Global Hash Tables Strike Back! An Analysis of Parallel GROUP BY Aggregation

Daniel Xue
University of Pennsylvania
danxue@seas.upenn.edu

Ryan Marcus
University of Pennsylvania
rcmarcus@seas.upenn.edu

ABSTRACT

Efficiently computing group aggregations (i.e., GROUP BY) on modern architectures is critical for analytic database systems. Today, hash-based methods predominantly use a partitioned approach, in which incoming data is partitioned by key so that every row for a particular key is sent to the same partition. In this paper, we revisit a simpler strategy: a fully concurrent aggregation technique using a shared hash table. While approaches using general-purpose concurrent hash tables have generally been found to perform worse than partitioning-based approaches, we argue that the key ingredient is *customizing the concurrent hash table for the specific task of group aggregation*. Through experiments on synthetic workloads (varying key cardinality, skew, and thread count), we demonstrate that in morsel-driven systems, a purpose-built concurrent hash table can match or surpass partitioning-based techniques. We also analyze the operational characteristics of both techniques, including resizing costs and memory pressure. In the process, we derive practical guidelines for database implementers. Overall, our analysis indicates that fully concurrent group aggregation is a viable alternative to partitioning.

PVLDB Reference Format:
Daniel Xue and Ryan Marcus. Global Hash Tables Strike Back! An Analysis of Parallel GROUP BY Aggregation. PVLDB, 19(3): 523 - 535, 2025.
doi:10.14778/3778092.3778110

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/danielxue/global-hash-tables-strike-back.

1 INTRODUCTION

Group aggregation, represented in SQL via GROUP BY, is a fundamental operation in analytical query processing, especially decision support workloads [25]. One analysis by the Snowflake team found that over half of production queries contain a GROUP BY [31]. While many different schemes for group aggregation exist (e.g., sorting), this paper will analyze hashing-based aggregation techniques.

Analytic database systems today are quite diverse. For example, DataFusion [19] follows a Volcano-style [13] block iteration approach (i.e., "pull"), whereas DuckDB [28] follows a HyPer-inspired morsel-driven parallelism [20] approach (i.e., "push"). Despite drastic differences in their execution models, nearly all of today's analytic database systems use partitioning techniques to parallelize group aggregations. To the best of our knowledge, the partitioning approach proposed by Raman et al. [29], which uses preaggregation to alleviate pressure from data skew, has become dominant in many modern analytic systems [17, 19, 20]. We provide background on partitioning-based techniques in Section 2.2.

An alternative to partition-based approaches is to use a global concurrent hash table. Instead of partitioning keys into groups, each worker concurrently accesses a global hash table. In theory, such a hash table has many operational benefits, such as lower memory usage, reduced impact of skew, and simpler implementations. In practice, despite several improvements to general-purpose concurrent hash tables [21, 23, 26], contention effects and synchronization overhead represent significant scalability barriers.

But is it really surprising that general-purpose concurrent hash tables perform worse than purpose-built solutions like partitioned group aggregation? After all, general-purpose hash tables must support a myriad of operations that are irrelevant to group aggregation, such as deletes and shrinking. General-purpose hash tables must also be optimized for a wide range of workloads, where deletes, inserts, and lookups might come from different threads, in different distributions, and at different times. A concurrent hash table optimized for group aggregation could sidestep most of this complexity: the only required operation is the aggregation of a new value, and it is reasonable to assume that every thread will invoke this operation consistently until all data is consumed.

Main Result. In this paper, we explore the design space of group aggregation algorithms using a global concurrent hash table, comparing against the state-of-the-art partitioned approach in morsel-driven [20] execution engines. Most significantly, we find that a simple, purpose-built concurrent hash table using linear probing can scale well on modern multi-core hardware and match or even outperform partitioning-based approaches. Our results do not show that one approach is better than the other, but instead highlight the operational benefits and costs of both. Throughout our exploration, we make specific recommendations for database implementers.

Our implementation of group aggregation with a global concurrent hash table closely tracks the implementation in MonetDB [4]. Each worker, upon receiving a row, first obtains an integer "ticket" for that row's grouping key from a hash table. This ticket uniquely identifies each group and serves as an index to locate the partially aggregated value for that group, which is then updated based on the value in the incoming row. We provide a detailed description of this procedure in Section 2.3. This two-phase procedure, ticketing and partial aggregate update, is repeated for each row and opens up a large number of possible designs, which we explore.

Proceedings of the VLDB Endowment, Vol. 19, No. 3 ISSN 2150-8097.
doi:10.14778/3778092.3778110

Ticketing. In the initial phase of fully concurrent group aggregation, each unique group is assigned an integer "ticket." This operation can be performed with a concurrent hash table that atomically checks if a key is already in the table, and then returns the ticket for that key if so and, if not, inserts a new ticket into the table. Surprisingly, many general-purpose hash tables cannot perform this operation atomically or do not optimize for this particular case. As a result, there is significant room for improvement by building a specialized fast path for this particular operation, while avoiding adding extra overhead for unnecessary operations like deletes. We test several implementations based on atomics and fine-grained locking and show that simple purpose-built hash tables can significantly outperform their complex general-purpose counterparts.

Partial Aggregate Update. After ticketing, each worker must apply the relevant aggregation function to the values associated with each row. This can be done in either a thread-specific way (e.g., each thread maintains local aggregation storage, merging the results at the end) or by using concurrent access to global space. We explore the pros and cons of both approaches and characterize their operational tradeoffs. Specifically, we find that concurrent access to global space works well in the absence of heavy hitters, and we propose a simple thread local approach that works well except when every grouping key is unique.

Organization. Our experimental study is organized as follows. In Section 2, we explain our assumed model of execution and introduce the basics of partitioned and fully concurrent aggregation. We also introduce the experimental setup used throughout our experiments. In Section 3, we investigate the design space of aggregation with a concurrent hash table. In Section 3.1, we explore the ticketing step, and in Section 3.2, we explore the aggregation step. In Section 4, we analyze the end-to-end performance of both approaches. In Section 5 we discuss related work, before concluding and outlining future directions in Section 6.

2 PRELIMINARIES

In this section, we first establish the model of query execution we are operating within and discuss the relevant constraints it places upon our work. We then provide an overview of the two models of aggregation assessed in this paper: partitioned and fully concurrent.

2.1 Model of Execution

Leis et al. [20] introduced the morsel-driven framework for parallel execution. In this model, parallelism is achieved by breaking down work into data fragments called "morsels." Morsels are dynamically assigned to threads in a pool, using work-stealing for even distribution. A thread carries a morsel through the entire pipeline before grabbing another. Morsel-driven parallelism is often used with columnar data representations (e.g., HyPer [20]), with morsels serving as units for vector-at-a-time execution. Operating on a dense vector of values enables important optimizations, such as SIMD instructions and amortization of interpretation overhead [15]. However, the morsel-driven model also restricts some operator implementations. To conform with pipelining, operators must operate on chunks of data at a time and cannot access the remaining tuples.

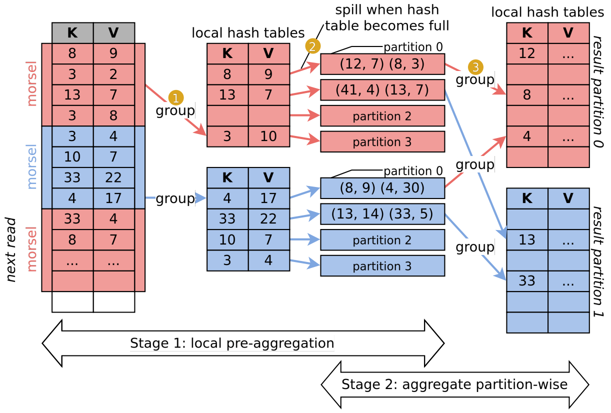

Figure 1: A sample execution of partitioned aggregation. This diagram is modified slightly from the one presented in [20] and is provided to contrast with the sample execution by our fully concurrent model on the same input in Figure 2.

In this work, we focus on the morsel-driven execution model due to its popularity in open source OLAP systems (e.g., DuckDB [28]). While our conclusions and recommendations may apply to other execution models (e.g., DataFusion [19] uses operator-level parallelism but adopts an aggregation strategy from a morsel-driven system), different execution models may also lead to significant changes in performance properties. We leave investigations under alternative execution models to future work.

2.2 Partitioned Aggregation Model

In a partitioned approach to aggregation, synchronization is avoided by splitting the key domain among threads. A naive partitioning strategy assigns incoming tuples to their proper thread, typically by taking the radix or mod of the key's hash, which then aggregates the assigned tuples locally. Unfortunately, this method suffers significantly from data skew. A simple solution is to use a local hash table to preaggregate heavy hitters before partitioning [20, 29, 36].

This method of partitioned aggregation (see Figure 1) consists of two stages: local preaggregation and partition-wise aggregation. First, each thread aggregates all values from its assigned morsel(s) in a fixed-sized local hash table ❶. When a hash table fills up, the partially aggregated values are spilled into partitions ❷. In the second stage, partitions are exchanged between threads, and each thread combines its partitions to compute the final answer ❸.

This algorithm is motivated by the need for skew resistance, avoiding uneven work distribution by spreading the work of aggregating heavy hitters among all threads in the local preaggregation stage. However, for higher cardinality workloads, there is repeated spilling from the local aggregation tables. The constant spilling results in each tuple essentially being aggregated twice, once in each stage of the algorithm, creating a significant source of overhead.

The local preaggregation approach to partitioned aggregation has been used by several real-world systems, including DuckDB [17] and DataFusion [19]. Due to the widespread adoption of this particular algorithm and its good scaling behavior [17, 36], it is used here as the baseline partitioned aggregation method.

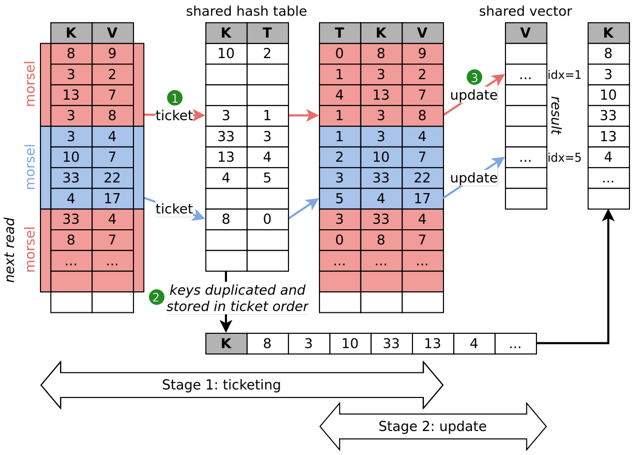

Figure 2: A sample execution of our fully concurrent aggregation model using the same instance as in Figure 1.

2.3 Fully Concurrent Aggregation Model

To perform a group by aggregation in a fully concurrent manner, each thread must aggregate an arbitrary morsel of data from start to finish, without any key distribution. We separate fully concurrent aggregation into two different steps: *ticketing* and partial aggregate *update*, as shown in Figure 2.

During ticketing, a concurrent shared hash table is used to map each key value to an integer "ticket" ❶. This mapping is one-to-one: each unique key is granted a single, unique ticket, and the ticket assigned to a key is consistent across all threads due to the use of a shared hash table. A ticket conceptually represents the location where the corresponding partial aggregate to be updated is stored, which in our implementation is simply an index of a vector. Alternatively, the ticketing step can be viewed as incrementally creating a perfect hash function. When we insert a new key, we also duplicate the key and store it in a vector in ticket order ❷, which is later used to reconstruct the key column. This stage is the "group by" part of group by aggregation.

As a result of the previous stage, we have a morsel-sized column of tickets, stored as a vector. This column of tickets (**T**) and the input morsel of values (**V**) are fed to the update step ❸, where the query's specified aggregation function(s) are applied (e.g., incremented for `COUNT`, added for `SUM`, etc.). This step is the "aggregation" part of group by aggregation.

Ticketing Indirection. Note that the layer of indirection separating the partial aggregate from the hash map is already commonplace in practice. For example, it is used by MonetDB [4] to enable vectorized execution, as well as by DuckDB [17] and DataFusion [19]. Indirection also enables optimizations only possible when acting on a dense column of values, such as SIMD, or to amortize the cost of dynamic type resolution in non-compiled systems.

Given that indirection is already commonplace, we formalize it in our model to take advantage of its logical properties. In particular, since each unique key is assigned a ticket exactly once, our hash table workload is reduced to only lookups and inserts. We take advantage of this fact by designing a customized concurrent hash table that supports only these operations. This indirection also pushes concurrent updates to the second stage. Thus, concurrent updates act on a vector of values rather than units of a hash table, allowing for alternate ways of synchronizing writes that do not degrade the performance of the ticketing hash table.

Table 1: Test machine hardware specifications.

Model	Cores	Clock GHz	L1 MB	L2 MB	L3 MB
AMD EPYC 9454P	48 / 96	2.75	3	48	256
Ampere Altra Q80-30	80 / -	3.00	10	80	-
Intel Xeon Gold 5412U	24 / 48	2.10	2	48	45

2.4 Experimental Setup

Since our analysis and discussion will integrate several microexperiments, we first describe our experimental setup. Readers may wish to skip this section and reference it as needed.

We primarily run our experiments on a machine running an AMD EPYC 9454P processor with 48 cores @2.75GHz ("AMD") and supporting simultaneous multi-threading (SMT) up to 96 threads. To test a diversity of architectures, we also run tests on an ARM system running an Ampere Altra Q80-30 with 80 cores (no SMT support) @3.00GHz ("ARM") and an Intel system running an Intel Xeon Gold 5412U with 24 cores (SMT up to 48 threads) @2.10GHz ("Intel"). See Table 1 for a detailed comparison of the hardware. All machines have 256GB of RAM.

For all platforms, clock boost was disabled, as was address space layout randomization (which caused significant performance variation from run to run). All experiments were implemented in Rust and compiled in release mode using the `target-cpu=native` flag.

Datasets. Our tests use synthetic datasets consisting of 100 million key-value pairs. Each key and value is a 64-bit integer (other than in Section 4.4). We varied the cardinality of the dataset, labeled low, high, and unique, which consist of 1000 unique keys (100% lookup/0% insertion on ticketing), 10 million unique keys (90% lookup/10% insertion), and 100 million unique keys (0% lookup/100% insertion) respectively, distributed perfectly uniformly. For the high cardinality dataset, we also add two types of skew: a Zipfian distribution with exponent parameter $s = 0.8$, as well as a heavy-hitter dataset where 50% of the dataset consists of the same key.

Workload. As a demonstrative example of common aggregation functions, we use `SUM` for all tests. All experiments (except those in Section 4.5) assume perfect cardinality estimation and thus perfectly sized hash tables and partial aggregate vectors. All results are obtained by taking the median latency of a given workload after 9 runs (not including a warm-up run).

3 FULLY CONCURRENT AGGREGATION

Here, we investigate the design space for each stage of fully concurrent aggregation. We discuss methods for and execute microbenchmarks of each stage in isolation to identify promising methods, which we later test end-to-end in Section 4.

Algorithm 1 Get-or-insert implementation for Folklore*.

```
function GET_OR_INSERT(key, ticket, table)
    $idx := key.hash() \bmod table.len()$
    loop        ▷ Atomic operations use acquire/release ordering.
        $(t, k) := (table[idx].t.load(), table[idx].k.load())$
        if $t \geq 2$ then        ▷ Fast path lookup.
            if $k = key$ then
                return $t - 2$
            else
                $idx := (idx + 1) \bmod table.len()$
            end if
        else if $table[idx].t.CAS(0, 1)$ then        ▷ Insert.
            $table[idx].k.store(key)$
            $table[idx].t.store(ticket + 2)$
            return $ticket$
        end if
    end loop
end function
```

3.1 Ticketing

As described in Section 2.3, the ticketing step is performed using a concurrent hash table to map each unique key to a unique and immutable "ticket." In this section, we first establish the interface of hash tables designed for ticketing and the process used to generate tickets. Then, we discuss various candidate hash table designs, followed by experimental evaluation and discussion.

Interface. When a new key arrives, an insert operation is required. Following a successful insertion of a key, all future requests must look up the same inserted ticket value. In many concurrent hash table designs, this lookup operation can be achieved with much less overhead than an insert (e.g., shared locks are cheaper than exclusive locks). However, since we do not know ahead of time whether an insert is necessary (i.e., we do not know if a particular key has already been given a ticket), it is crucial to provide an efficient fast path for an atomic get-or-insert operation. Such an operation is rarely supported out-of-the-box by concurrent hash maps, but the same effect could often be achieved using an entry API or with a lookup followed by a non-overwriting insert.

To efficiently retrieve keys in the same order as our aggregate values when materializing our final results, we also need the hash table to store a copy of keys in ticket order. This creates some performance overhead but is generally efficient since there is no contention–only the thread issuing a ticket is responsible for storing a copy. The ticket-order copy of keys can also be maintained as the only copy to decrease memory overhead since the ticket value can be used to lookup the key during hash map operations, but would come at the cost of an additional cache miss per lookup.

Hash Table Designs. We benchmark a variety of state-of-the-art hash table designs. These include cuckoo hashing [21], Iceberg hashing [2, 26], Rust's Leapfrog [6] library, which uses leapfrog probing [27], and Rust's popular DashMap [34].

We also implement a variant of the Folklore hash table proposed by Maier et al. [23], a lockless linear probing hash table. Our implementation leverages that we control the value being stored (i.e., the ticket). We reserve 0 to indicate the key-value pair of the slot is empty and 1 to indicate a write is in progress. These properties allow our design to use only a single-word CAS instruction instead of the two-word version (which is not universally supported) required by the canonical implementation. We denote our implementation Folklore* to indicate the differences. The GET_OR_INSERT procedure for this design is outlined in Algorithm 1.

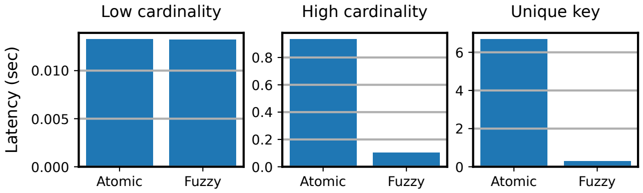

Figure 3: Performance of the Folklore* hash table with a fuzzy ticketer as opposed to an incrementing atomic counter. Performance is evaluated with latency (lower is better).

Our implementations additionally benefit from vectorized execution that optimizes hashing and amortizes the acquisition of a read lock on the shared table (used to ensure the correctness of resizing).

Micro Experiment: Generating Ticket Values. Generating a ticket value is simple on the surface, but actually presents a potential source of contention. Multiple threads must avoid issuing the same ticket to different keys. A naive implementation of a "ticketer" would use an atomic counter, incremented whenever a thread needs a new ticket. However, contention on the counter is high for insert-heavy workloads. To combat this issue, one can use a *fuzzy ticketer* that assigns each thread a range of tickets to issue at a time. Each thread only needs to concurrently access an atomic value when it has exhausted its assigned range. The tradeoff is that the vector of partial aggregates may no longer be perfectly dense, but the number of gaps is bounded linearly by the number of threads and does not cause much overhead to remove.

There is a large latency gap on insert-heavy workloads between using a pure atomic counter and our fuzzy ticketer, as seen in Figure 3. This micro-benchmark is tested on our best-performing hash table design, Folklore*, at 48 threads. Latency in the high cardinality workload is greater for the pure atomic method by a factor of 9.0x, and even worse for unique keys. Since using a single atomic ticket value degrades performance when there are many unique keys, we recommend that implementers amortize the cost of concurrent accesses across multiple inserts, such as with our fuzzy ticketer.

Micro Experiment: Memory Allocation. For memory-intensive database workloads, such as high-cardinality aggregations, the performance of the memory allocator can have a notable impact on the overall latency [9]: a global hash table must be allocated. The allocation of a large global hash table for high cardinalities is not only time-consuming but is crucially also single-threaded. This single-threaded work is a constant cost that does not scale with threads and can severely degrade performance at high thread counts. In Table 2, we can see that at 24 threads, pre-execution memory allocation can be more than half of total latency on some systems.

Table 2: Impact of zero-allocation on Folklore* performance on AMD and Intel at 24 threads with unique keys.

System	AMD		Intel	
Zero-allocate	No	Yes	No	Yes
Total latency (s)	0.48	0.39	0.91	0.28
Allocation (s)	0.24	0.00	0.67	0.00
Ticketing (s)	0.23	0.38	0.23	0.27

Zero-allocation can be used to optimize the memory allocation step and improve the scaling of concurrent hash tables. Although zero-allocation is restricted to only certain types, this restriction is not a problem for the ticketing use case as long as the hash table is designed to not prematurely access unused key fields. Zero-allocation, often a special type of call to the allocator (e.g., `calloc` in C), enables a number of optimizations by the operating system. A particularly important optimization is "copy-on-write," which defers allocation of a page to its first access rather than on the initial call to the allocator. As a result, memory allocation is no longer on the single-threaded critical path. When using zero-allocation, we see in Table 2 that the initial allocation becomes negligible in exchange for a modest performance hit during the actual ticketing step. Thus, we encourage practitioners to consider zeroed memory allocation strategies for large tables.

Ticketing Performance. In Figure 4, we show the performance and scaling behavior of these designs across different cardinalities and data distributions (see Section 2.4). We find that Folklore* consistently performs best among the tested designs, despite its simple implementation. Crucially, Folklore* performance is resilient to data skew due to its fast path lookup. It also exhibits excellent scaling for low cardinality workloads, although the speedup it achieves at high thread counts degrades as cardinality increases. At 48 threads (the number of real cores on the system), Folklore* achieves a 37.6x speedup for low cardinality, 22.8x for high cardinality, and 13.8x with unique keys. SMT generally yields little benefit.

Surprisingly, cuckoo hashing exhibits very poor scaling in high contention lookup workloads, despite us implementing a fast path and prior literature indicating good performance [21]. We attribute this behavior to the use of fine-grained locks on buckets. Even though read locks are used on the fast path, bookkeeping by the locks may cause contention when multiple threads access the same resource in rapid succession. DashMap also uses read locks, which could explain its similar performance characteristics.

In Figure 5, we perform topdown analysis [35] on a selection of the hash table designs to understand the performance characteristics of each method. A more detailed explanation of our methodology can be found in Section 4.2. Our hypothesis regarding bookkeeping on locks during cuckoo hashing aligns with the spike in backend bound behavior when multi-threading on the low cardinality dataset (contention on atomics, used for bookkeeping in read locks, is considered backend bound). Similar behavior occurs for the heavy hitter dataset and for DashMap (not shown).

In contrast, LeapMap and Folklore* are fully lock-free (and Iceberg is lock-free on its fast path lookup). These characteristics explain the gulf in scaling behavior in the low cardinality case, where the speedup at high thread counts is starkly higher for the latter three methods compared to the former two that use read locks. Further, by avoiding write locks with all methods, we avoid idle time from contention. Thus, it is essential to have a lockless "fast path" for reading previously-inserted values.

Figure 5 also reveals that Folklore* shows less backend bound behavior compared to other methods, suggesting superior cache performance, a benefit of linear probing. In fact, when looking at raw performance counters, across the data distributions and thread counts (up to 48), in all but two cases Folklore* achieves the lowest number of cache misses, and in the majority has the lowest cache miss rate. For example, for the high cardinality dataset at 48 threads, Folklore* has 1.7e8 cache misses and a 39.9% cache miss rate. The next best by total cache misses (Leapfrog) has considerably higher cache miss count and miss rate at 2.1e8 and 46.2% respectively. That said, Folklore* still spends a substantial proportion of time backend bound at high cardinalities, explaining the poorer scaling as cardinality increases, as memory now becomes a bottleneck.

Discussion. An interesting finding from the experiment results is that for our lookup and insert-only workload, even a simple linear probing design (Folklore*) achieves excellent performance. A contributing factor is that the typical downside to linear probing, inefficient deletions, is a non-issue given our workload. We conclude that to perform efficient ticketing, fancy hash tables are not required: linear probing is all you need.

One meta-explanation for Folklore*'s unexpected dominance lies in the benchmark workloads used by hash table researchers. The existing literature on concurrent hash table designs often does not test certain workloads important for our application. For example, neither of [14, 21] test a lookup-only workload (aligning with low cardinality key spaces), and all workloads tested contained updates and deletions (of which there are none in our formulation of group aggregation). Although this particular selection of workloads makes sense for a general-purpose hash table, the results do not apply to group aggregation. Put another way, the omission of read-heavy, delete-free workloads can cause hash table designs to be poorly optimized for our ticketing use case despite strong performance in mixed, general-purpose workloads. We recommend that database implementers use caution when evaluating benchmarks for general-purpose hash tables since group by aggregation has a distinct profile that is often overlooked.

Even when benchmarking specifically for aggregation use cases, as in Maier et al. [23], chosen workloads often assume in-table aggregation (which is not feasible without query compilation, since there are too many combinations of key types and aggregation functions). These workloads are update-heavy and do not take advantage of the lookup and insert-only semantics of ticketing. This misalignment has yielded misguided conclusions that concurrent hash tables are insufficient for aggregation purposes in the presence of skew. On the contrary, the simple nature of a lookup and insert workload makes skewed workloads a non-issue for many existing concurrent hash table designs.

Finally, we make the observation that our definition of ticketing reduces the task to that of finding a minimal perfect hash function; that is, the purpose of ticketing is to assign each key a unique, densely packed integer, which is exactly what a minimal perfect

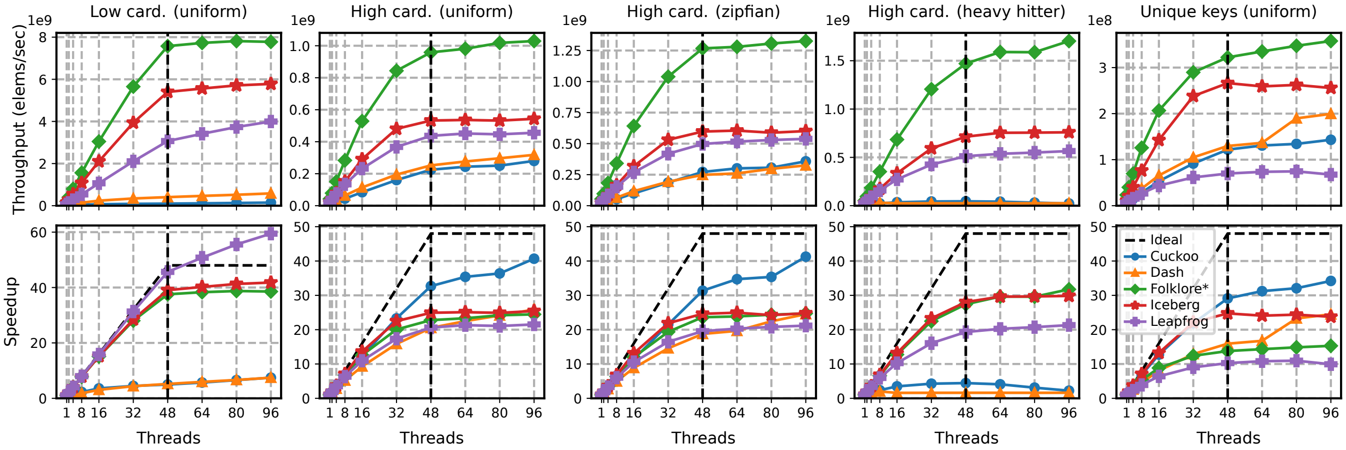

Figure 4: Scaling behavior of various hash maps for ticketing. The top row measures performance as throughput (higher is better) against thread count. The bottom row plots the speedup factor against thread count, measured as the single-threaded latency divided by the latency at a given thread count. Ideal scaling is the linear function *speedup = threads*, which is plotted as a dashed line. The vertical dashed line represents the threshold thread count between physical cores and SMT.

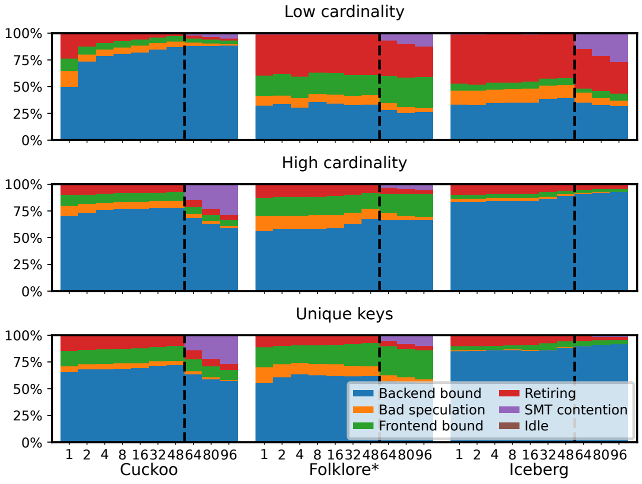

Figure 5: Breakdown of time spent on work by processor per workload and thread count. Shows both idle time and topdown analysis of time performing work (see Section 4.2).

hash function does. Gaffney and Patel [11] found that significant speedups can be achieved when integrating perfect hash functions into DuckDB's aggregation pipeline. Our formulation of ticketing aligns perfectly with this notion of perfect hashing, and ticketing could likely also greatly benefit from perfect hashing (i.e., "skipping" the ticketing step). Perfect hashing could essentially remove all contention from the ticketing phase, greatly accelerating fully concurrent aggregation.

3.2 Partial Aggregate Update

In the update step, we "actually do" the aggregation and modify the partial aggregate value corresponding to each ticket based on the associated row. Concurrency control problems that we avoided in the ticketing step have been deferred to this stage. In this section, we explore two classes of update methods, concurrent and thread local. We evaluate and discuss the methods and identify the situations where each exhibits good performance and scaling.

Concurrent Update Method. A naive but general-purpose solution to managing concurrency in this step is to protect each cell of the vector of partial aggregate values with a lock. For each ticket, we acquire the lock on its cell in the vector of partial aggregates, update the value, and release the lock.

Another simple approach is to have each partial aggregate be an atomic. It is straightforward to perform aggregation functions such as COUNT, SUM, and MIN/MAX, but other more complicated aggregation functions may not be easy to implement with atomics. One general solution is to perform a lookup on the current partial aggregate, apply the update, and then use a CAS instruction to update the partial aggregate. However, this method increases the number of atomic operations, and hence contention, and can potentially suffer from the ABA problem [7] during implementation.

These fully concurrent update methods are simple to implement and memory efficient; however, they suffer at higher levels of contention: if there is a heavy hitter in the data, every thread may simultaneously and repeatedly try to update the same partial aggregate value. There is potential for operation-specific optimizations, such as the inclusion of a fast-path for MIN/MAX that avoids an atomic write if not needed, but in the general case, skewed workloads continue to present issues.

Thread Local Update Method. To mitigate issues with contention, we explore a thread local approach to updates where each worker thread updates partial aggregates in its own thread local vector, eliminating all contention at the cost of a merge of all partial aggregate values at the end of the aggregation. Although the total work of the merge step scales with the number of threads, because the vectors of partial aggregates are all in the same order (ordered by ticket), the merge is trivially parallel and cache-efficient, partially mitigates the overhead. This merge can be viewed as the "transpose" of the merge done at the end of the partitioned case: in the partitioned case, each worker has fully aggregated values but only some of the keys, whereas in the fully concurrent case, each worker has partially aggregated values but all of the keys.

Unfortunately, the work per thread does not decrease asymptotically as thread count increases. Since each of k threads is assigned $\frac{n}{k}$ tickets to merge (where n is the number of unique keys), and there are k threads worth of partial aggregates per ticket, each thread aggregates over $\frac{n}{k}k$ partial aggregates, yielding $O(n)$ runtime.

Another challenge is that memory usage scales linearly with the number of threads and distinct keys, requiring a vector the size of the entire key space per thread. For very large datasets at high thread counts, this overhead could be a concern. One mitigating factor is the fact that the vectors of partial aggregates are dense, as opposed to the hash tables allocated per thread in partitioned aggregation, reducing the gap between the two techniques' memory usage. We quantify the memory overhead in Section 4.6.

Evaluation. In Figure 6, we plot the performance of these update methods in isolation–that is, without ticketing. Keys are given as an integer from 0 to the max key, which is used directly as a ticket (i.e., a perfect hash function). We set up the experiment in this way to isolate the scaling behavior of each specific aggregation method. It is important to note that contention effects are far more pronounced in these isolated benchmarks because when running end-to-end, the time it takes to ticket causes fewer threads to be at the update step at the same time, decreasing contention.

Both thread local and atomic updates demonstrate strong scaling behavior in some workloads but neither is a clear winner across all workloads. Thread local updates have superior performance in the presence of high contention (i.e., low cardinality datasets or skewed distributions) but degrade in performance as cardinality increases. This is because, at lower numbers of unique keys, the underlying vectors of partial aggregates are small enough that there is minimal overhead from merging. At low cardinality, thread local updates achieve a substantial speedup of 33.7x at 48 threads.

At higher cardinalities, the thread local method demonstrates inverse scaling behavior at higher thread counts, actually decreasing in throughput after 32 threads for the high cardinality case and 8 threads in unique key case. Still, at its peak, the thread local achieves a 9.2x speedup (at 32 threads) on the high cardinality case. The unique case has a peak speedup of only 3.5x (at 8 threads). Crucially, though, the speedup is consistent no matter the skew–since the partial aggregates have no contention, skew can only help due to greater cache locality and leads to no performance degradation. This property makes thread local updates a good option even with relatively high cardinality workloads due to its consistency.

Meanwhile, atomic updates exhibit better scaling in workloads with low contention and high cardinality. For the high cardinality and unique key workloads, when in the absence of skew, atomic updates are a clear winner. In fact, in the high cardinality case, atomics achieve a blazing 34.7x speedup at 48 threads. Surprisingly, atomic updates exhibit serviceable scaling even in the presence of some skew (achieving 10.3x speedup on the Zipfian workload) but fall short when the skew becomes too great (the heavy hitter workload). Fine-grained locked updates exhibits similar behavior to atomic updates, since it is bottlenecked by similar contention effects, but tends to be the worst performer due to its higher overhead.

Table 3: Desiderata fulfilled by each update method. For cardinality, skew, and thread count, the range of workloads where the given update method performs well in is given. For memory usage, the asymptotic behavior is given as a factor of n (the number of unique keys) and k (the number of threads).

	Atomic	Locking	Thread Local
Cardinality	High-Unique	High-Unique	Low-High
Skew	Low-Medium	Low	Low-High
Threads	Low-High	Low-High	Low-Medium
Memory	$O(n)$	$O(n)$	$O(kn)$

Discussion. Given the varied performance characteristics of these update methods, summarized in Table 3, ideal performance can be achieved by adaptively choosing the best method. Although imperfect, optimizers can estimate the number of unique keys [16] to choose the update method on a per-query basis.

There is a concerning performance gap for workloads that are both high cardinality and highly skewed. However, we note that the most extreme performance degradation occurs in rarer cases, such as when there is a particularly high number of duplicate values or when elements are unique. Furthermore, it is important to note poor scaling in the partial aggregate update step does not preclude overall good performance from fully concurrent aggregation. In Section 4.2, we find that the ticketing stage typically accounts for the bulk of overall execution time (since hash table operations tend to have much higher overhead than updates on a vector). Therefore, since we have found that ticketing exhibits good scaling behavior, provided that the poor scaling in the update step is not bad enough to be a bottleneck, the overall system still can scale well even with degraded performance in the update step.

We believe that future work can make significant improvements to this step. In particular, a system that can combine both atomic or locked updates with thread local updates could take advantage of the benefits of both systems. This discussion bears similarity to hybrid aggregation approaches [5, 10] that combine a shared hash table with thread local hash tables for heavy hitters. We also believe that there is room for optimizations that take advantage of the vectorized nature of these updates. If locks can be obtained a vector at a time, we may be able to reduce locking overhead.

4 END-TO-END EXPERIMENTS

In this section, we combine the ticketing and update steps of fully concurrent aggregation and benchmark end-to-end performance. We also implement partitioned aggregation in the same environment and compare the results. For all fully concurrent aggregations, we use Folklore* for the ticketing step because it had the highest throughput across all tested cases.

Recall that our experimental setup is described in Section 2.4. Our experimental analysis is broken into four sections. **Scaling (4.1)** presents experiments measuring throughput and scaling behavior across different workloads. **Explaining Behavior (4.2)** aims to

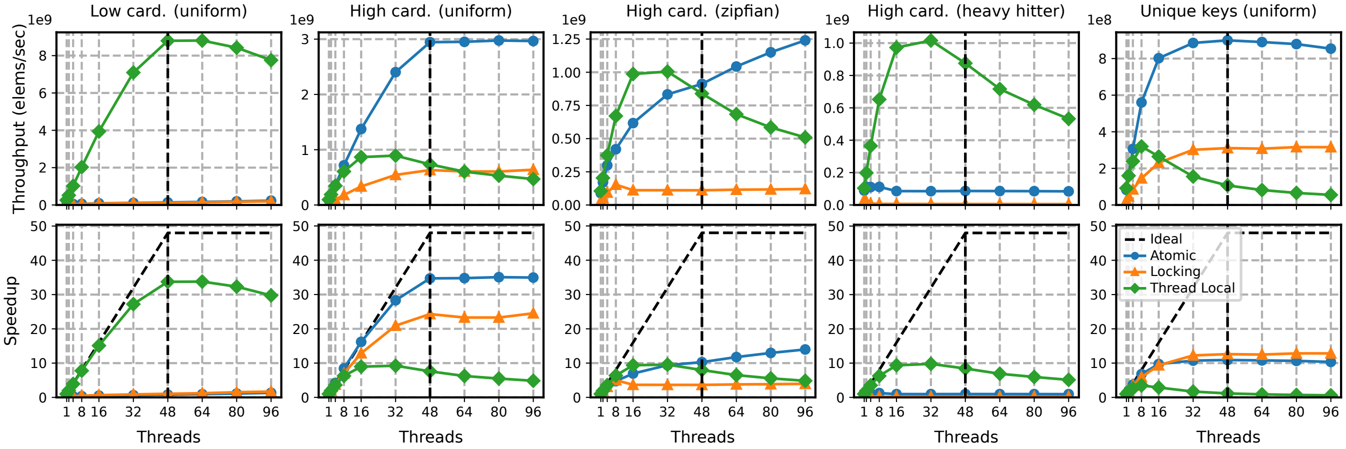

Figure 6: Scaling behavior of various partial aggregate update methods across different data distributions.

explain the observed scaling behavior by analyzing time spent on each stage of aggregation and profiling results. **Other Platforms (4.3)** repeats the main experiment on other architectures and discusses the differences in performance. **Tuple Size (4.4)** varies the size of the aggregate tuple. **Resizing (4.5)** compares aggregation methods in the case of poor cardinality estimation resulting in resizing behavior. **Memory (4.6)** compares the peak memory usage of each aggregation method.

4.1 Scaling

We evaluate end-to-end performance (including both the ticketing and update steps) in Figure 7. We test the atomic and thread local methods for fully concurrent aggregation and graph the results against those for partitioned aggregation using local preaggregation. The locked update approach is omitted because it performs strictly worse than atomics as a concurrent update method.

Low Cardinality. At low cardinality, fully concurrent thread local aggregation outperforms the other methods at all thread counts, achieving 1.8x the throughput of the partitioned approach at 48 threads. As in Section 3.2, atomic aggregation does not scale.

High Cardinality. In the high cardinality case, we find that thread local aggregation has a clear advantage over partitioning at lower thread counts. At this cardinality, the local hash table used by partitioned aggregation spills most entries, which causes each value to be aggregated twice (once locally and once partition-wise). However, at high core counts the benefit narrows. For the non-heavy-hitter distribution, the atomic method also displays an advantage over partitioning, even with moderate Zipfian skew.

Thread local aggregation does not start decreasing in throughput until after 48 threads, despite doing so at lower thread counts in the isolated update benchmarks from Section 3.2. Upon analyzing the time spent on each stage in Section 4.2, we see that the update step is fast enough at low thread counts such that the behavior of the ticketing step dominates the runtime.

Surprisingly, partitioned aggregation continues to scale even when thread count exceeds the number of physical cores. However, even at the high range of thread counts, partitioned aggregation is around parity or slightly worse than fully concurrent methods. The exception is the heavy hitter case, where partitioned aggregation becomes a clear winner as a result of the performance gains from local preaggregation (which captures all the heavy hitter entries). However, this is a relatively extreme case of preaggregation effects.

Unique Key. In the pure insert workload, fully concurrent aggregation also exhibits superior performance at high thread count, but only when using atomic updates. Thread local aggregation scales reasonably well at lower thread counts, but performance significantly degrades as the number of threads increases. Therefore, implementations using thread local updates should note to limit the maximum thread count under certain workload conditions. In this dataset, since there is no contention on the underlying partial aggregations, using atomics creates very little overhead.

Comparisons. Notably, the performance advantage of fully concurrent aggregation against partitioned aggregation does not come from its scaling behavior, with partitioned aggregation achieving excellent scaling at high thread counts. However, fully concurrent aggregation has significantly lower hash table lookup overhead, since it only inserts into the ticketing hash table a single time (rather than performing aggregation locally and partition-wise).

In almost all cases, some form of fully concurrent aggregation achieves parity or outperforms partitioning. The only exception is with high cardinality and heavy hitters at high thread counts due to the poor performance of atomics in the presence of skew. However, our heavy hitter case is a particularly extreme case of skew with 50% the same value. More realistic Zipfian-distributed workloads are handled by atomic updates without too much issue.

Furthermore, the fact that thread local aggregation achieves its maximum speedup at a relatively low number of threads is advantageous in many real-world applications where multiple queries are executed at once (and thus not all system resources can be allocated to a single query). Practitioners should consider if using twice as many resources (on partitioning) for a small lift in throughput is worthwhile, or if those additional resources are better allocated to more concurrent queries. We imagine that this tradeoff is acceptable for some, but not all, systems.

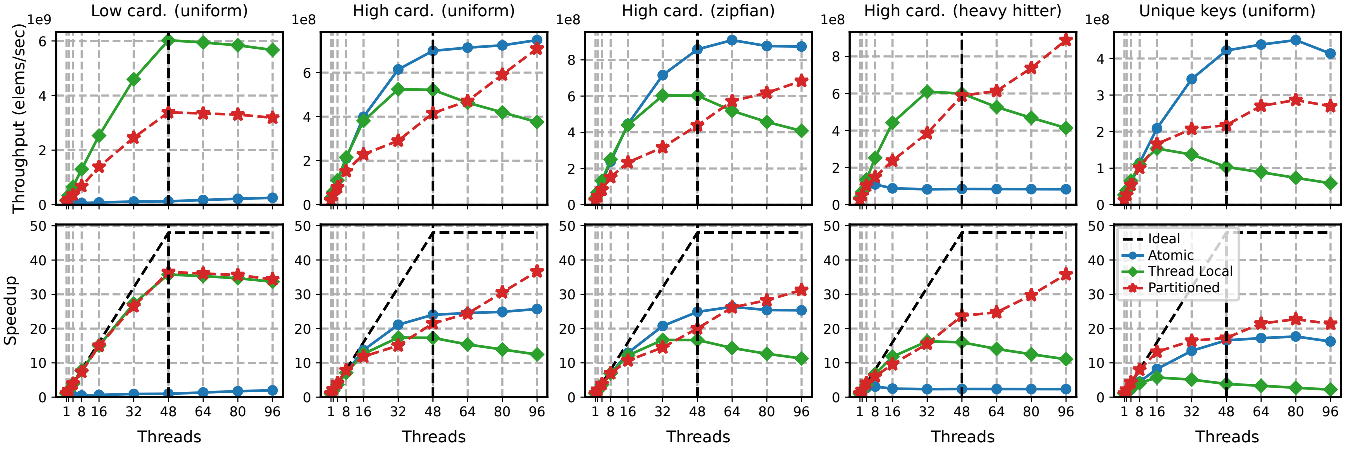

Figure 7: End-to-end evaluation of scaling behavior of fully concurrent aggregation methods vs. partitioned aggregation. Folklore* is used for the ticketing step for fully concurrent aggregation.

Recommendations. As we discussed in Section 3.2, ideally the method for partial aggregate updates should be chosen based on workload. Given that in almost all cases some form of fully concurrent aggregation achieves markedly superior performance over partitioning, a dynamic system could yield a significant speedup over current practice. However, perhaps one of the more important findings of this paper is that no single aggregation method (including partitioning) is desirable in all cases. That being said, thread local aggregation shows significant versatility. It has acceptable tail behavior and superior throughput in many workloads, especially those with lower cardinalities, which are by far the most common case in analytics workloads ([31] finds that 70% of aggregation queries have below a thousand unique keys and another 27% have below a million unique keys). Furthermore, its solid performance at low thread counts across all workloads makes it ideal for cases where resources are shared or limited. Therefore, if choosing only one aggregation method for a system, fully concurrent aggregation with thread local updates is a strong candidate. While we do not endorse one single aggregation method as the definitive choice for a database, this work on fully concurrent aggregation expands the toolkit available for designing a well-optimized query executor.

4.2 Explaining Behavior

In this section, we attempt to explain the scaling behavior of each aggregation method. We first take a high-level look by breaking down the time spent on each task (ticketing, update, etc.). Then, we analyze specific performance counters for a deeper understanding of how execution characteristics change with thread count.

Task Breakdown. First, we time each stage of the aggregation process and plot the proportion of time spent in Figure 8. The ticketing and update stages are as discussed in Section 3 and the preaggregation and partition-wise aggregation stages are as discussed in Section 2.2. The materialization stage consists of the work to turn results into a columnar format that can be pushed to the next query operator, including the cost of merging thread local partial aggregates and combining each partition's aggregated values.

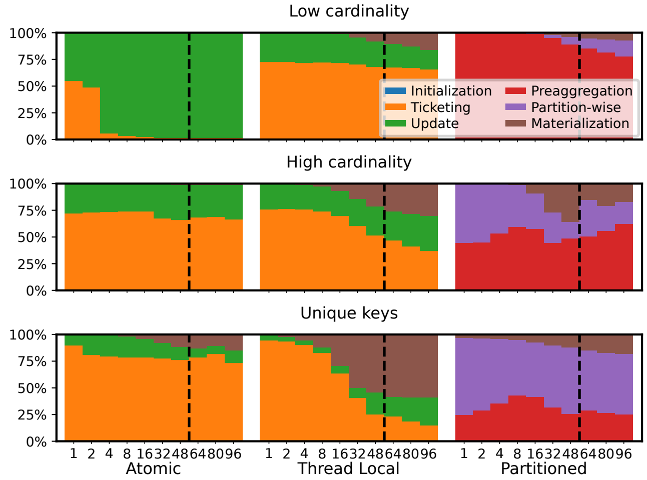

Figure 8: Percent of time spent on each step of aggregation.

We find that for fully concurrent aggregations, ticketing typically takes significantly more time than updating the partial aggregates. The exception is the low cardinality case for atomic updates, which at only 4 threads becomes the major performance bottleneck. Therefore, the choice of update step method should be driven more by tail performance. As long as it does not become a performance bottleneck, even relatively poor scaling in the update step is not insurmountable. This is the primary factor that allows thread local updates to maintain relatively good performance.

Further, we find that the thread local method's materialization cost becomes increasingly significant as thread count increases, which aligns with our experimental results and theoretical findings from Section 3.2. Materialization is fast enough in many cases that its poor scaling is not a bottleneck. However, at high thread counts with very high cardinalities, the materialization step can become the driving factor for degraded performance.

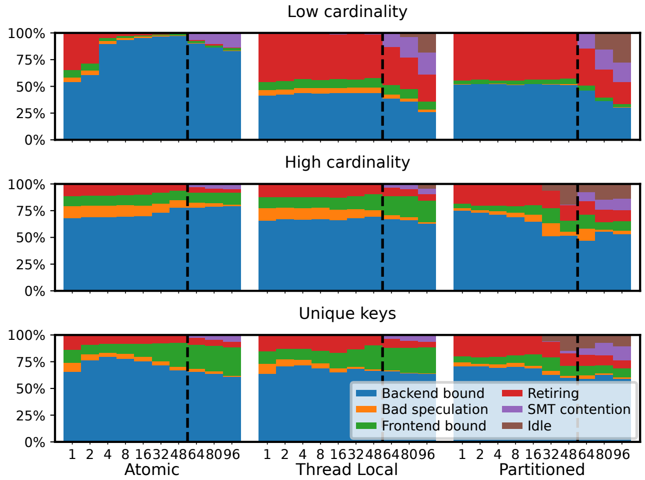

Figure 9: Breakdown of time spent on work by processor per workload and thread count. Shows both idle time and topdown analysis of time performing work.

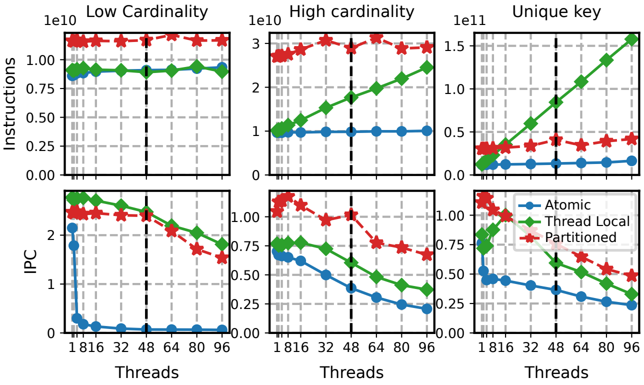

Figure 10: Instruction count and IPC vs. threads.

The materialization effects of partitioned aggregation are also particularly interesting. The materialization of each partition-wise hash table and copying of values to the final vector of all outputs is parallelized. The decrease in materialization time at very high thread counts appears to contribute to the improved scaling of partitioned aggregation even after thread count exceeds real core count. We attribute some of this odd behavior to the size of the hash table of partition-wise aggregates, which must be walked in order to materialize the key and value output columns. The size of each partition's hash table decreases with the number of partitions.

Performance Counters. We also gather performance counter data for each workload using Linux's `perf` tool to observe how execution characteristics vary with thread count. Analysis of the resulting data is made difficult by the complicated nature of the workload, whose performance is driven by multiple factors, including contention, time complexity scaling, etc. We therefore take a layered approach and separate the analysis into three questions:

- **Q1: How much time is spent performing "work."** Specifically, we measure the proportion of time a worker thread is scheduled on a core. The complement of this time is idle time, which occurs due to lock contention, synchronization effects, or lack of resources.
- **Q2: Of the time spent on "work," what are the performance characteristics?** These performance characteristics impact the rate that the processor performs work, that is instructions per cycle (IPC). We also employ topdown analysis [35] for granular insights into the bottlenecks.
- **Q3: How much "work" is there?** We measure the amount of work with total instruction count.

In Figure 9 we break down work time with topdown analysis, as well as plot the split of time working and idling (Q1, Q2). In Figure 10 we plot the instructions and IPC (Q2, Q3).

At a low cardinality, we see that atomic updates quickly becomes backend bound (contention from atomic loads is considered a backend bound effect, not idle time, unlike lock contention). In contrast, the thread local and partitioned methods show relative stability in its execution characteristics until idle time and SMT contention start to become a factor. In addition, a high proportion of instructions are retiring, indicating that execution is compute-bound and not bottlenecked by memory, speculation effects, etc.

At higher cardinalities, more work is spent backend bound across the board. This behavior is expected given that at high cardinalities, every hash table lookup and partial aggregate update is expected to be a cache miss. One interesting phenomenon is that partitioning actually becomes less backend bound as thread count increases in the high cardinality case, the opposite behavior of the fully concurrent approaches. Since memory bandwidth is less a bottleneck for partitioning in this workload, it is reasonable that we see continued scaling at high thread counts here.

Now, turning to the instruction and IPC counts in Figure 10, we confirm that the total work performed by thread local updates increases with thread count due to materialization. Also revealing is that while the base instruction count of partitioned aggregation stays stable, it is considerably higher than the other methods, owing to partitioning's added overhead. The counterbalancing effect is its relatively higher IPC at higher cardinalities. This behavior helps to explain why SMT benefits partitioned aggregation in particular. The high instruction count and IPC indicates that the processor is better able to utilize the time spent stalling on memory or other resources to continue execution on other compute-bound work.

4.3 Other Platforms

In addition to the AMD platform, we also run experiments on machines with ARM and Intel processors. The different hardware characteristics of each are described in Section 2.4. We graph the throughput achieved on these systems in Figure 11. We find that although the best technique for a given configuration may vary, the overall shape of the graphs remains similar to our results in Figure 7.

On ARM, we find that partitioned aggregation performs particularly poorly on the high cardinality case. A larger amount of time was spent on materialization, which combines all the thread local hash tables into two output columns of keys and values. This step is almost purely reads and writes to memory. Therefore, we believe

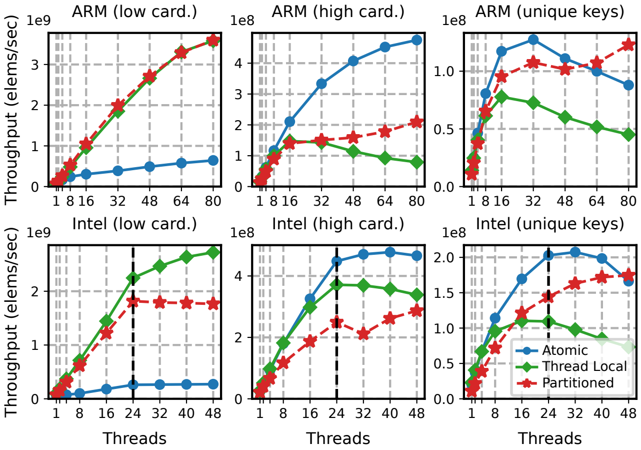

Figure 11: Scaling behavior of aggregation on ARM (Ampere Altra Q80-30) and Intel (Xeon Gold 5412U) machines. Extended benchmarks and profiling results on skewed datasets can be found at rm.cab/ghplots.

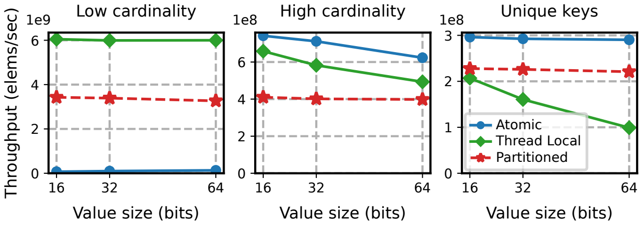

Figure 12: Throughput vs. size of value being aggregated.

that this poor performance could be related to the lower memory bandwidth of our ARM setup, which is 3200MT/s, compared to 4800MT/s on our AMD setup and 5600MT/s on our Intel setup.

Note that the Intel chip has fewer cores, so the graphs do not reveal severe performance degradation for thread local aggregation at high thread counts. Here, the performance of thread local updates is excellent in all but the unique case, and even then performs admirably. Atomic updates, in turn, perform much better than partitioned aggregation at higher cardinalities. These results reinforce our discussion that on more resource-constrained machines, fully concurrent aggregation is an especially convincing choice.

4.4 Tuple Size

Up until now, we assumed that all values were 64-bit integers. We now investigate the performance impact of the size of the tuple. Specifically, we hold the thread count (48) and key size (64-bit) constant and vary the value size to 16-bit, 32-bit, and 64-bit integers. We choose to vary the value size because it impacts the update step, whereas the key size mostly influences the speed of hashing. We plot throughput against value size in Figure 12.

We find that the effect on performance is most pronounced for higher cardinality workloads, which aligns with their backend bound behavior and thus sensitivity to factors such as memory bandwidth. The most significant impact is on thread local aggregation, whose throughput clearly decreases at higher tuple sizes, especially in the unique keys case. This behavior appears driven by the many sequential reads required when combining thread local results. The number of pages read from memory (and thus cache misses) during this operation scales with tuple size. On the other hand, atomic updates and partitioning have mostly random memory accesses, which incur a cache miss regardless of tuple size.

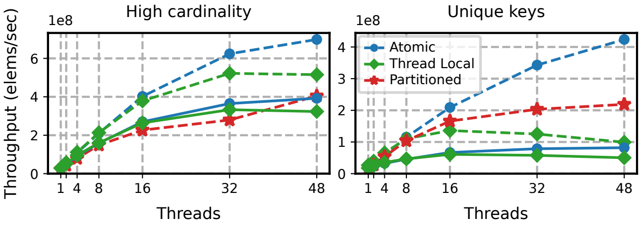

Figure 13: Scaling behavior of fully concurrent aggregation with resize. Dashed lines are without resizing and solid lines are with. Partitioned plotted for reference.

4.5 Resizing

A particularly thorny challenge with concurrent hash tables is how to resize them efficiently. In most cases, resizing requires all other threads to pause work to accommodate reallocation and migration. Although cardinality estimation can often allow a properly sized initial allocation [16], poor estimates are always a possibility.

To test the impact of hash table resizing, we adopt Maier et al.'s [23] method for contention-less fully concurrent migration of hash table entries for Folklore*. In this experiment, we set the capacity of the ticketing hash table and partial aggregate vectors to be half of the required capacity, forcing a resize.

Figure 13 shows that the fully concurrent approach suffers from significant performance degradation at higher thread counts in the presence of resizing. At 48 threads, atomics increase in latency by 1.8x and thread local by 1.6x. However, in the very worst case, unique keys, the degradation is much more extreme with a 5.2x latency increase for atomics and 2.0x for thread local. Thus, while not a show stopper, fully concurrent aggregation appears to be sensitive to the cost of resizing, especially in the most extreme cardinality cases. Future work on improving the performance of such resizes has significant headroom for improvement. Resizing performance should be an important dimension for analyzing the suitability of concurrent hash table designs for database applications, and we hope that this work motivates more research in the area.

4.6 Memory

Our model of execution assumes that all data structures fit within memory. It is not obvious how to adapt fully concurrent aggregation to disk spilling, unlike partitioning-based approaches [17]. Therefore, understanding the severity of memory usage can be relevant for the feasibility of aggregating over large datasets.

From Section 3.2, it is clear that thread local updates create significant memory overhead while atomic updates are space efficient. Analyzing partitioned aggregation is a much less straightforward

Table 4: Peak memory usage of different aggregation methods, measured in GB.

Threads	1			8			48		
Method	Atomic	Thread Local	Partitioned	Atomic	Thread Local	Partitioned	Atomic	Thread Local	Partitioned
Low card.	0.001	0.001	0.000	0.004	0.004	0.004	0.024	0.028	0.024
High card.	0.299	0.299	2.898	0.302	0.824	3.133	0.322	3.828	2.971
Unique keys	4.098	4.098	4.625	4.102	9.317	5.000	4.121	39.143	5.137

task since much depends on the spilling behavior. However, when assessing the worst case where most keys are spilled, the memory usage is upper-bounded by the total number of elements, which may be much higher than just the number of unique keys.

Comparing the peak memory usage of each technique in Table 4, we find that our theoretical model holds. Atomic aggregation performs best, while partitioned aggregation displays very high memory overhead for higher cardinality cases. Thread local aggregation exhibits surprisingly good memory usage characteristics, using much less memory than partitioned aggregation until the high end of thread count and cardinality. However, since we would limit the number of threads in practice, the memory usage would not reach the extreme case. Although memory considerations should be carefully considered on a system and workload basis, our results indicate that fully concurrent aggregation has substantial advantages over partitioning across many cases, although it is not obvious how to spill to disk.

5 RELATED WORK

The performance differences between concurrent aggregation using a shared hash table versus a partitioned approach using a local aggregation table have been studied extensively over the years [5, 22, 36]. Notably, prior work does not propose indirection to reduce hash table operations to a lookup and insert-only workload, instead performing updates within the hash table itself, which requires significant concurrency control. As a result, the consensus in the literature is that fully concurrent is not feasible due to contention costs in the presence of skew. One exception is [30], which does use indirection but in the context of FPGAs. Unlike our work, [30] performs updates in the hash table but prevents simultaneous access using a hardware cache (not adaptable to general-purpose CPUs) to synchronize all update requests on the same key.

Research from outside the database domain on concurrent hash table designs is also relevant to our work. In Section 3.1, we tested state-of-the-art hash table designs from or related to those described in prior work [14, 21, 23, 26]. However, as previously noted, even prior work that considers hash aggregation did not customize their tables to only support the required operations, instead focusing on general-purpose workloads. While the body of work on concurrent hash tables is informative, each design must be critically reevaluated in the context of our specialized use case to make a determination about their performance for aggregations.

In addition to the use of shared hash tables, many other aggregation methods have been proposed. We compared our work most directly to partitioned methods, which have been successfully integrated into many real-world systems [17, 19, 20, 29]. Various hybrid approaches that leverage local aggregations for heavy hitters and a global shared table for other values have been proposed in order to balance performance and memory concerns [5, 10]. Ideas from these works may be beneficial for resolving the challenges we found with our fully concurrent aggregation at high thread counts and cardinalities. Sort-based aggregations, which are exceptionally efficient when data is already sorted [18], have also been extensively evaluated [12, 32, 33]. Müller et al. [24] note that when aggregating keys larger than CPU registers, sorting and hashing are computationally similar. In the Google F1 system, where distinct value counts are especially difficult to estimate, Do et al. [8] showed that specialized sorting techniques could be more robust than hash-based aggregation. Sort-based aggregations also have operational advantages in out-of-core databases, as an external sort is generally simpler than external hashing [8, 33].

While this work focused on morsel-driven systems, other execution models, such as operator-level parallelism [13, 19] and pipeline parallelism [37], are also popular. It seems plausible that different aggregation methods may be optimal under different execution models, and we leave such investigations to future work.

Finally, a closely-related operation to hash aggregations is hash joins. Similar debates about partitioning have occurred in the literature [1], and specialized hash tables (multi-maps) tend to have large performance benefits [3, 20]. This body of work has had a similar overarching takeaway that specialized hash tables are crucial to achieving good performance, but their methods are generally incompatible with the needs of fully concurrent aggregation (e.g., multi-maps, probing).

6 CONCLUSIONS AND FUTURE WORK

The go-to group by aggregation technique in modern analytic DBMSes is partitioning-based. In this work, we made the case for a simpler approach using a global concurrent hash table. For this simpler approach to match the performance of purpose-built partitioning techniques, a purpose-built hash table is required. We explored the operational benefits and drawbacks of both approaches and offered guidance to implementers along the way.

In future work, we plan to investigate more complex hash table designs as well as hybrid aggregation strategies (e.g., with teams of threads sharing an aggregation table to reduce contention). Another promising direction for investigation is understanding how new hardware features–like vectorized atomic operations–could be used to accelerate group aggregation even further. Additionally, the fully concurrent approach we analyze here suffers from issues with resizing and spilling, which are ripe areas for future work. Finally, this work only examined hash aggregation in a morsel-driven execution model: investigations into sort-based techniques, along with other execution models, could also yield interesting results.

REFERENCES

[1] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, Virtual Event China, 168–180. https://doi.org/10.1145/3448016.3452831

[2] Michael A. Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Iceberg Hashing: Optimizing Many Hash-Table Criteria at Once. *J. ACM* 70, 6 (Dec. 2023), 1–51. https://doi.org/10.1145/3625817

[3] Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2024. Simple, Efficient, and Robust Hash Tables for Join Processing. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*. ACM, Santiago, Chile, 1–9. https://doi.org/10.1145/3662010.3663442

[4] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Conference on Innovative Data Systems Research*. www.cidrdb.org, Asilomar, CA, USA, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[5] John Cieslewicz and Kenneth A. Ross. 2007. Adaptive aggregation on chip multiprocessors. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, Vienna, Austria, 339–350.

[6] Rob Clucas. 2025. Leapfrog. Retrieved 2025-09-01 from https://github.com/robclu/leapfrog

[7] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2010. Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, Carmona, Spain, 185–192. https://doi.org/10.1109/isorc.2010.10

[8] Thanh Do, Goetz Graefe, and Jeffrey Naughton. 2022. Efficient Sorting, Duplicate Removal, Grouping, and Aggregation. *ACM Transactions on Database Systems* 47, 4 (Dec. 2022), 1–35. https://doi.org/10.1145/3568027

[9] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. ACM, Amsterdam, Netherlands, 1–3. https://doi.org/10.1145/3329785.3329918

[10] Philipp Fent and Thomas Neumann. 2021. A practical approach to groupjoin and nested aggregates. *Proceedings of the VLDB Endowment* 14, 11 (July 2021), 2383–2396. https://doi.org/10.14778/3476249.3476288 Publisher: Association for Computing Machinery (ACM).

[11] Kevin P. Gaffney and Jignesh M. Patel. 2024. Is Perfect Hashing Practical for OLAP Systems?. In *Conference on Innovative Data Systems Research*. www.cidrdb.org, Chaminade, HI, USA. https://www.cidrdb.org/cidr2024/papers/p65-gaffney.pdf

[12] G. Graefe, A. Linville, and L.D. Shapiro. 1994. Sort vs. hash revisited. *IEEE Transactions on Knowledge and Data Engineering* 6, 6 (Dec. 1994), 934–944. https://doi.org/10.1109/69.334883

[13] G. Graefe and W.J. McKenna. 1993. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. IEEE Comput. Soc. Press, Vienna, Austria, 209–218. https://doi.org/10.1109/icde.1993.344061

[14] Robert Kelly, Barak A. Pearlmutter, and Phil Maguire. 2020. Lock-Free Hopscotch Hashing. In *Symposium on Algorithmic Principles of Computer Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 45–59. https://doi.org/10.1137/1.9781611976021.4

[15] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment* 11, 13 (Sept. 2018), 2209–2222. https://doi.org/10.14778/3275366.3275370 Publisher: VLDB Endowment.

[16] Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating filtered group-by queries is hard: Deep learning to the rescue. In *International Workshop on Applied AI for Database Systems and Applications*. VLDB Endowment, Los Angeles, CA, USA.

[17] Laurens Kuiper, Peter Boncz, and Hannes Mühleisen. 2024. Robust External Hash Aggregation in the Solid State Age. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, Utrecht, Netherlands, 3753–3766. https://doi.org/10.1109/icde60146.2024.00288

[18] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment* 5, 12 (Aug. 2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[19] Andrew Lamb, Daniël Heres, and Raphael Taylor-Davies. 2023. Aggregating Millions of Groups Fast in Apache Arrow DataFusion 28.0.0. https://arrow.apache.org/blog/2023/08/05/datafusion_fast_grouping/

[20] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, Snowbird, Utah, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[21] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent Cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, Amsterdam, Netherlands, 1–14. https://doi.org/10.1145/2592798.2592820

[22] Hua Luan and Lei Chang. 2022. An experimental study of group-by and aggregation on CPU-GPU processors. *Journal of Engineering and Applied Science* 69, 1 (Dec. 2022). https://doi.org/10.1186/s44147-022-00108-1 Publisher: Springer Science and Business Media LLC.

[23] Tobias Maier, Peter Sanders, and Roman Dementiev. 2018. Concurrent Hash Tables: Fast and General(?)! *ACM Transactions on Parallel Computing* 5, 4 (Dec. 2018), 1–32. https://doi.org/10.1145/3309206 Publisher: Association for Computing Machinery (ACM).

[24] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, Melbourne, Victoria, Australia, 1123–1136. https://doi.org/10.1145/2723372.2747644

[25] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*. VLDB Endowment, Seoul, Korea, 1049–1058.

[26] Prashant Pandey, Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, Guido Tagliavini, and Rob Johnson. 2023. IcebergHT: High Performance Hash Tables Through Stability and Low Associativity. *Proc. ACM Manag. Data* 1, 1 (May 2023), 1–26. https://doi.org/10.1145/3588727 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[27] Jeff Preshing. 2016. Leapfrog Probing. Retrieved 2025-09-01 from https://preshing.com/20160314/leapfrog-probing

[28] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam, Netherlands, 1981–1984. https://doi.org/10.1145/3299869.3320212

[29] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU acceleration: so much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (Aug. 2013), 1080–1091. https://doi.org/10.14778/2536222.2536233

[30] Bashar Romanous, Skyler Windh, Ildar Absalyamov, Prerna Budhkar, Robert Halstead, Walid Najjar, and Vassilis Tsotras. 2021. Efficient local locking for massively multithreaded in-memory hash-based operators. *The VLDB Journal* 30, 3 (May 2021), 333–359. https://doi.org/10.1007/s00778-020-00642-5 Publisher: Springer Science and Business Media LLC.

[31] Jan Vincent Szlang, Sebastian Bress, Sebastian Cattes, Jonathan Dees, Florian Funke, Max Heimel, Michel Oleynik, Ismail Oukid, and Tobias Maltenberger. 2025. Workload Insights from the Snowflake Data Cloud: What Do Production Analytic Queries Really Look Like? *Proceedings of the VLDB Endowment* 18, 12 (Aug. 2025), 5126–5138. https://doi.org/10.14778/3750601.3750632

[32] Gaurav Vaghasiya and Shiva Jahangiri. 2024. [Experiments & Analysis] Hash-Based vs. Sort-Based Group-By-Aggregate: A Focused Empirical Study [Extended Version]. https://doi.org/10.48550/arXiv.2411.13245 arXiv:2411.13245 [cs].

[33] Gaurav Vaghasiya and Shiva Jahangiri. 2024. A Hybrid Approach to Group-By and Aggregation Query Execution. In *2024 IEEE International Conference on Big Data (BigData)*. IEEE, Washington, DC, USA, 3799–3808. https://doi.org/10.1109/BigData62323.2024.10825803

[34] Joel Wejdenstål. 2025. DashMap. Retrieved 2025-09-01 from https://github.com/xacrimon/dashmap

[35] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, CA, USA, 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

[36] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. 2011. Scalable Aggregation on Multicore Processors. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. ACM, Athens, Greece, 1–9. https://doi.org/10.1145/1995441.1995442

[37] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Boston, MA, 10. https://doi.org/10.5555/1863103.1863113