node, this pointer skips several nodes and references the head of the next build chunk. This allows multiple array nodes to be used as a single build chunk. A definition can be seen in Listing 5.2, where `far` is the pointer to the first node in the subsequent build chunk. When using these multi-node build chunks, the build chunk in a published morsel is updated to the `far` pointer of the current build chunk rather than the `next` pointer.

Recall that regular tuple nodes are inserted in the global hash table in the same collision lists as these large array nodes. We want to avoid adding this `far` pointer to regular nodes as this would significantly increase node size while providing no benefit.

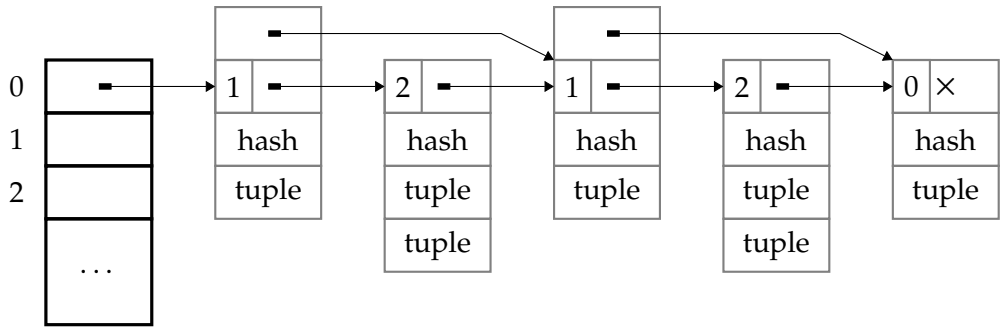


Figure 5.1: Structure of collision list with multiple node build chunks.

Thus, we only add the `far` pointer to array nodes. Array nodes can be differentiated from regular tuple nodes by checking if the length field in the top 16 bits of the `far` pointer is non-zero. The node does not contain a far pointer if this field is zero. Whether a node is a regular node or an array node, any incoming pointer will reference the node's `next` pointer. This is so that the `next` pointer is always at a known offset, allowing the length bits to be checked and the type of node discerned. An illustration of this structure can be seen in Figure 5.1. Here, we see three build chunks. The first two build chunks each consist of two array nodes. The last build chunk is a single regular node, with length bits set to zero.

Having described the structure of sub-morsels, we move on to an explanation of the stealing algorithm.

Listing 5.2: Type definition of a tuple array node

```
1 struct TupleArrayNode {
2   TupleArrayNode* far;
3   TupleArrayNode* next;
4   uint16 length;
5   uint64_t hash;
6   char tuples[TUPLE_SIZE][length];
7 }
```

Sub-Morsel Stealing

As described above, the sub-morsel is the unit of work that a busy thread will publish for other threads to steal. An initial implementation of the stealing algorithm works as follows. We have two arrays of sub-morsels, each with a length equal to the number of threads. The array `published` consists of published sub-morsels, and the array `stolen` consists of stolen sub-morsels. Assume we have two threads: thread 0 is busy

probing a long collision list, and thread 1 is idle. While thread 0 probes the collision list, it will store the subsequent sub-morsel of work at `published[0]`. Thread 1 wishes to help complete the join, so it scans the `published` array, looking for sub-morsels to steal. It finds the sub-morsel at `published[0]` and copies it to `stolen[1]` Then it updates `published[0]` to the next sub-morsel of work. Assuming more chunks are in the collision list, the next sub-morsel will consist of the same materialized probe tuple but the following chunk of build tuples. Updating the published sub-morsel is necessary to avoid emitting results more than once. A diagram of this example can be seen in Figure 5.2. In Figure 5.2a, thread 0 publishes a sub-morsel to `published[0]`, and in Figure 5.2b thread 1 steals the published sub-morsel.

Separate Sub-Morsels per Join

An important piece is missing from the above description — what if there are multiple joins in a given pipeline? We simply duplicate the `published` and `stolen` arrays for each join in the pipeline. Thus, we have a published sub-morsel and a stolen sub-morsel for every join and thread. Then, when the idle thread searches for a sub-morsel to steal, it iterates over the published sub-morsels for every join and every thread.

Is it necessary to have separate sub-morsels for each join? It is indeed necessary for published sub-morsels, as a given thread may need to publish sub-morsels simultaneously for multiple joins. Unlike published sub-morsels, each thread can only steal a single sub-morsel at a time. Threads only steal sub-morsels when idle. If a thread is processing a stolen sub-morsel, it is not idle and will not steal another sub-morsel. Nevertheless, separate sub-morsels are needed per join, as joins may have different probe tuple sizes.

Lastly, it should be asked whether a thread can have both stolen and published sub-morsels simultaneously. The answer is yes, though, for different joins. If a thread is processing a stolen sub-morsel, a join higher in the pipeline could be highly skewed. In such a situation, the thread will publish the sub-morsel for the higher join, allowing idle threads to help.

5.2 Scheduling

To understand sub-morsel scheduling, we take a step back and describe the scheduling of morsels in the Umbra database system. Each source operator, such as a base table scan, defines a `pickMorsel()` method, which allocates morsels. This method finds a morsel to run if one exists and returns a flag if no morsels remain. Rather than returning morsels, the method stores allocated morsels in thread local storage to be

accessed when the morsel is subsequently processed. Importantly, this method is thread-safe, as all threads call it in parallel.

Once a thread obtains such a morsel, it processes each tuple in the morsel against the code for the pipeline. Since Umbra is a code-generating system, the code for a pipeline is a compiled function that is called with the morsel as an argument. With this in mind, we see the morsel scheduling loop in Listing 5.3.

Listing 5.3: Morsel scheduling loop

```
1  while pickMorsel():
2    runPipelineMorsel()
```

We now update the scheduling algorithm to handle Sub-morsel Stealing in Listing 5.4. Each thread runs the standard morsel processing loop until all morsels have been assigned to threads. At this time, threads that have finished their morsels can help those that are still busy. An idle thread may not find a sub-morsel to steal immediately but will continue to search until all threads have completed their regular morsels. This method of waiting for sub-morsel is not ideal. Instead, the idle threads should sleep and only wake when sub-morsels become available, allowing their resources to be used by other queries. This optimization is left as future work.

Listing 5.4: Morsel and sub-morsel scheduling loops

```
1  while pickMorsel():
2    runPipelineMorsel()
3
4  while any thread still processing regular morsels:
5    if pickSubMorsel():
6      runPipelineSubMorsel()
```

5.2.1 Picking Sub-Morsels

The `pickSubMorsel()` method performs the stealing described in the previous section. Here, an idle thread iterates over the published sub-morsels of all joins and threads to find a stealable sub-morsel. This algorithm is shown in Listing 5.5. It should be noted that the published sub-morsels are modified by multiple threads, as both a stealing and publishing thread must update the published sub-morsel so that it always contains the next available work. Given this, care is required to guarantee that concurrent modifications of published sub-morsels are done safely. We defer an explanation of this aspect of the algorithm until Section 5.3.

Listing 5.5: Algorithm to steal a sub-morsel

```
SubMorsel published[numJoins][numThreads]
SubMorsel stolen[numJoins][numThreads]

def pickSubMorsel():
  for j in 1:numJoins:
    for t in 1:numThreads:
      if t != currentThread:
        SubMorsel& pub = published[j][t]
        SubMorsel& steal = stolen[j][currentThread]
        if pub.buildChunk:
          steal = (pub.buildChunk, copy(pub.probeTuple))
          pub.buildChunk = pub.buildChunk.far
          return True
  return False
```

5.2.2 Running Stolen Sub-Morsels

In Listing 5.4, the pipeline code run on stolen sub-morsels differs from that of regular morsels. This is because the code to run depends on the join in the pipeline for which the sub-morsel was stolen. The code for a given join consists of the logic to probe that join and all code above the join in the pipeline. We see the algorithm used to process a stolen sub-morsel in Listing 5.6. It also iterates over the join operators in the pipeline to find the stolen sub-morsel. This iteration could be optimized by storing the location of the recently stolen morsel. As the number of joins tends to be small, this optimization would likely have little effect. Thus, we leave it as future work. Once the stolen sub-morsel is located, the probing code specific to the join is run in `probeSubMorsel()`.

Listing 5.6: Algorithm to run a sub-morsel

```
def runPipelineSubMorsel():
  for j in 1:numJoins:
    if stolen[j][currentThread].buildChunk:
      joinOperators[j].probeSubMorsel(stolen[j][currentThread])
```

5.2.3 Probing Sub-Morsels

Unsurprisingly, sub-morsels require a different algorithm for probing from standard morsels. Additionally, Sub-morsel Stealing necessitates some updates to the standard probing algorithm. All code shown in this section is generated code within Umbra, though for simplicity, this is not shown here. We describe the existing standard probe

method in Listing 5.7. It involves obtaining the collision list for a given probe tuple from the hash table. It then traverses the collision list, emitting any matching build and probe tuple pairs. The `TupleNode` type is defined in Listing 4.1, and an example diagram is shown in Figure 4.1a. The `emit()` function shown here consists of the code for the rest of the pipeline.

Listing 5.7: Algorithm to probe regular morsels

```
1  def performProbe(probeTuple):
2    TupleNode* node = hashTable[hash(probeTuple.key) % tableSize]
3    while node:
4      buildTuple = node.tuple
5      if buildTuple.key == probeTuple.key:
6        emit(buildTuple, probeTuple)
7      node = node.next;
```

Changes to Regular Probe Method

As mentioned, the above algorithm needs to be updated for Sub-morsel Stealing. Recall that the updated collision list consists of chunks of tuple array nodes rather than single tuple nodes. Each chunk consists of one or more array nodes, with each array node containing a `length` field and a `next` pointer. If the length is zero, the node contains a single tuple and no `far` pointer. If the length field is non-zero, the node contains `length` tuples and has a `far` pointer.

We now show the updated standard probe algorithm in Listing 5.8. The probing thread first checks if the `length` field is non-zero, meaning that it is an array node and that the collision list is likely long. In this case, the thread publishes a sub-morsel consisting of the probe tuple and the second build chunk accessed through the node's `far` pointer. The second build chunk is published as the current thread is already processing the first chunk.

The thread then begins probing. This consists of three nested loops: the first over chunks, the second over nodes within a chunk, and the inner loop over tuples in a node array. The inner loop iterates over the node's tuple array until it reaches the limit as specified by the length field. The second loop traverses array nodes until it reaches the next chunk. This is determined by checking if the current node matches the `far` pointer of the first node in the chunk, as this points to the subsequent chunk. When a chunk is completed, the outer loop finds another chunk to process. The next chunk cannot be used directly, as it was published and may have been stolen by another thread. In this case, the other thread will have updated the `buildChunk` pointer in the published sub-morsel, which the current thread can now use as its next chunk. Upon claiming

another chunk, the published build chunk must be updated to the following build chunk.

It should be noted that the method as written is not thread-safe. In particular, the modifications to the published sub-morsel require care to maintain thread safety; this is covered in detail in Section 5.3.

Listing 5.8: Updated algorithm to probe regular morsels

```
1  def performProbe(probeTuple):
2    SubMorsel& pub = published[currentJoin][currentThread]
3    chunk = hashTable[hash(probeTuple.key) % tableSize]
4
5    if chunk.length:
6      pub = (chunk.far, copy(probeTuple))
7
8    while chunk:
9      for (node = chunk; node != chunk.far; node = node.next):
10       for buildTuple in node.tuples:
11         if buildTuple.key == probeTuple.key:
12           emit(buildTuple, probeTuple)
13     chunk = pub.buildChunk
14     pub.buildChunk = pub.buildChunk.far
```

Sub-Morsel Probe method

The method to probe a stolen sub-morsel is more straightforward than the updated regular probing method; it can be seen in Listing 5.9 It involves finding matches within a single chunk, accessed through the buildChunk field of the stolen sub-morsel. The two loops traverse the nodes within the chunk and the tuples with each node. This is identical to the two inner loops in Listing 5.8, with the only difference being that the stolen probe tuple is used to find matches and emit result pairs.

Unlike the previous algorithm, this method does not update the published sub-morsel. Updating the published sub-morsel occurs beforehand when the sub-morsel was stolen, as seen in Listing 5.5. This simplifies the `probeSubMorsel()` function, as by not accessing the published sub-morsel, it is thread-safe.

Listing 5.9: Algorithm to probe stolen sub-morsels

```
1  def probeSubMorsel():
2    SubMorsel& steal = stolen[currentJoin][currentThread]
3    chunk = steal.buildChunk
4    for (node = chunk; node != chunk.far; node = node.next):
5      for buildTuple in node.tuples:
6        if buildTuple.key == steal.probeTuple.key:
7          emit(buildTuple, probeTuple)
```

5.3 Concurrency

For clarity, the previous sections have avoided the issue of concurrency. As multiple threads modify published sub-morsels, reads and writes must be thread-safe; we address this issue here. In the preceding algorithms, three sections require changes for thread safety.

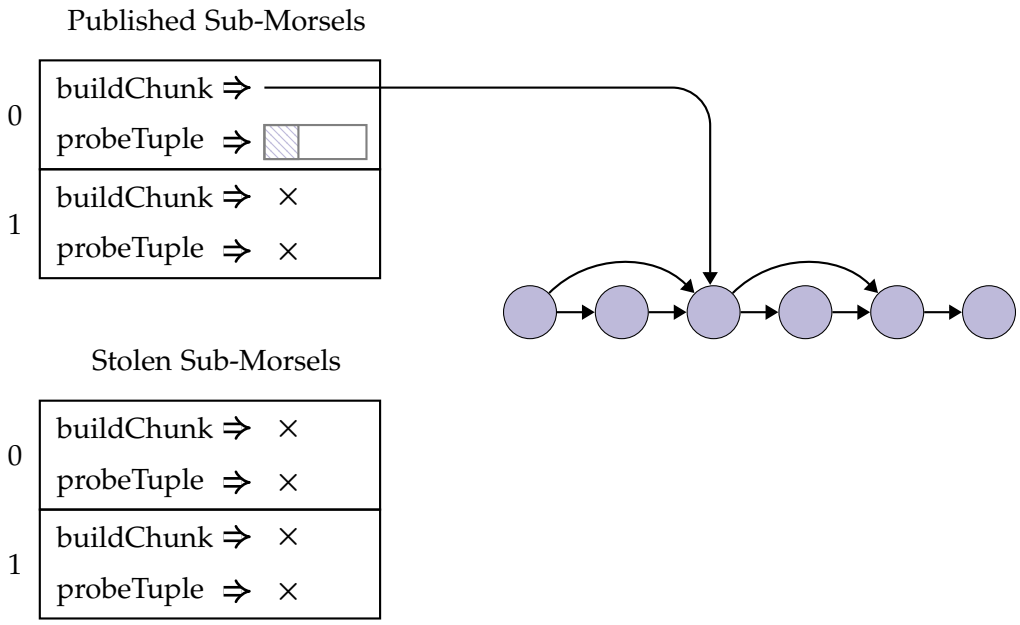
1. Publishing first build chunk in a collision list — line 6 of Listing 5.8.
2. Claiming and advancing published build chunk — lines 13-14 of Listing 5.8.
3. Stealing sub-morsel and advancing build chunk — lines 11-12 of Listing 5.5.

In the first case, both the `probeTuple` and the `buildChunk` pointer must be set by the publishing thread before any other thread can attempt to copy the `probeTuple` value or update the `buildChunk` pointer. To achieve this, each published sub-morsel has a shared mutex; while publishing a sub-morsel, the publishing thread holds an exclusive lock on the mutex. Though this could lead to contention, this is not a significant concern. While all threads are processing regular morsels, no threads will be attempting to steal morsels. Hence, each access to the mutex during this stage will be un-contended. Later, stealing threads will require read access to the shared mutex. This will cause some contention, but this is an acceptable tradeoff if there is skew.

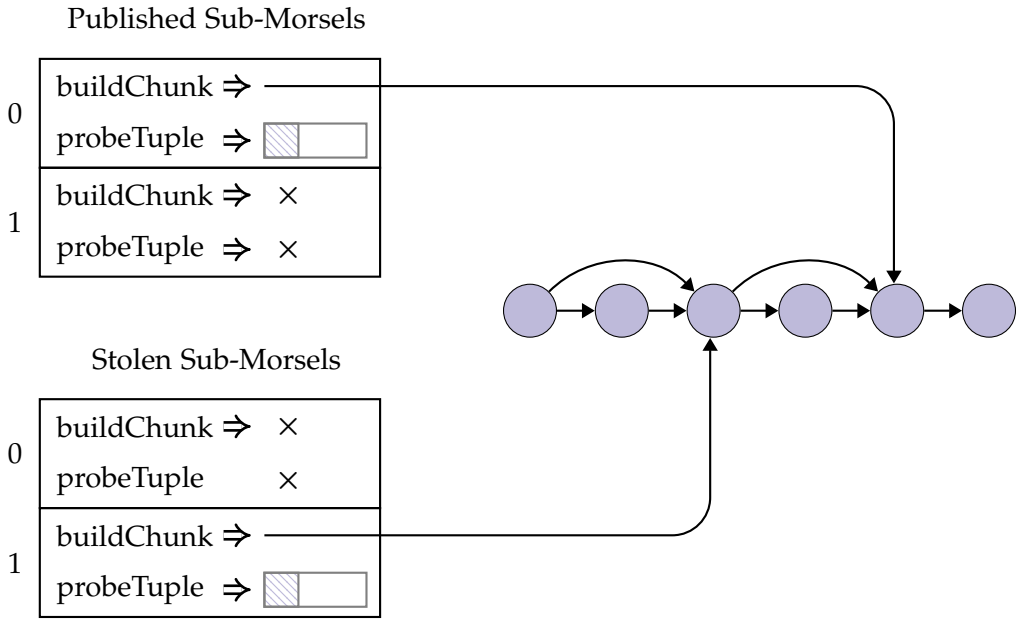
The second case occurs when the thread that published a sub-morsel reaches the end of a build chunk and needs to advance the published sub-morsel to the next build chunk. By accessing the sub-morsel's `buildChunk` pointer through an atomic reference, this can be achieved without locks. Using an atomic compare-and-swap function, the current `buildChunk` value is obtained, then `buildChunk` is updated to `buildChunk.far`.

The final case uses both the shared lock and an atomic reference. When a sub-morsel is stolen, it must be copied from the published area of the publishing thread to the stolen area of the stealing thread. This involves copying both the `probeTuple` value and the `buildChunk` pointer. Then the `buildChunk` pointer in the published sub-morsel must be updated, as in case 2. The published sub-morsel's mutex is acquired in shared mode to achieve this. This allows the `probeTuple` value to be copied by other threads while guaranteeing that the publishing thread does not update the value while it is being copied. A copy is necessary so that the stealing thread can access the `probeTuple` value if the publishing thread finishes the current collision list and moves on to another probe tuple. After acquiring the published sub-morsel's read lock, the `buildChunk` value is obtained and updated atomically to `buildChunk.far`. The stolen `buildChunk` value is then set to the obtained build chunk. Finally, the `probeTuple` value is copied from the published sub-morsel into the stolen sub-morsel, and the read lock is released. With these changes, the previous algorithms are thread-safe.

The following section discusses methods used to identify skewed input data. We can reduce the costs of joining non-skewed data by only using skew-optimizing algorithms on skewed data.



(a) When probing the first collision list node, thread 0 publishes a sub-morsel by copying the probe tuple into `published[0].probeTuple` and setting `published[0].buildChunk` to the next unprocessed build chunk.



(b) Thread 1 steals thread 0's published sub-morsel by copying `published[0].probeTuple` into `stolen[1].probeTuple`, assigning `published[0].buildChunk` into `stolen[1].buildChunk`, and updating published[0].buildChunk to the next unclaimed build chunk.

Figure 5.2: Illustration of sub-morsel stealing algorithm.

6 Selecting Join Algorithms

In Figure 2.1c, we saw that high skew can result in extremely long query execution times. It seems reasonable to always implement the techniques described in this thesis to avoid this scenario. However, most queries do not involve highly skewed joins and are thus not susceptible to these problems. Using the methods described in this thesis would result in worse performance for most queries.

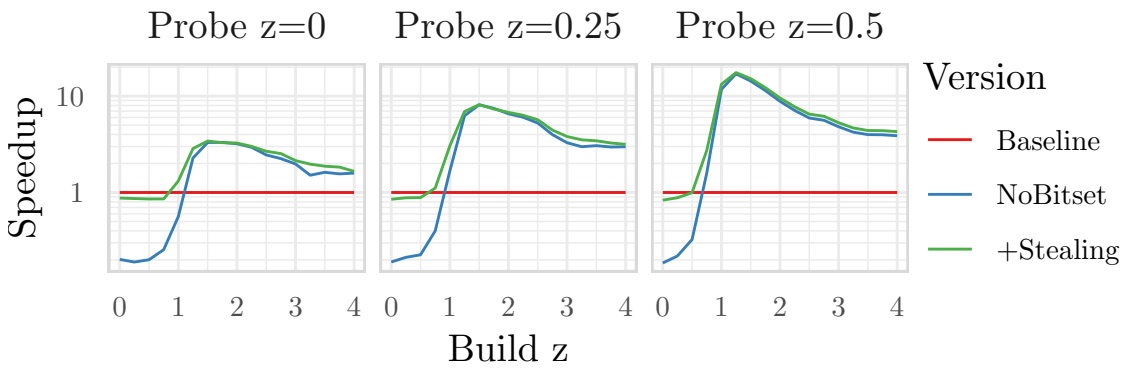


Figure 6.1: Comparison of speedup between *NoBitset*, which applies skew optimization techniques to all keys, and *+Stealing*, which only applies techniques to suspected skewed keys.

For example, in Figure 6.1, we see a comparison of three Umbra versions on the benchmark described in Section 7.2.1 The baseline version is Umbra without the techniques of this thesis, the *NoBitset* version is Umbra with compaction and Sub-morsel Stealing applied for every key, and *+Stealing* is Umbra with compaction and Sub-morsel Stealing applied selectively to skewed keys. The system used for this experiment is described in Section 7.1.2.

Notice that the best case speedup is similar for the two test versions. In this case, most data is skewed, so it helps to apply skew optimization techniques to every key. But if the data has low skew, such as when the build Z parameter is 0, the version that always optimizes for skew has a significant slowdown relative to the baseline. On some queries, the slowdown exceeds 5x.

The amount of slowdown shown above is unacceptable. Some slowdown is a

worthwhile tradeoff to avoid the worst execution times on skewed data. Nevertheless, slowing down typical workloads should be avoided. More pragmatically, given the importance placed on TPC-H benchmark times, any method that significantly reduces a database's TPC-H performance is unlikely to be adopted. It is thus incumbent on any database algorithm to avoid reducing the performance of common workloads as a side effect of improving performance on rare workloads.

This section discusses how a slowdown of typical workloads can be avoided. More specifically, we describe techniques to apply skew optimization techniques to skewed keys and avoid the overhead of these methods for low-skew keys.

6.1 Compile-time vs. Runtime

As Umbra is a code-generating database, there is a separation between the phase when the code for a given pipeline is generated and when that code is executed. Much of the information about the distribution of the joined relations is not available at code-generation time — it is only known during the execution of the query. For example, if the build relation is not a base table, its size is unknown at compile-time as the query that outputs the build relation has yet to be executed.

Unfortunately, the decision to use the standard join algorithm versus a skew-optimized join algorithm must be made when the pipeline code is generated. Generating code for both versions is possible, but applying this reasoning to other operators causes a rapid increase in code size. Thus, we must choose which version to generate at compile-time, using only the available information.

Nevertheless, using runtime information to limit additional work on keys that are not highly skewed is reasonable. With this in mind, the skewed join algorithm attempts to classify keys as skewed or non-skewed by counting their frequency at runtime. This allows skewed keys to use the more complicated skew logic, while non-skewed keys use the fast path.

The following section shows the sketch-based methods used to select the join algorithm, using information available at compile-time. Next, we look at the runtime approach based on key frequency.

6.2 Compile-time techniques

Statistics about the distribution of keys in the build and probe relations can help discern whether a join requires techniques to handle skew. For example, if the number of distinct keys is close to the cardinality of the relation, then the relation is not skewed, and skew handling techniques will not improve performance.

On the other hand, if the cardinality is significantly larger than the number of distinct keys, there are many duplicate keys. This could mean many keys are duplicated a small number of times or few keys are duplicated many times. In this case, determining whether skew handling techniques will be helpful requires deeper analysis.

Clearly, obtaining key distribution statistics can play a part in selecting the proper join algorithm. In this section, we look at techniques available at compile-time for estimating such statistics.

6.2.1 Distinct Count

Determining the exact number of distinct keys in a relation requires processing every key and recording the number of distinct values. This requires $O(n)$ space and is thus too expensive. The HyperLogLog algorithm provides a means to estimate the number of distinct keys in constant space [12]. Umbra uses HyperLogLog sketches to estimate distinct key counts for base relations [22]. This works for base relations, but relations higher in the tree do not have tuples available for sketching before runtime. Thus, Umbra uses various heuristic techniques for non-base relations to estimate the distinct key count.

6.2.2 Self Join Size

Alon et al. developed a technique similar to HyperLogLog for estimating a statistic known as the second frequency moment [1]. This statistic is equal to the sum of the squared frequencies of the distinct values of a distribution.

This can be described more succinctly as self-join size. To understand this statistic, it is helpful to consider some extreme distributions. Consider a relation where every key is unique. If the relation is joined with itself, every key will only find a join partner with itself. Thus, the output will be the same size as the input relation.

On the other hand, consider a relation with only one unique key value. In this case, every tuple will be joined with every other tuple — this is equivalent to a cross-join. Thus, the output cardinality will be the square of the input cardinality.

This statistic provides a useful notion of the skewness of a relation. If the self-join size is equal to the input cardinality, the relation is not skewed. If the self-join size is equal to the square of the input cardinality, the relation is very skewed.

Umbra provides these sketches for base relation columns. As with distinct count, Umbra combines these estimates using heuristics when higher in the operator tree.

6.2.3 Effectiveness of Compile-time Sketches

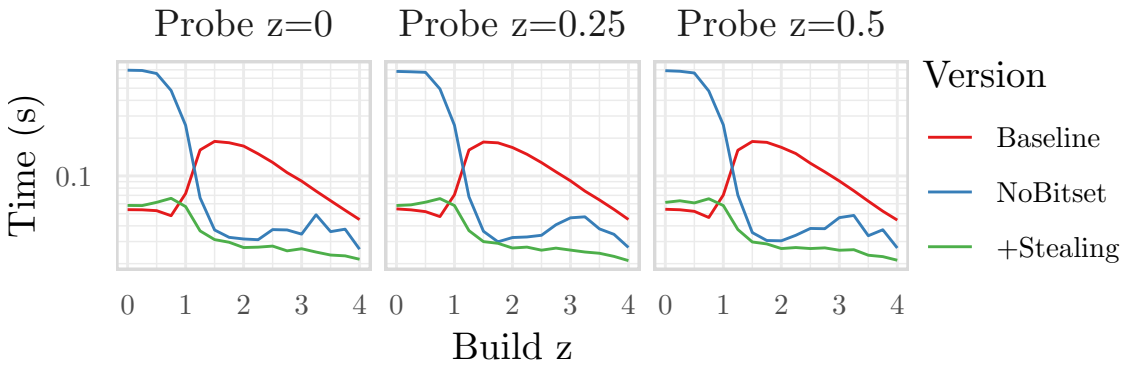
Despite the potential for sketches to estimate the skewness before code generation, we have yet to find a way to use them effectively. One way to use these sketches is to generate the skew-optimized join if statistics derived from sketches exceed certain bounds and otherwise to generate the standard join. We have tried to find suitable bounds on the Zipf micro-benchmarks using this technique. Each tested bound had too many false positives or negatives to be helpful. Perhaps more effective values could be found with a more systematic search of bound values. We leave this as future work. These compile-time techniques are not used in any tests in the evaluation section.

6.3 Probabilistic Counting

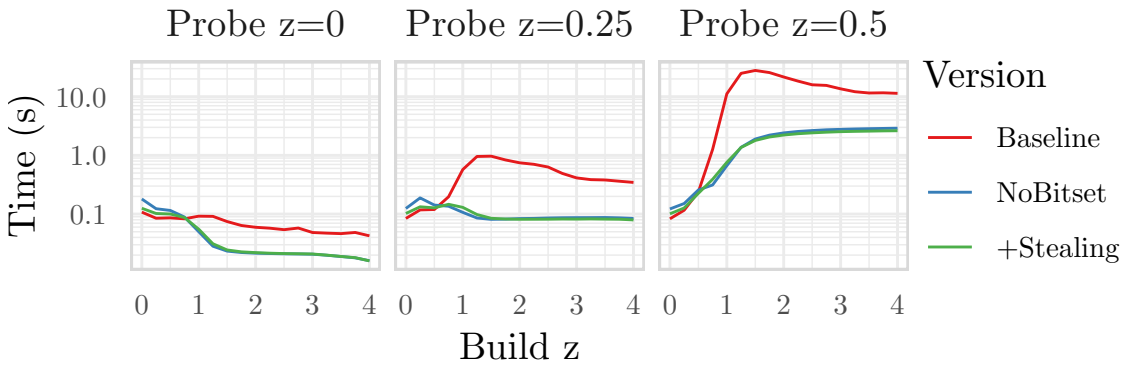
As described in Chapter 4, one of the techniques used in this thesis is compacting tuples with common keys into dense array nodes rather than linked lists. This results in significant speedups when there are many duplicates. Unfortunately, we see a slowdown when few duplicates exist, such as in Figure 6.1 when the build Z parameter is 0. The reason for this slowdown can be seen in Figure 6.2, where we show the execution times of the build and probe stages for the same benchmarks. The materialization stage is omitted as the time spent is negligible. We see that, though the *NoBitset* version has probe times comparable to the *+Stealing* version, the build times on low skew queries are far worse. For these queries, the additional time spent building array nodes does not significantly improve performance during the probe stage. The original collision lists are relatively short and can be traversed quickly. Thus, there is little payoff in compacting these lists, and the additional time was wasted. This is especially true for the *NoBitset* version, which spends a great deal of time during the build stage.

As a side note, the fact that the *NoBitset* build stage is slower than *+Stealing* build stage at high skew is attributable to a difference in the implementations. *+Stealing* adds local skew table chains to the global hash table synchronously, as there are expected to be few such chains. To compare low skew build stages fairly, the *NoBitset* version does this insertion in parallel, as many values are expected. This causes a slowdown in *NoBitset* at high skew when there are few such chains.

To avoid this situation, we wish to only compact tuples if we have seen many duplicate keys. The obvious way to achieve this is to build a hash table from hash values to their frequency, incrementing the frequency every time we see a given hash value. While the observed frequency of a given hash value is below some bound, we insert the tuple in the global hash table as usual. If the frequency exceeds the bound, we process tuples with this hash value thread-locally. This entails adding the tuples to a local hash table and copying tuple nodes into dense tuple array nodes.



(a) Build Stage



(b) Probe Stage

Figure 6.2: Comparison of stage execution time between *NoBitset*, which applies skew optimization techniques to all keys, and *+Stealing*, which only applies techniques to suspected skewed keys.

This technique is generally what we do, with the caveat that we do not count the frequency of the hash values as described. Storing a count for every hash value would require creating an array of counters parallel to the main hash table. We did not implement this, but we believe the space required would be too costly. Nevertheless, it would be worth testing this method with 8-bit counters; we leave this as future work.

Instead, we count the frequencies probabilistically, storing a single bit per hash directory slot. Rather than counting until we have reached some bound n, we set the bit with a probability of $\frac{1}{n}$ on every insert. This is similar to counting until a bound is reached, though with some added noise; some frequent keys will not be optimized for skew, and some infrequent keys will be optimized for skew. Nevertheless, this technique has been found to work well. This is likely because the small size of the bitset allows it to be stored in the cache. Thus, accesses are fast.

This technique can be seen as a simpler version of the probabilistic counting described by [20]. Four bits could be used per slot to improve the counting accuracy using this technique. We considered testing this method but felt that the increase in space would likely not be worth the additional accuracy.

The *+Stealing* version used in the previous benchmarks uses the probabilistic bitset as described above. As seen in Figure 6.1, using this bitset to selectively enable skew optimizing techniques on a per hash value basis limits the downsides of the techniques proposed in this thesis. The following section will evaluate the proposed skew optimization techniques on several benchmarks. Both test versions shown in the subsequent section use the bitset described in this section.

7 Evaluation

Database systems are frequently evaluated on the TPC-H benchmark. Unfortunately, these benchmarks are not suitable for evaluating the techniques described in this thesis. More specifically, TPC-H datasets are generated so that keys are uniform and exhibit minimal skew. This being the case, we found alternative benchmarks that properly exercise the skew handling join. Nevertheless, we provide TPC-H benchmark output to show that, while these techniques do not improve performance, nor do they reduce performance on the TPC-H workloads. We start this chapter by describing the systems used to run the benchmarks and then proceed with the benchmark results. Following this, we provide the results of each benchmark.

7.1 System Setup

Two systems were used to run these benchmarks. Due to its size, the Cardinality Estimation benchmark was run on a system with more resources. All other benchmarks were run on the second, smaller system.

7.1.1 System for Cardinality Estimation Benchmark

We run the Cardinality Estimation benchmark on a NUMA system with 4 sockets, each with an Intel E7-4870v2 CPU with 15 cores and 2 hyper-threads per core, running at 2.30GHz. The system has 1TB of RAM.

Each query was run three times, and the minimum of the three runs was reported.

7.1.2 System for other Benchmarks

All other benchmarks were run on a system with an Intel i9-7900x CPU with 10 cores, each with 2 hyper-threads, running at 3.30GHz. The system has 130GB of RAM. Each query is run five times for all such benchmarks, and the minimum execution time is used in the analysis.

7.2 Benchmarks

The benchmarks can be split into three groups. First, we have micro-benchmarks with Zipf distributed relations. These provide a worst-case skew scenario and show the benefits of our techniques. Next, we have TPC-H and the related JCC benchmarks. Though the techniques in this thesis make little improvement over the baseline on these benchmarks, we show that our technique does not reduce performance. Lastly, we show results on the Cardinality Estimation benchmark. We show that our system notably improves query time on these workloads.

For each benchmark, we compare three code versions. The *Baseline* version is Umbra without the skew handling features implemented in this thesis. The *Compact* version includes the feature described in Chapter 4, which compacts collision lists into dense arrays. The *+Stealing* version includes the features from both chapter 4 and Chapter 5. Thus, in addition to compacting tuples, this version allows sub-morsels to be stolen, improving the load balance between threads.

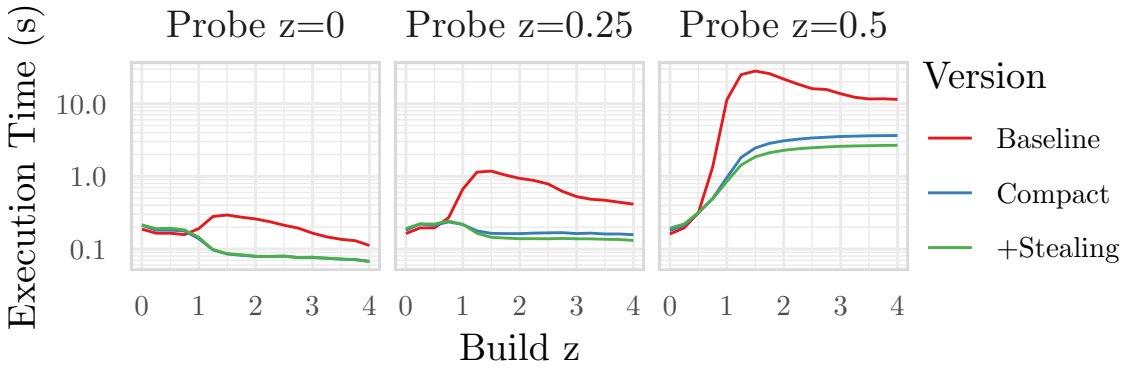
7.2.1 Zipfian Micro-benchmarks

To test the baseline, we evaluate our test code using the same datasets used in Section 2.4. These consist of build and probe relations, each with 1e7 tuples. The tuples consist of two 64-bit integers, a key, and a foreign key. The foreign key is distributed with a Zipf distribution with varied Z parameters. For the build relation, we vary Z between 0 and 4; Z of 0 is a uniform distribution. For the probe relation, we vary Z between 0 and 0.5; larger probe Z values result in huge output sizes and execution times.

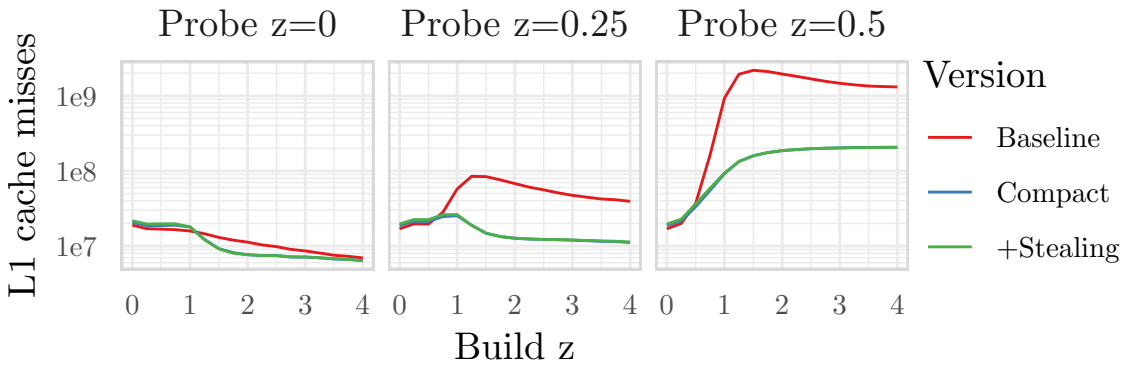
Each probe relation has two versions, one shuffled and one approximately clustered by the foreign key. The clustering is done by loading the base relation from a sorted CSV file. This does not guarantee that the base relation is ordered identically but results in most duplicate foreign keys being clustered together. The clustered version is used to exercise the situation when there is a significant work imbalance between probe morsels. Following are the results.

Probe Shuffled

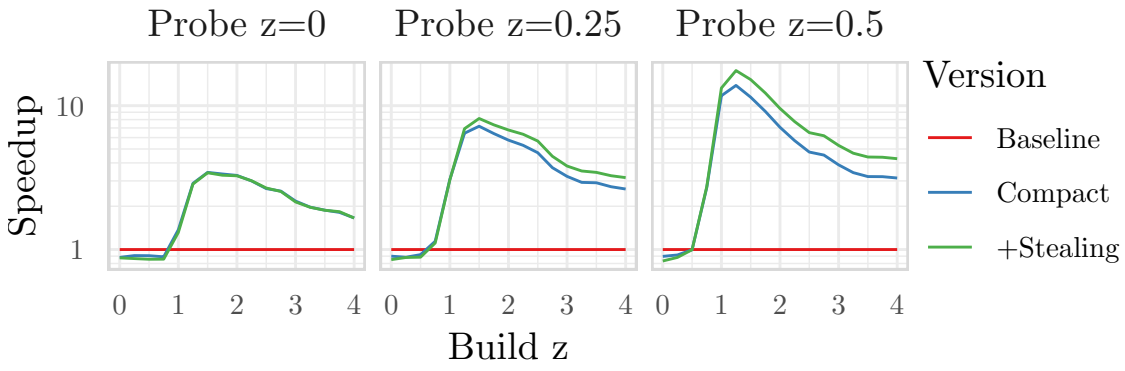
In Figure 7.1a, we compare the execution time between the two test versions and the baseline. Below, we see the associated cache misses in Figure 7.1b. Notice that, as in Figure 2.1b, there is a strong resemblance between the cache misses and the relative execution time. This holds true for each of the three code versions. However, unlike Figure 2.1b, the test versions no longer strongly resemble the output size in Figure 2.1a. The two test versions reduced the execution time significantly by un-tethering the



(a) Execution Time



(b) L1 cache misses



(c) Speedup of test versions relative to baseline

Figure 7.1: Comparison of joins on Zipf distributed build and probe relations for three code versions. Probe relations have shuffled order.

Version	Max Speedup	Mean Speedup	Max Slowdown
Compact	13.8	3.67	1.13
+Stealing	17.5	4.35	1.20

Table 7.1: Summary of results for Zipf tests with shuffled probe relation

cache-miss rate from the output size. The tests have some workloads with low Z values that result in a slowdown. Note that the test versions here do not use the sketch-based filters described in Chapter 6.

Finally, we plot the speedup of the test versions relative to the baseline. Both make significant improvements over the baseline, with *+Stealing* performing better than *Compact* overall. We see a summary of results in Table 7.1. The following section shows similar tests, this time with clustered probe relations.

Probe Clustered

We now see similar results on workloads with a clustered probe side. The workloads result in a high imbalance of work between threads and are more suited for the *+Stealing* test version. In Figure 7.2a, we compare the execution time between the three code versions. Note that unlike in Figure 7.1a, *Compact* shows an increase in execution times when the probe size Z parameter is greater than 0. In this case, *+Stealing* maintains a low execution time at a probe Z of 0.25 and shows a much smaller increase than *Compact* at a probe Z of 0.5

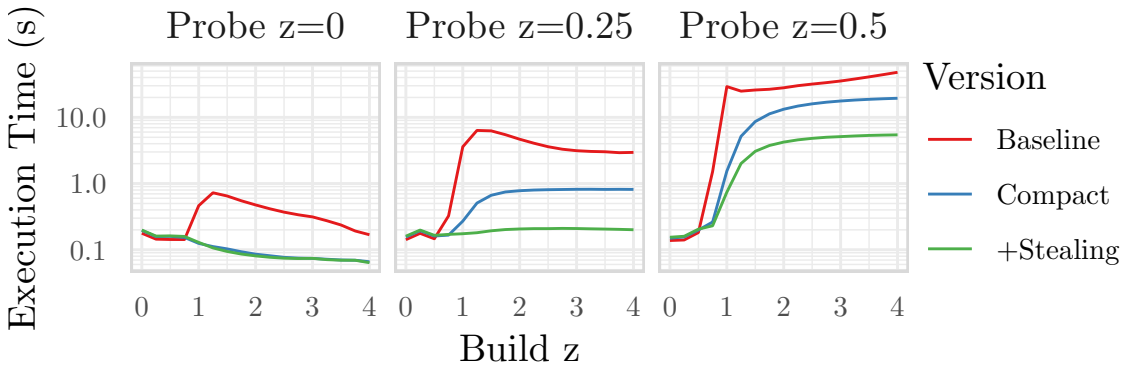
We now look at the thread utilization in Figure 7.2b. Both *Baseline* and *Compact* significantly drop in thread utilization at high probe Z values. Though *+Stealing* stays high throughout, this should be taken with a large grain of salt. In *+Stealing* threads spin when they do not have work; this essentially guarantees near 100% thread utilization. Ideally, threads should instead wait on a condition variable for work to become available or for the query to end. Importantly, this would allow concurrently running queries to use the hardware threads. We leave this as future work.

In Table 7.2, we see a summary of the results on the clustered workload. Given the workload, *+Stealing* performs better with a larger maximum and average speedup.

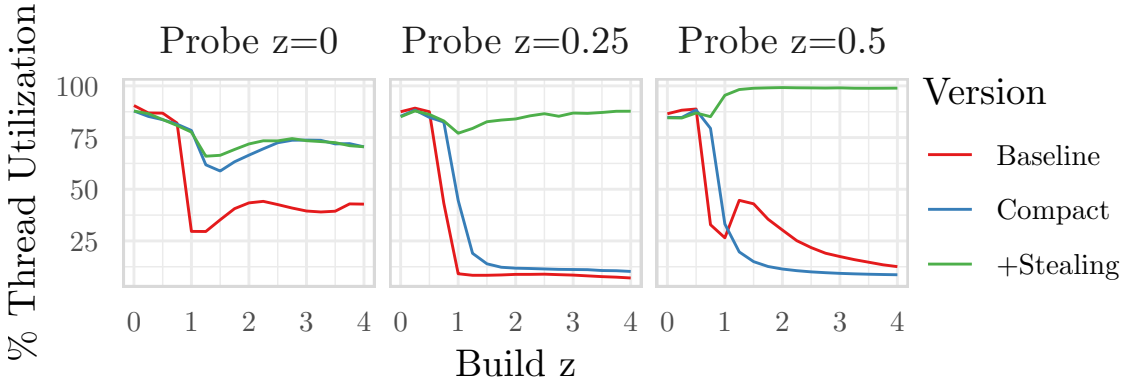
In the following section, we show results on TPC-H and related benchmarks.

7.2.2 TPC-H & JCC Benchmarks

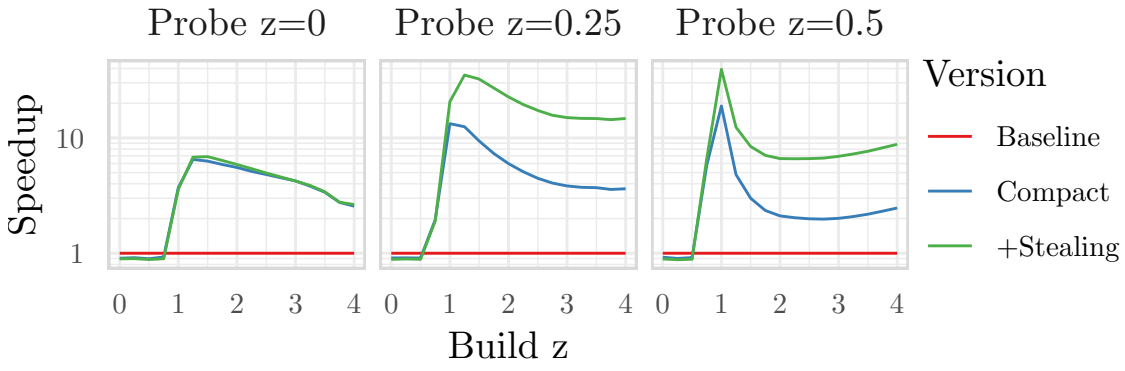
In this section, we show results on the TPC-H benchmark. Unlike the micro-benchmarks above, TPC-H queries are varied in form to mimic realistic queries used in a business



(a) Execution time



(b) Thread Utilization. *+Stealing* values should be interpreted cautiously as the scheduling loop spins, causing a nearly 100% thread utilization measurement.



(c) Speedup relative to baseline

Figure 7.2: Comparison of joins on Zipf distributed build and probe relations for three code versions. Probe relations have clustered order.

Version	Max Speedup	Mean Speedup	Max Slowdown
Compact	18.9	4.02	1.12
+Stealing	39.3	9.32	1.14

Table 7.2: Summary of results for Zipf tests with clustered probe relation.

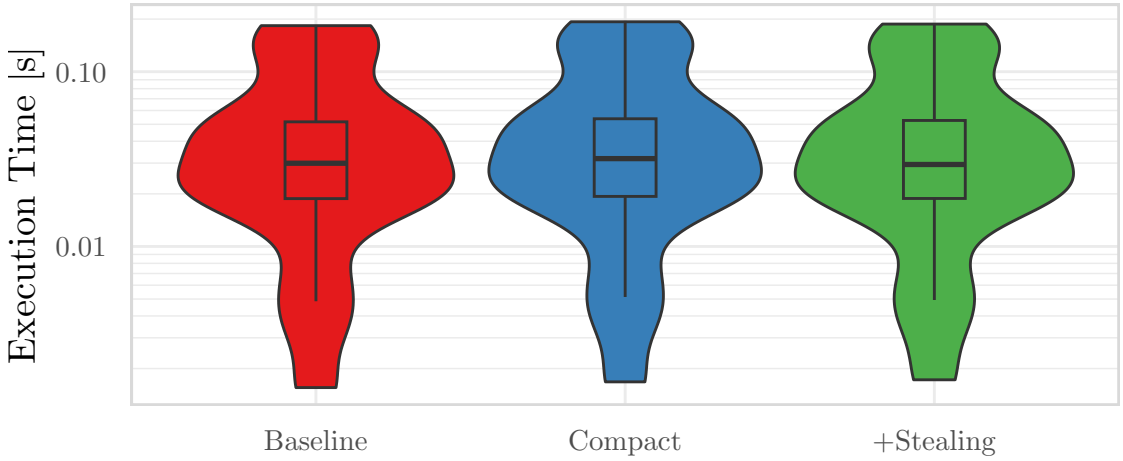


Figure 7.3: Comparison of execution times on TPC-H scale factor 10

setting. The TPC-H dataset is generated with a uniform process. Thus, the results are not very skewed. Hence, our code does not show execution time improvements; we include these results to show that they do not cause a significant slowdown.

In addition to the standard TPC-H benchmark, we include results on a related skewed benchmark. Specifically, JCC is a benchmark that shares the schema of TPC-H but has a skewed generation process. In addition to allowing skew with a single column, JCC incorporates skew across correlated columns, making it more like real-world datasets.

We start with the TPC-H results and then show the JCC results.

TPC-H Results

In Figure 7.3, we show the distribution of the TPC-H queries on the three code versions — *Baseline*, *+Stealing*, and *Compact*. The distributions are quite similar, with the *+Stealing* version's mean speedup being very close to 1. The summarized results are in Table 7.3.

Version	Max Speedup	Mean Speedup	Max Slowdown
Compact	1.22	0.95	1.15
+Stealing	1.22	1.01	1.11

Table 7.3: Summary of results for TPC-H benchmark at scale factor 10

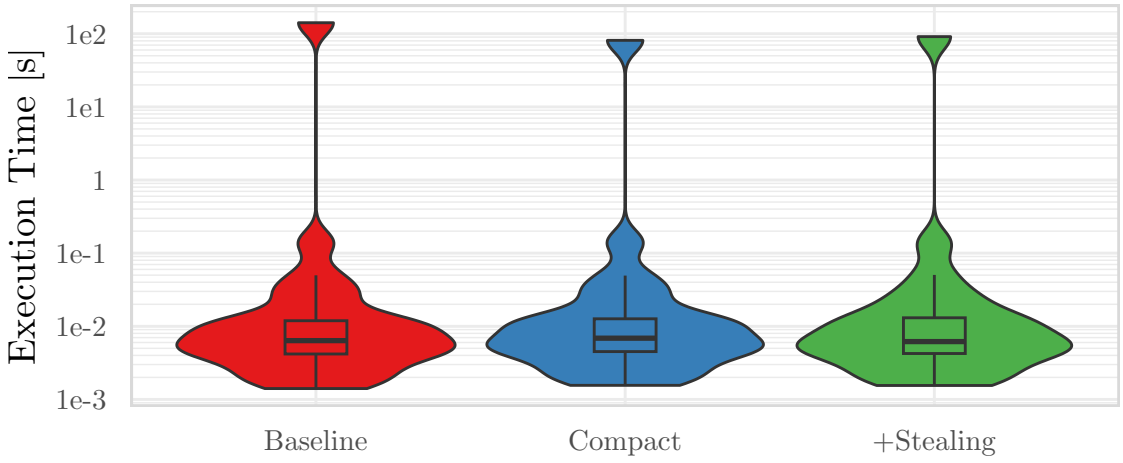


Figure 7.4: Comparison of execution times on JCC scale factor 1

Version	Max Speedup	Mean Speedup	Max Slowdown
Compact	1.74	0.98	1.21
+Stealing	1.55	1.02	1.19

Table 7.4: Summary of results for JCC benchmark at scale factor 1

JCC

In Figure 7.4, we show the distribution of the JCC query execution times on the three code versions. The summarized results are in Table 7.4.

The distributions are again reasonably similar, with the test versions having minimal speedup over the baseline. However, the test versions have a decent speedup on the longest-running query. Unlike the other benchmarks, *Compact* outperforms +Stealing in this case. Surprisingly, the methods proposed in this thesis do not perform better on a skewed dataset like JCC. This warrants additional investigation. We now move on to a dataset where the techniques of this thesis help greatly.

7.2.3 Cardinality Estimation Benchmark

In a recent paper on cardinality estimation, Chen et al. provided a large data set and associated queries for testing cardinality estimation techniques [7]. The dataset relies on six real-world datasets and adds associated queries. Due to its purpose for cardinality estimation, the structure of the join graphs is complex and highly varied. A notable feature is that the queries are classified as cyclic or acyclic. We will see that whether a query is cyclic or acyclic has a significant effect on whether the methods proposed in this thesis improve a given query's execution time. Additionally, many of the datasets have a graph structure; for example, the Epinions dataset consists of a node for each user on the Epinions website and a directed edge between users and other users that they trust. As discussed in Section 2.5.4, join queries over graph datasets are susceptible to skew issues. This benchmark provides evidence that the techniques in this thesis help mitigate the effects of skew in joins over graph-structured data.

Limiting Execution Time

The benchmark includes 3608 queries. We run each query 3 times per benchmark run to obtain accurate timing. However, some queries exceed 10 minutes in execution time; thus, running all queries is not feasible. Instead, we ran each query with a timeout of 90 seconds for all three repetitions. This results in a situation where the baseline may time out on a query for which the test version did not, and vice versa. To rectify this, we re-run the timed-out queries if another code version did not time out. Thus, the baseline and test code versions have timing data for any query on which at least one did not time out. This amounted to 3299 queries in total.

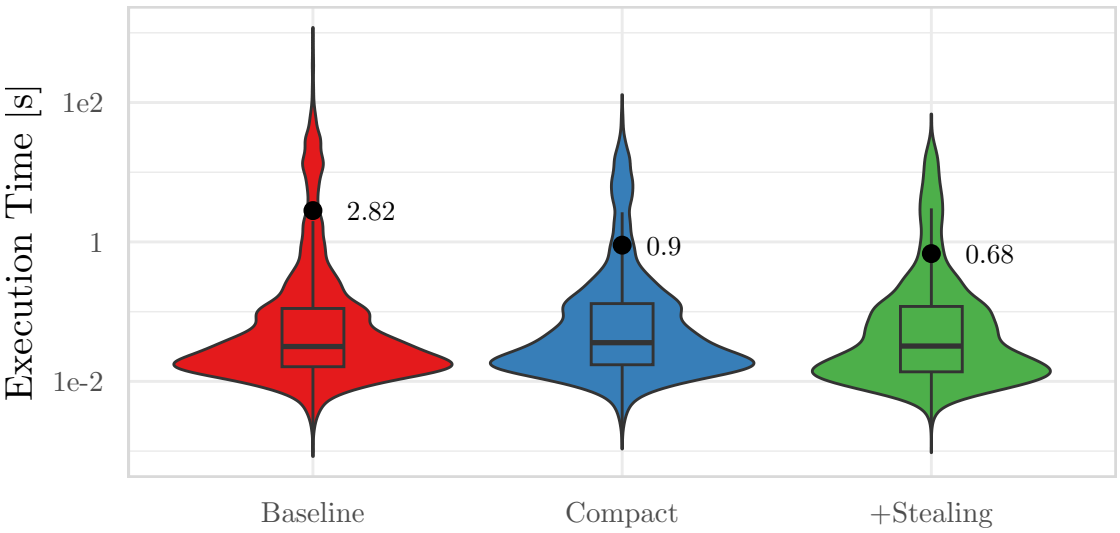


Figure 7.5: Comparison of execution times on Cardinality Estimation benchmark

Results

In Figure 7.5, we show the execution times for the baseline and well as the two tested code versions. Most of the queries are short, but a small number take much longer. Note that the y-axis has a logarithmic scale, with the longest query time taking over 1000 seconds. The mean execution time is annotated in the plots, making the level of skew more evident. For all three versions, the mean time is well above the median, meaning execution times are highly skewed. Though most query times are similar between versions, some very large *Baseline* execution times result in the *Baseline* mean execution time being around three times that of the test versions.

We observe considerable differences in mean execution time if we plot cyclic and acyclic queries separately as in Figure 7.6. For all three versions, acyclic queries take longer than cyclic queries. All versions behave similarly for cyclic queries, and *Baseline* mean execution time is the smallest of the three, though by a narrow margin. Whereas, *Baseline* has a much larger mean execution time for acyclic queries. It is clear that both test versions drastically reduce the mean execution time while making little change to the median execution time.

We now compare the two test versions. In Figure 7.7, we see the speedup of each test code version against the baseline, again separated by the workload type. The *+Stealing* version performs well on cyclic and acyclic queries. It is slightly better than the baseline on cyclic queries, whereas on acyclic queries, the mean speedup is 71%

Seeing the speedup for queries at a more granular level is helpful. In Figure 7.8,

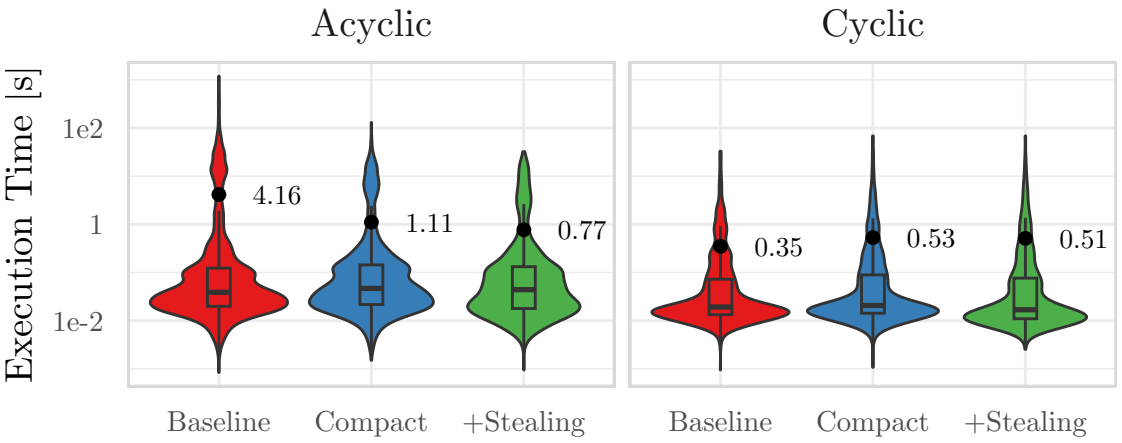


Figure 7.6: Comparison of execution times on Cardinality Estimation benchmark, separated by cyclic and acyclic queries.

we see the speedup over the baseline plotted against the baseline execution time. Unsurprisingly, we see that in both test versions, the largest speedup values are on queries with long baseline execution times. Most queries have a speedup of around 1 — they are relatively unchanged between baseline and tests. A small fraction of queries had very long baseline times, for which the test versions provided a large speedup.

We now focus on a subset of the CE benchmark. Figure 7.9 is similar to the previous plot but is restricted to Epinions queries. The *+Stealing* version performs particularly well on this data. Most notably, a group of queries with baseline times over 10 seconds can be seen to have a larger speedup on *+Stealing* than on the *Compact* version. Every one of these queries uses fewer than 3 of the 120 available threads in the baseline version. The additional speedup of *+Stealing* over *Compact* is clearly due to the scheduling of sub-morsels. Though the two test versions have similar results, it is clear that the combination of compaction and Sub-morsel Stealing is more effective than compaction on its own.

Finally, we end this section by summarizing the speedup of the two test versions in Table 7.5. We see that *Compact* has a mean speedup over *Baseline* and that *+Stealing* is a significant improvement over *Compact*. This is particularly evident in the maximum and mean speedups. Given the variety of queries included in the Cardinality Estimation benchmark, this sizable speedup shows that the techniques of the thesis are worthwhile optimizations for a database system.

In the next section, we discuss the results at a high level and describe some areas for future improvement.

Version	Max Speedup	Mean Speedup	Max Slowdown
Compact	39.4	1.13	3.64
+Stealing	87.8	1.49	4.39

Table 7.5: Summary of results for Cardinality Estimation benchmark

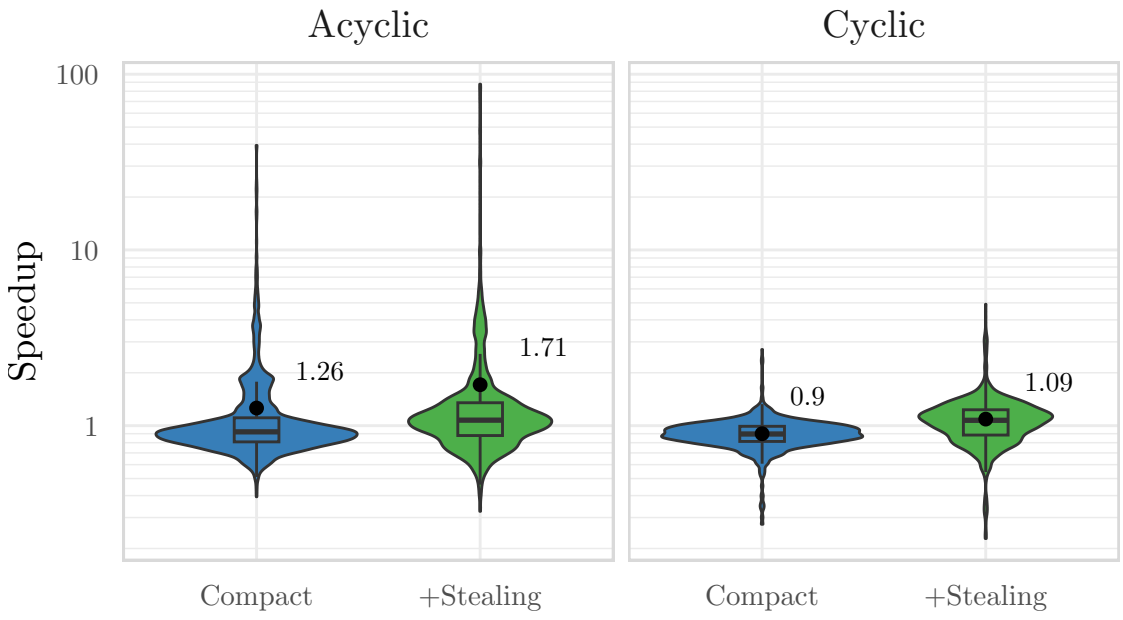


Figure 7.7: Speedup over baseline on Cardinality Estimation benchmark, separated by cyclic and acyclic queries.

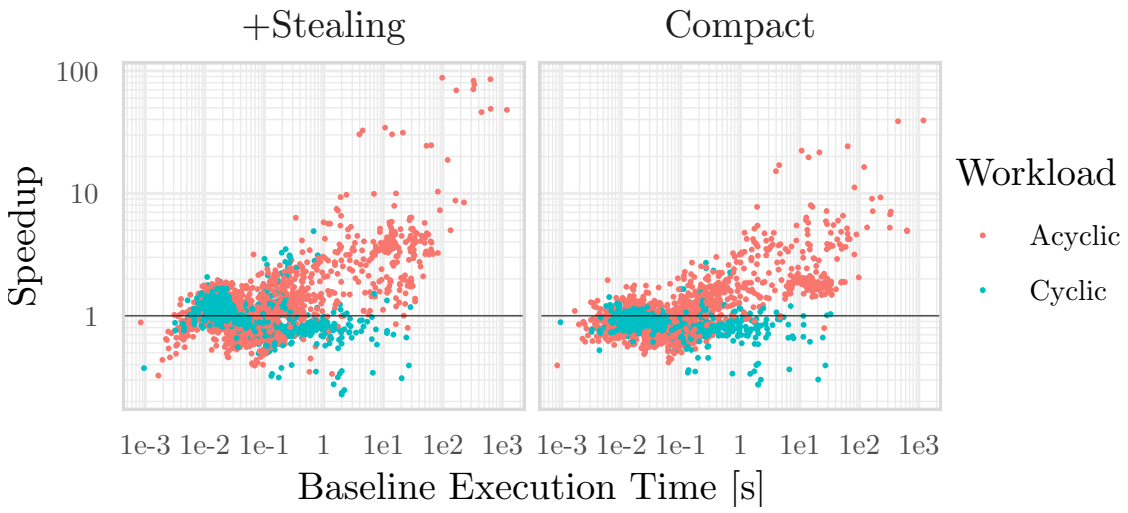


Figure 7.8: Speedup versus Baseline execution time for Cardinality Estimation benchmark.

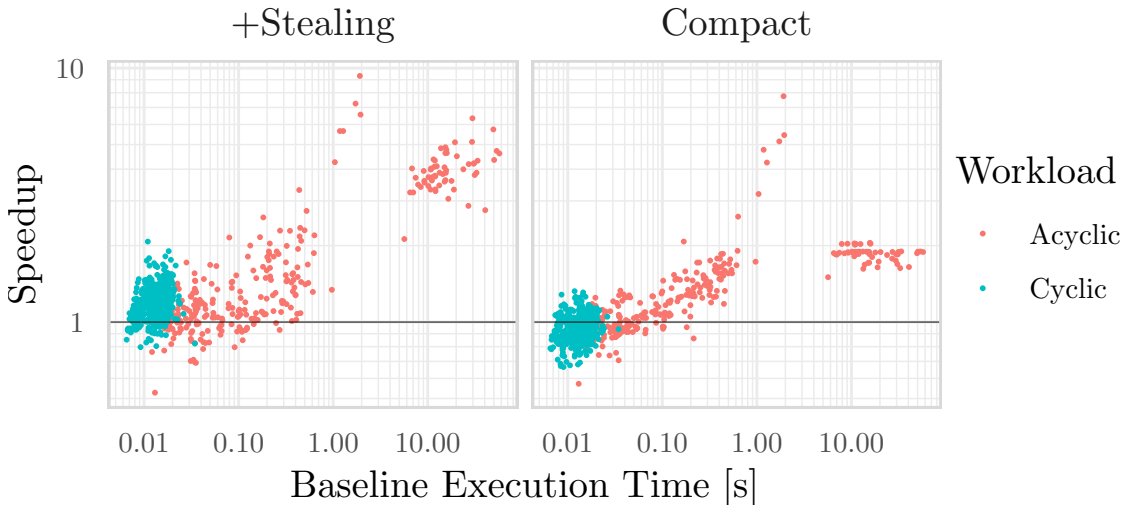


Figure 7.9: Speedup versus Baseline execution time for Epinions queries on Cardinality Estimation benchmark.

8 Discussion

In the previous section, we tested the two techniques proposed in this thesis on various benchmarks. We showed that these methods provide significant performance improvements on skewed workloads. We now discuss these results in the broader context and evaluate the applicability of these techniques. Additionally, we discuss areas for future improvements.

Node Compaction

In the previous section, we tested array Node Compaction with and without Sub-morsel Stealing. Though for most benchmarks, the version with Sub-morsel Stealing was faster, it is clear that Node Compaction has significant performance improvements on its own. This is notable, as array Node Compaction is much simpler to implement than Sub-morsel Stealing. Node Compaction simply involves the addition of a new hash join operator; Sub-morsel Stealing requires modification of the morsel scheduling algorithm. For this reason, Node Compaction is likely more applicable to other in-memory database systems.

Sub-Morsel Stealing

Benchmark results show that combining the techniques of compaction and Sub-morsel Stealing is more effective than using compaction alone. Though this fact is not surprising, we were surprised by the amount of improvement caused by Sub-morsel Stealing. Previously, we believed that Sub-morsel Stealing would provide value in relatively rare circumstances. For example, when the probe side has skewed keys clustered in the base relation. This situation seems unlikely to occur frequently in real-world databases. Nevertheless, since such situations result in very long execution times, we still felt it worthwhile to implement Sub-morsel Stealing. The results of the Cardinality Estimation benchmark suggest that this situation is far more likely to occur than we had previously thought. While the queries on this dataset are synthetic, they are diverse in structure, and the underlying data comes from real-world datasets.

This suggests that improvements to the Sub-morsel Stealing algorithm could improve overall performance. There are several areas where this technique can be improved.

Stealing a sub-morsel currently involves copying the published probe tuple. Though this performs well on the provided benchmarks, it could be problematic on larger tuples. Additionally, since a read-lock must be held while copying, this could increase contention. A potential improvement is to replace the copied tuple with a reference counting pointer. This would allow for cheap copying of the published probe tuple while keeping space usage minimal.

A more important improvement involves the sub-morsel scheduling loop. The current implementation continuously spins, looking for sub-morsels until all regular morsels are exhausted. This potentially wastes resources as other threads cannot use the spinning thread, whether or not it finds sub-morsels. Rather than spinning, idle threads should use condition variables to sleep until a sub-morsel is available or all regular morsels are completed. This is a necessary improvement before Sub-morsel Stealing can be implemented in a production system.

Finally, we discuss a more high-level improvement to the Sub-morsel Stealing algorithm. Section 5.1.1 described an alternative means of defining sub-morsels. In this technique, a sub-morsel would consist of a sub-range of a regular morsel. We chose not to implement this method as we did not believe it would handle situations with very high skew. Nevertheless, this technique would likely perform better on data with relatively low skew than the Sub-morsel Stealing algorithm implemented in this thesis. Given the significant performance benefit of the Sub-morsel Stealing algorithm, we believe this other technique deserves further investigation.

Handling Non-Skewed Data

A primary problem in this thesis was finding how to improve join performance on skewed data while not reducing performance on uniform data. Unfortunately, we did not completely achieve this goal. On the TPC-H and Zipf benchmarks with low Z values, we saw some queries where *+Stealing* was slower than the baseline. Given the significant improvements on skewed data, we believe a small slowdown on some queries is an acceptable tradeoff.

We found that any extra work to optimize a join for skewed data will slow down the algorithm on uniform data. Thus, a primary part of our method involves limiting the amount of additional work needed on uniform data. For example, consider when a previously unseen key is inserted in the hash table. The baseline algorithm will insert the tuple directly in the global hash table. Our method will check a single bit in the skewed bit set, then insert in the global hash table, then potentially set the bit in the bit set.

There are other ways to improve join performance on uniform data. As described in Chapter 6, we attempted to use sketches to estimate the skewness of relations before

code generation. These sketches estimate various statistics about relations. We selected bounds for the estimated statistics and only enabled the skew optimization techniques if the estimates exceeded these bounds. Unfortunately, we found no bound values with a high enough accuracy to be useful. We did not systematically search for bound values but selected the values heuristically. A more systematic approach might find more effective boundary parameters.

This concludes the discussion section of this thesis. In the final chapter, we summarize the work and reach conclusions about the techniques developed.

9 Conclusion

This thesis introduces two techniques for improving the execution time of in-memory hash joins in morsel-driven systems. These techniques mitigate two issues when the data is highly skewed — poor cache performance and poor thread utilization.

Hash tables often use linked lists to hold the tuples in a hash bucket. Because linked list nodes are not adjacent in memory, they tend to have poor cache performance. This problem is exacerbated by skew keys, as a skewed build side will result in a long collision list, and a skewed probe side will result in repeated collision list iteration.

To deal with this issue, we introduced Node Compaction. This involves copying sections of collision linked lists into compact nodes containing arrays of tuples. This method results in a significant reduction in the cache miss rate when joining many skewed relations.

Skewed build keys can also result in poor thread utilization. This can occur if a probe tuple matches a significant number of build tuples while other probe tuples have far fewer matches. The thread joining the tuple with many matches will take longer than other threads to complete its work. Other threads may need to wait for the lagging thread, resulting in increased execution time and poor thread utilization. We solve this issue using a technique called Sub-morsel Stealing. This method breaks long collision lists into chunks and allows multiple threads to join a single probe tuple with different parts of the collision list. An idle thread can steal a chunk and perform the join on this chunk. This allows the work to be more evenly distributed between threads, improving thread utilization and decreasing execution time.

We implemented both of these techniques in the Umbra database system. We then ran various benchmarks to compare our method with the baseline Umbra version. Our method has a mean speedup over all TPC-H queries of 1% and a maximum slowdown of 11%. This is unsurprising, as our technique is designed to handle highly skewed data rather than uniform data like TPC-H. Our techniques significantly improve over the baseline on a large subset of the Cardinality Estimation benchmark queries [7]. On this benchmark, our methods have an average speedup of 49%, with one query having a speedup of over 87x. Given the limited downside on uniform data and the significant improvements on skewed data, the techniques developed in this thesis have the potential to improve the performance of hash join operators in main-memory morsel-driven database systems.

List of Figures

1.1 Illustration of a join between two relations with uniformly distributed keys. The output size is small and evenly distributed between morsels. 2
1.2 Illustration of a join between highly skewed build and probe relations. Due to the skew, the output size is large, though still evenly distributed between morsels. 3
1.3 Illustration of a join between skewed build relation and low-skew probe relation. The output is not well distributed, with one morsel containing most of the results. 4

2.1 Evaluation of baseline Umbra hash join on Zipf distributed build and probe relations. 14
2.2 Comparison of join with probe relation clustered by key versus randomly shuffled by key. 17

4.1 Collision list node structure for different versions of the hash table. . . . 27
4.2 An example of the hash table insertion algorithm. The global hash table and the skew bit set are pictured above. Below, the thread-local skew table holds the tuples with potentially skewed hash keys. A description of the numbered steps is provided Section 4.3.3. 33
4.3 Insertion algorithm example continued 34

5.1 Structure of collision list with multiple node build chunks. 38
5.2 Illustration of sub-morsel stealing algorithm. 46

6.1 Comparison of speedup between *NoBitset*, which applies skew optimization techniques to all keys, and *+Stealing*, which only applies techniques to suspected skewed keys. 47
6.2 Comparison of stage execution time between *NoBitset*, which applies skew optimization techniques to all keys, and *+Stealing*, which only applies techniques to suspected skewed keys. 51

7.1 Comparison of joins on Zipf distributed build and probe relations for three code versions. Probe relations have shuffled order. 55

7.2 Comparison of joins on Zipf distributed build and probe relations for three code versions. Probe relations have clustered order. 57
7.3 Comparison of execution times on TPC-H scale factor 10 58
7.4 Comparison of execution times on JCC scale factor 1 59
7.5 Comparison of execution times on Cardinality Estimation benchmark . 61
7.6 Comparison of execution times on Cardinality Estimation benchmark, separated by cyclic and acyclic queries. 62
7.7 Speedup over baseline on Cardinality Estimation benchmark, separated by cyclic and acyclic queries. 63
7.8 Speedup versus Baseline execution time for Cardinality Estimation benchmark. 64
7.9 Speedup versus Baseline execution time for Epinions queries on Cardinality Estimation benchmark. 64

List of Tables

7.1 Summary of results for Zipf tests with shuffled probe relation 56
7.2 Summary of results for Zipf tests with clustered probe relation. 58
7.3 Summary of results for TPC-H benchmark at scale factor 10 59
7.4 Summary of results for JCC benchmark at scale factor 1 59
7.5 Summary of results for Cardinality Estimation benchmark 63

Bibliography

[1] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In: *STOC*. 1996, pp. 20–29.

[2] P. Bagwell. Fast Functional Lists, Hash-Lists, Deques, and Variable Length Arrays. Technical Report LAMP-REPORT-2002-003. Ecole polytechnique fédérale de Lausanne, 2002.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: sort vs. hash revisited. In: *PVLDB*. Vol. 7(1). 2013.

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

[5] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In: *SIGMOD*. 2011, pp. 37–48.

[6] P. Boncz, A.-C. Anatiotis, and S. Kläbe. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In: *Performance Evaluation and Benchmarking for the Analytics Era, TPCTC*. 2018, pp. 103–119.

[7] J. Chen, Y. Huang, M. Wang, S. Salihoglu, and K. Salem. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. In: *PVLDB*. Vol. 15(8). 2022, pp. 1533–1545.

[8] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. In: *Communications of the ACM*. Vol. 13(6). 1970.

[9] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. In: *Journal of Algorithms*. Vol. 55(1). 2005, pp. 58–75.

[10] J. Dean. Software engineering advice from building large-scale distributed systems. `http://research.google.com/people/jeff/stanford-295-talk.pdf`. Accessed: 2023–11-10. 2007.

[11] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In: *VLDB*. 1992, pp. 27–40.

[12] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: *Discrete Mathematics & Theoretical Computer Science*. 2007.

[13] T. Goetghebuer-Planchon. Tessil/hopscotch-map: A fast and memory efficient hash map. Accessed: 2023-11-12. 2022. URL: `https://github.com/Tessil/hopscotch-map`.

[14] G. Graefe. Volcano-an extensible and parallel query evaluation system. In: *IEEE Transactions on Knowledge and Data Engineering*. Vol. 6(1). 1994, pp. 120–135.

[15] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In: *SIGMOD*. Vol. 23(2). 1994, pp. 243–252.

[16] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Data Base Machine and Its Architecture. In: *New Gen. Comput.* Vol. 1(1). 1983, pp. 63–74.

[17] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In: *SIGMOD*. 2014, pp. 743–754.

[18] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. In: *IEEE Transactions on Knowledge and Data Engineering*. Vol. 14(4). 2002, pp. 709–730.

[19] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In: *ICDT*. 2005, pp. 398–412.

[20] R. Morris. Counting large numbers of events in small registers. In: vol. 21(10). 1978, pp. 840–842.

[21] T. Neumann. Efficiently compiling efficient query plans for modern hardware. In: *PVLDB*. Vol. 4(9). 2011, pp. 539–550.

[22] T. Neumann and M. Freitag. Umbra: A Disk-Based System with In-Memory Performance. In: *CIDR*. 2020.

[23] H. Q. Ngo, C. Ré, and A. Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. In: *SIGMOD Rec.* Vol. 42(4). 2014, pp. 5–16.

[24] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. In: 9(3) (2015), pp. 96–107.

[25] W. Rodiger, S. Idicula, A. Kemper, and T. Neumann. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In: *ICDE*. 2016, pp. 1194–1205.

[26] D. A. Schneider and D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In: *SIGMOD*. 1989, pp. 110–121.

[27] M. H. Selmi. Evaluation of Adaptive Join Indexes. MA thesis. University of Augsburg, Technical University of Munich, Ludwig Maximilian University, 2023.

[28] Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling lists. In: *Conference on LISP and Functional Programming*. 1994, pp. 185–195.

[29] B. Wagner, A. Kohn, and T. Neumann. Self-Tuning Query Scheduling for Analytical Workloads. In: *SIGMOD*. 2021, pp. 1879–1891.

[30] C. B. Walton, A. G. Dale, and R. M. Jenevein. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. In: *VLDB*. 1991, pp. 537–548.