Locality-Adaptive Parallel Hash Joins using Hardware Transactional Memory

Anil Shanbhag, Holger Pirk, and Sam Madden

MIT CSAIL, Cambridge, USA
`{anil,holger,madden}@csail.mit.edu`

Abstract. Previous work [1] has claimed that the best performing implementation of in-memory hash joins is based on (radix-)partitioning of the build-side input. Indeed, despite the overhead of partitioning, the benefits from increased cache-locality and synchronization free parallelism in the build-phase outweigh the costs when the input data is randomly ordered. However, many datasets already exhibit significant spatial locality (i.e., non-randomness) due to the way data items enter the database: through periodic ETL or trickle loaded in the form of transactions. In such cases, the first benefit of partitioning — increased locality — is largely irrelevant. In this paper, we demonstrate how hardware transactional memory (HTM) can render the other benefit, freedom from synchronization, irrelevant as well.

Specifically, using careful analysis and engineering, we develop an adaptive hash join implementation that outperforms parallel radix-partitioned hash joins as well as sort-merge joins on data with high spatial locality. In addition, we show how, through lightweight (less than 1% overhead) runtime monitoring of the transaction abort rate, our implementation can detect inputs with low spatial locality and dynamically fall back to radix-partitioning of the build-side input. The result is a hash join implementation that is more than 3 times faster than the state-of-the-art on high-locality data and never more than 1% slower.

1 Introduction

As the clock rate of processor cores has stagnated, parallelization has become the primary means to saturate the increasing memory bandwidth of modern computers. Fortunately, in the field of data management, many problems have efficient data-parallel solutions. In particular analytical queries can often saturate the bandwidth using horizontal partitioning of the input and parallelized computation by different cores on each partition, followed by merging of results.

This approach works particularly well, when the result is small and the merge trivial, as in operations such as grouped aggregation with few groups.

When the result is large, as, e.g., in the case of hash joins the balance shifts: the overhead of partitioning and/or merging may well become the most expensive step. One way to avoid this overhead is to update the hash table in-place. Doing so requires the use of locks or atomic instructions during inserts, which, unfortunately, also introduces significant overhead. Which of these two approaches is
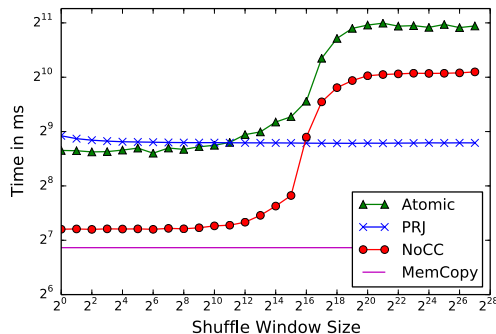
Fig. 1: Sharing vs. Partitioning in Hash-Building

better is not immediately evident and depends on spatial locality in the underlying data.

Such locality may stem from many real-world effects such as, a) periodic bulk updates (common in data warehouses) which create locality in date attributes or an attribute that is correlated with the load order (e.g., physical location), b) trickle loading through transactions in OLTP systems which also creates locality in date columns, c) automatically assigned IDs which are often monotonically increasing counters or d) the occurence of temporarily hot items (e.g., star wars action figures before movie releases). In fact, spatial locality, is one of the most commonly exploited aspects of code and data in computer architecture design.

To illustrate the relative costs of these two approaches depending on the degree of spatial locality, we compared a state-of-the-art, hand-optimized radix-based hash-building phase [1] to an implementation using a shared hash table with linear probing, that is protected using atomic instructions (both implementations use identity hashing). To isolate the effects of insert locality, the input data keys are unique (from 1 to 2^{27}) but (knuth-)shuffled within a sliding window (the authors of [1] study the case of fully shuffled data, i.e., window size equal to input data size). By varying the size of the window, we can study the sensitivity to locality in the input data. Figure 1 shows that for small shuffle windows, the shared hash table approach using atomic instructions ("Atomic") performs slightly better than the radix-based partitioning approach ("PRJ"). For larger shuffle windows, we observe the typical effects of poor cache locality: a staircase like pattern that exposes the sizes of the CPU's caches. The (radix-)partitioned implementation is robust against poor locality because the partitioning creates locality using a hardware-conscious implementation.

However, neither implementation performs close to the memory bandwidth, which gives hope for an even faster implementation. To illustrate this, we implemented a third variant: insert values into the shared hash table without protecting the buckets ("NoCC"). Since the values are unique and there are no collisions, this implementation yields the correct result in this case (although,

of course, this would not be true in general). The graph shows that this implementation outperforms the others by more than a factor 3. NoCC comes very close to the time taken by `memcopy` to write the same amount of data. While this illustrates the overhead of the existing approaches, unfortunately, it is not easily generalizable to cases in which there are conflicts.

Fortunately, there is a third technique that can achieve correctness and performance close to the "NoCC" approach by exploiting the hardware transactional memory (HTM) features in modern (Intel) processors (and, hopefully, AMD soon). In this paper, we explore the potential of this (relatively new) technology for the purpose of parallel hash-joins. In particular:

- We extensively study the effect of locality and transaction size on the performance of HTM-protected parallel hash-building.
- We devise a hash-building technique that dynamically balances the per-transaction overhead and the abort rate that comes with larger transactions.
- We use the number of aborted transactions as an indicator of poor locality allowing us to adaptively fall back to the partitioning implementation [1] when appropriate.

This results in a hash join implementation that outperforms existing ones by about a factor 3 when spatial locality in the input is high and never performs significantly (i.e., more than 1 percent) worse than the state of the art.

We structured the rest of this paper as follows: in Section 2, we discuss primitives to ensure memory consistency. In Section 3, we present the current state of the art in parallel hash joins. In Section 4, we develop our approach to parallel hash joins by step-by-step analyzing and addressing the relevant bottlenecks. This section also contains our evaluation. We conclude in Section 5.

2   Synchronization Primitives

Building a hash table in parallel involves concurrent inserts into to the table. In this section, we briefly review the different means used to ensure correct results: transactional memory, atomic instructions and input partitioning.

2.1   Transactional Memory

The key idea of a transaction is that multiple operations can be combined into a unit that is executed atomically and in isolation from others. Multiple transactions can be executed concurrently as long as they affect different objects - otherwise one of the transactions fails. Transactional memory is the application of this concept to the reading and writing of a system's memory, thus providing an alternative to fine-grained locking which is more expensive and prone to deadlocks.

The concept of Transactional Memory existed for a long time [4]. Shavit and Touitou [12] are credited for the first Software Transactional Memory(STM)

proposal. Due to entirely software controlled validation overhead, STM causes significant slowdown during execution and found little resonance in the database systems community. Only recently hardware vendors, such as Intel [14] and IBM [5] realized transactional memory support in hardware (HTM).

While IBM relies on dedicated HTM components, Intel extended the CPU's cache-coherence infrastructure, based on the well known MESI protocol [3], to enable HTM. In MESI, each cache line can be in one of the four states: modified, exclusive, shared or invalid. Different caches are kept in-sync by snooping each other's load and store requests. For example, when a core updates a cache line which is also present on other caches in shared state, the local state is updated to modified and other caches update their state for the cache line to invalid. The key idea used to implement HTM in Haswell is to use the L1 cache as a buffer for executing transactions. All the updates made by the transaction happen locally in the L1 cache and the changes are propagated to main memory only if the transaction successfully commits. Since the cache coherence protocol is an integral part of multicore CPUs and commits/aborts require no extra communication across cores, the transactional execution has very little overhead.

The downside of using HTM in Haswell is that the transaction size is limited to size of L1 cache. Even though this limitation imposes constraints on the type of transactions that can be executed, it has caught the attention of database researchers due to its low overhead. The HTM support has been used to implement database transactions in in-memory databases [7, 13]. To the best of our knowledge, ours is the first work that explores using HTM for adaptive parallel hash-building in hash joins.

2.2   Atomic Instructions

Another means to provide mutual exclusion without using a lock is using atomic instructions. Atomic instructions allow the programmer to concurrently access and update variables of basic data types. Like HTM, atomic instructions rely on the MESI cache coherence protocol to provide atomicity and isolation. When a core wishes to read a variable, the corresponding cache line is loaded in shared mode. On a store request, the core sets the cache line to modified and the cache line gets invalidated on the other cores. A more expensive operation like compare-and-swap requires loading the cache line as exclusive before comparing and swapping.

Spinlocks are implemented using atomic instructions and require one atomic operation to set the lock variable and one store to update the variable. Since join columns are often integers or dictionary-encoded strings, atomic instructions can be used to directly update the hash table entries, making it much faster than using spinlocks in low-contention scenarios.

2.3   Partitioning

Consistency primitives are needed in the event of concurrent data access to the same address. An alternative approach is to partition the input such that no

such conflicts arise. While partitioning itself can be expensive, there is no need for synchronization in the processing phase. In particular for the construction of hashes, this technique has been applied to great effect.

3   State-of-the-Art Hash-Building

One of the fundamental differences when comparing a database's hash-table requirements to those of generic hash-tables implementations stems from the bulk-processing nature of database query processing: where generic hash-tables have to guarantee consistent reads after every insert, database query processing usually only requires consistency at the end of the build process. For that reason, few databases incorporate off-the-shelf hash-table implementations.

Consequently, hash building has been extensively studied in the database literature in the context of hash joins. Two main lines of thought exist: the first argues that the best performance can be achieved by using a hardware-conscious (radix) partitioning to minimize cache misses during the build-phase [6]. The second approach holds that a hash-build implementation can be efficient by using a single shared hash table across threads and synchronizing using locks [2]. Through careful evaluation, Balkesen et al. [1] showed that the radix partitioning approach performs significantly better than the shared hash table approach for fully shuffled data. Since this approach is the current state-of-the-art (and also our prime competitor), we discuss it in more detail in the following.

3.1   Radix Partitioned Hash Joins

The main insight motivating radix partitioning based hash-building is that when the hash table is larger than the cache size, almost every insert into the hash table causes a cache miss. This can be avoided by pre-partitioning the data using an approximation of the hash-function and building a hash-table per partition, thus improving insert locality. Since the partitions are filled sequentially, the memory access locality in the partitioning phase is improved. Manegold et al. [9] noted that since each partition ends up residing on a different memory page, having a very high fanout ($= inputSize/L1Size$) results in excessive TLB thrashing. To circumvent this, the input data is partitioned in multiple passes (two is usually sufficient). Each pass has a fanout less than or equal to the number of TLB entries. Each pass looks at a different set of bits from the hashed value, hence the name *Radix Partitioning*.

In addition to its cache-friendliness, the radix partitioning approach is easily parallelized. The input relation is divided into horizontal partitions. In the first pass, each part is scanned independently to generate a histogram over the input data, so that the exact output size is known per thread per partition. A single contiguous output array is allocated. A synchronization barrier is used to indicate the end of first pass, at which point each thread computes the prefix-sum over the relevant histograms to find the exact offset of each partition it writes to. Finally, the threads execute a second pass over its partition of the data to write the

tuples to the right place in the output array, without any synchronization. We end up having two passes over data per radix partition pass. For our evaluation dataset, we require two radix partitioning passes i.e. four passes in total.

3.2   Using Atomics

To access the suitability of atomic instructions for parallel hash-building, we implemented a shared hash table using the built-in C++11 atomics. The implementation performs linear probing on insert and a single catch-all overflow bucket that is used after checking a configurable number of slots. Note that, while it is possible to implement a lock-free hash-table with bucket chaining, the build-time overhead (most importantly creating an intermediary copy of a bucket) is much higher than the catch-all scheme we implemented. The implementation is mostly straight forward with a single optimization: we first use an atomic read to ensure the slot is empty, following which a compare-and-swap is used to insert the tuple. The swap might fail if the slot has been filled by another thread since the read was performed. We found that the benefits of cheap checking to ensure a slot is free outweighs the cost for occasionally failing to insert.

3.3   Hash-Function Design

Naturally, the selection of an appropriate hash-function is crucial when designing a hash-table. The desirable property of uniform output distribution has to be balanced against locality preservation and computational efficiency. The design space ranges from cryptographic hash functions [11] that have near-perfect output through efficient hashes such as Murmur Hashing[1] to simple modulo hashing. Different in-memory database systems took different decisions to address this question: while HyPeR and Pivotal Gemfire/Apache Geode aim for skew-resilience using Murmur Hashing, MonetDB implements the locality-preserving and cheap modulo hashing. In this paper, we limit ourselves to modulo hashing. We leave the development of a hybrid hash-function (preserving cache-line locality while mitigating global skew) to future work.

4   Exploiting Locality using HTM

The hypothesis we want to establish and substantiate in this section is that is is possible to develop a single pass hash table implementation that, given enough locality in the input data, becomes (close-to) memory bandwidth bound. While the ultimate goal of our efforts is an adaptive hash-join, we construct it "bottom up": by analyzing the impact of the relevant hardware- and data-characteristics. We analyze the effects and costs of Virtual Memory (§4.2), Atomic Instructions (§4.3), HTM overhead (§4.4) and the impact of locality (§4.5). Only after establishing the importance of these factors, do we develop our final contribution

---

[1] `https://sites.google.com/site/murmurhash`

(§4.6 to §4.8): a hash-join implementation that adapts to the degree of locality in the underlying data, using an HTM-based approach for high-locality and the state-of-the-art partition-based approach [1] for low-locality data.

The key ingredient of our approach is the use of Intel's Restricted Transactional Memory (RTM). Before diving into the experiments and implementation, however, let us briefly discuss our experimentation setup as well as the data-structure we use for the hash-table.

4.1   Setup

To ensure comparability with previous work, we adopted the same workload as previous work [1, 2]: unique keys (32 bit) carrying a 32 bit payload with every tuple finding exactly one join-partner (Appendix B contains the results for non-unique keys including conflict handling). Note that, while there is currently some discussion about the prevalence of this specific case in practical applications, it constitutes a harder challenge than the case of larger payloads in which copy overhead dominates the costs. We also feel that, in particular in highly-optimized, column-oriented databases, small payloads are the rule, rather than the exception.

The parameter we are interested in is the locality of the input: where previous work applied a global (knuth-)shuffle to the input, we slide window through the array and shuffle one value in the window per slide step - the size of the window is the primary parameter of our experiments. This allows us to create data locality ranging from fully sorted (window size 1) to fully shuffled (window size equal to input size).

The structure of our hash-table is similar to that of bucket-chaining based hash table implementations [1, 2]: insert conflicts are handled using buckets of size three that are chained in a linked list on overflow. Later in Section 4.6 we describe the full design with pseudocode. While bucket chaining is vulnerable to inputs with a *very* high number of conflicts, we found that performance for (non-unique) uniform random data is very similar to that of unique, shuffled data (see Appendix B)

The experiments were run on a single socket Intel E3-1270 v5 @ 3.60GHz (Skylake with 4 Cores, 8 hardware threads) fully equipped with 64GB DDR-4 RAM (2133 Mhz bus clock) running Ubuntu Linux 15.10 (Kernel 4.2.0-30). The input sizes were 134 million tuples (2^{27}) on each side, fitting comfortably in main-memory. All experiments were compiled using gcc 5.2.1 using the "`-O3 -march=native`" flags and we report the average of 5 runs.

An interesting aspect to study, would be the interplay of HTM and NUMA. Unfortunately, multi-socket CPUs that reliably[2] implement RTM only became available in early 2016 - too late to be included in this study. We did, however, find a number of relevant, hitherto undocumented, effects that we describe in the following.

---

[2] Earlier implementations suffered from a bug that caused Intel to deactivate the feature in a microcode update

4.2   Interaction with Virtual Memory

The first effect we noticed when developing our approach is the fragility of Intel's transactional memory implementation with respect to the events that cause transactions to abort. The hazardous effects of, e.g., the size of the working set of the transaction or evictions due to associativity are well documented [14] and are of little importance to us. However, in our experiments, we noticed an effect that has substantial impact on our design: when accessing unmapped virtual memory, transactions fail with no chance of success upon retry but without indicating so using the respective status flags. This is due to the intricate interplay of restricted transactional memory and lazy physical page allocation. When allocating zeroed-out memory using `calloc`, the Linux Kernel does not eagerly allocate the memory and run a loop to initialize it. Instead, it maps all allocated pages to a single, read-only page that is statically initialized to zero. Any write to that page causes a page fault and subsequent copying of the read-only page (copy-on-write). If the write is protected by a transaction, however, the page fault immediately aborts the transaction without triggering the page fault. Consequently, the read-only page is never copied and a retry of the transaction will fail. This problem percolates to subsequent transactions on the page causing all transactions to fail.

This effect has been reported in the context of updates to tree-indices [8] and resolved by initializing pages using an atomic instruction before retrying a transaction. In the case of hash-building, an alternative is to eagerly pre-fault memory which avoids complex conflict handlers in the critical path. Unfortunately this prevents some optimizations that are often applied to the process of hash-building. Generously over-allocating the hash table is, e.g., an effective to means deal with the unknown domain of input values. This optimization is no longer feasible when the entire table has to be pre-faulted. We're considering the extension of our work to cases with unknown domains for future work.

4.3   Hardware Transactions vs. Atomics

To establish a baseline for the usefulness of hardware transactions, we started by comparing them to their most direct competitor: atomic instructions. Note that this comparison is not entirely apples to apples because Intel's restricted transactions are not guaranteed to succeed. However, we will show in the rest of this section that, given enough locality, restricted transactions virtually always succeed (fewer than one abort in 10,000 transactions) when protecting every insert with its own transaction (denominated TS=1).

In Figure 2, we compare the cost of the two techniques in the extreme cases: building a hash table using identity hashing on fully sorted or fully shuffled data. The figure illustrates that protecting an insert using a hardware transaction is, in fact, cheaper than using atomic instructions in both cases. This is to be expected because Compare-and-Swap (used in atomic insert) is a more expensive operation compared to an optimistic load and store. The reason for this lies in the fact that the CPU has to guarantee that the write was actually performed. For
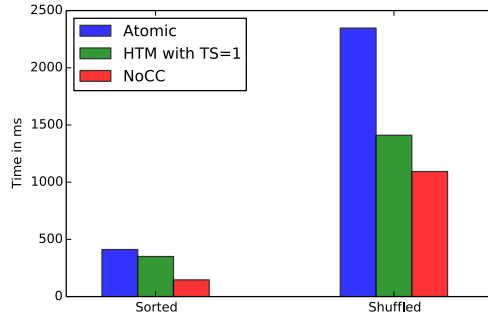
Fig. 2: Transactional Memory vs. Atomic Instructions

that, it has to MESI-invalidate the cache line in all other cores and (potentially) write it back to memory. This makes little difference in the sorted case (since virtually no cache-lines are shared) but is substantial in the shuffled case because cache-lines are frequently shared. However, using HTM is still significantly more expensive than processing without concurrency control: around 5 times in the sorted case and 20% on shuffled data. Consequently, we, turn our attention to means to reduce the per-transaction cost next.

4.4 Transaction Overhead

(a) Sorted Data   (b) Fully Shuffled Data
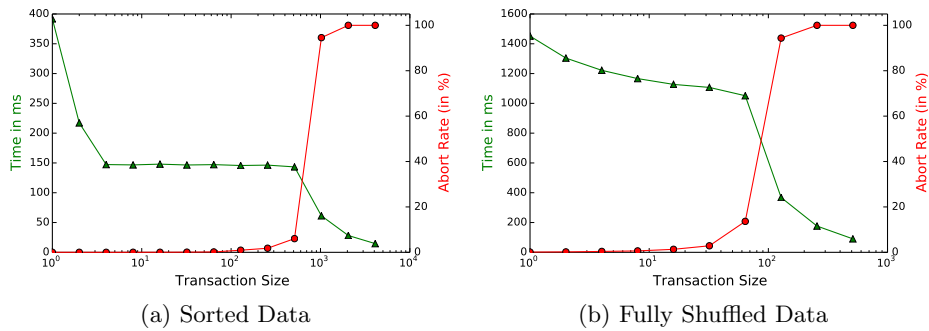
Fig. 3: Assessing Per Transaction Overhead

It is natural to expect some overhead for setting up a transaction. This overhead is, of course, relative to the cost for other operations. As established in Figure 1 on page 2, locality in the input data, which translates into access locality for the hash-buckets, is arguably the determining factor for overall performance. Consequently, we assessed the overhead to protect hash-inserts by transactions

for the two extreme input data distributions: sorted (optimal locality) and fully shuffled (least locality[3]), while varying the number of inserts per transaction. The per-transaction overhead can, thus, be amortized over multiple inserts. The results (Figure 3a and Figure 3b, respectively) show that, in the presence of locality, the overhead of setting up a transaction for every insert is almost $4x$[4] and becomes apparent in the steep drop left of the plateau in Figure 3a. The picture changes when considering fully shuffled data: while the overhead is still significant, it is no longer the dominating cost factor (cache misses are).

However, larger transactions increase the chance of transaction aborts even if there are no actual conflicts at CPU word granularity: factors such as suboptimal cache associativity and context switching which lead to L1 cache evictions lead to aborts. This effect can even be observed in the sorted data case (Figure 3a): the abort rate starts to increase notably at around 64 inserts per transaction. When the active set size grows beyond the size of the L1 cache, all transactions abort - as expected.

For fully shuffled data (Figure 3b), the abort rate increases much earlier as the inserts are spread out over many cache lines, which amplifies the active set size and increases the probability of false conflicts. This naturally raises the question of the impact of locality on performance and abort rate, which we study in the following.

4.5   Impact of Locality

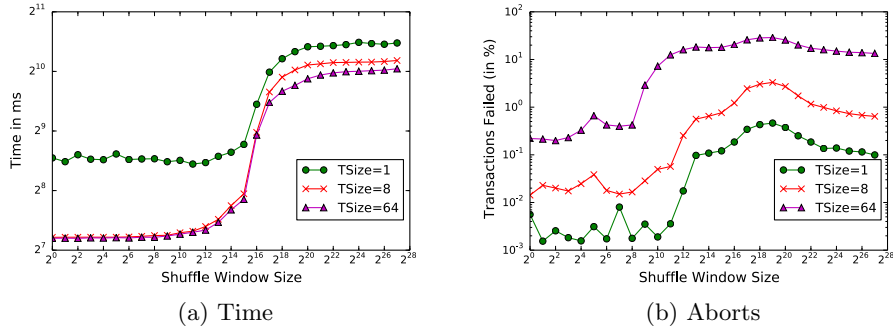

(a) Time   (b) Aborts

Fig. 4: Varied Size And Shuffle Window

To assess the impact of locality on the build performance and abort rate we applied the same shuffle that was used to create Figure 1 to the input data and evaluated the HTM-based implementation. In addition to the shuffle window

---

[3] Note that fully shuffled unique data items have even worse locality than uniform uniform randomly generated (non-unique) data items because there is zero probability for re-accessing a data item

[4] the ratio is even higher for smaller datatypes

size, we varied the size of the transactions and measured build time (Figure 4a) and abort rate (Figure 4b). Figure 4a re-iterates the point that, given enough locality, larger transactions perform better. The point of the (inevitable) cost explosion, however, is only slightly influenced by the transaction size.

Figure 4b, on the other hand, shows that the abort rate is strongly influenced by the size of the transactions (as expected): transactions of size 1 rarely fail when locality is high (left side of the chart) while larger transactions have a significantly higher chance of failing. With less locality (right side of the chart), the abort rate for all transaction sizes increases up to around two orders of magnitude.

Note that these experiments did not include the retrying of transactions and would, thus, not guarantee that values to actually get inserted into the hash table. We will discuss an implementation without that shortcoming in the following.

4.6   Putting It Together

The last step to providing a full hash-build implementation is to develop a strategy to deal with aborted transactions. For that purpose, we simply record the input position range associated with the aborted transactions in a preallocated buffer while building the hash table. We also record the tuples that landed in full buckets in an overflow buffer. When the build-phase is complete, we perform a wrap-up phase that traverses the abort- and overflow-buffers, resolves the positions and inserts the values into the hash table. Note that the bucket chaining happens only in the wrap-up phase, we do not do bucket chaining in the transactions to keep their cache line footprint small. The pseudocode of the full implementation is given in Figure 5.

We found that it is not worth parallelizing the wrapup phase due to its unique characteristics: for high-locality data, its cost are insignificant relative to the overall runtime because there are few failed transactions. For low-locality data, the cost of the wrap-up phase is dominated by cache- and TLB-thrashing, leaving even a single CPU core mostly idle.

With the wrap-up in place, the hash-build implementation is complete. As established, however, optimal performance hinges on the appropriate selection of the transaction size. To remove transaction size as a tuning parameter we adopt a simple adaptation strategy: we start with a transaction size of 16, monitor the abort rate and define a high- and a low-watermark. When the abort rate exceeds the high-watermark we half the transaction size, when it drops below the low-watermark we double it. We found 0.4% to be a good low and 2% a good high watermark and check every 16×1024 inserts.

Figure 6 shows the total runtime of the static transaction sizes as well as our adaptive approach (TSize-Adaptive). Note how the adaptive approach matches the performance of the best static case and occasionally outperforms them, selecting optimal parameters between the static values.

We experimented with varying number of threads and noticed that the performance does not improve after 3 threads, indicating that the application is

```
struct Bucket:
  Tuple tuples[3]
  int count
  int nextBucketIndex

HashTable table
table.buckets = Bucket[ceil(numTuples/3)]

// Each thread gets an input range [start,end)
for (i = start; i<end; i += transactionSize):
  status = _xbegin()
  if status == _XBEGIN_STARTED:
    for (j = i; j < i + transactionSize; j++):
      slot = hash(tuple[j].key)
      for (k = slot; k < slot + probeLength; k++):
        if table.buckets[k] is not full:
          table.buckets[k].add(tuple); break
      if not inserted:
        overflow.add(tuple)
    xend()
  else: // Transaction Failed
    failedTransactionRanges.add(i)

// Wrap-Up
for (i in failedTransactionRanges)
  // Insert tuples[i] to tuples[i + transactionSize]
for (i in overflow)
  // Insert overflow[i]
```

Fig. 5: HTM-Enabled Hash-Building

memory-bound. While hyper-threading is expected to increase cache contention, we did not observe any significant difference in performance with 4 thread (one per core) and using all 8 threads. This is because the adaptive approach used in TSize-Adaptive adjusts the transaction size to keep the abort rates low and the application remains memory-bound.

4.7   Fallback for fully shuffled data

When there is sufficient locality in the data, our adaptive approach performs best. However, for large shuffle windows, the radix-partitioned approach is still more efficient. Fortunately, as can be see in Figure 4b, the large shuffle windows coincide with high transaction abort rates. We use this insight to develop a hybrid approach that falls back to the radix-partitioned approach when it detects poor locality. We implement this in a straight-forward manner: we use the first 16×1024 tuples of each thread for training and inspect the average abort rate at
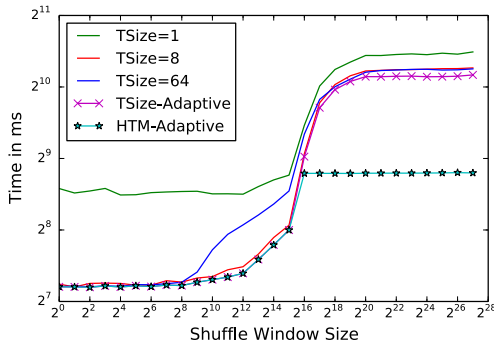
Fig. 6: Adaptive Hash Building (Including Wrap-up)

the end of the training phase[5]. If the abort rate exceeds a threshold (we found 4% to be appropriate in our experiments), we fall back to the radix-partitioning implementation. The training phase runs on a small subset of tuples and hence the overhead (about $4ms$ when using 8 threads) is negligible. Figure 6 shows how the adaptive approach with fallback (HTM-Adaptive) effectively adapts to the data distribution. While our approach does not currently deal with skewed data locality, the identical data structure format of the two approaches make the development of a fully adaptive strategy straight forward.

4.8 Probing

To assess the impact of the presented optimizations on the performance of a full join, we also evaluated performance including the probe phase (counting the number of matches). Note that parallelizing the probe itself is usually not difficult because it does not modify the hash table. However, probe performance is, just as build performance, affected by data access locality. Just like the build, the probe can, therefore benefit from a pre-partitioning step if the tuples are partitioned according to their hashes (or an approximation thereof). Since this effect is independent of the way the hash table is built, we only evaluated the probe using input data with perfect locality (i.e., sorted input data).

In Figure 7, we present the final result of our efforts: an adaptive hash-join implementation using HTM abort rate as runtime feedback variable. The figure illustrates the performance of the full join: build and probe (labeled HTM-Adaptive). It shows that, while our implementation effectively falls back to partitioning the input if locality is low, it outperforms the partitioned approach by more than 3x when locality is high.

For reference, we also included the results of a fully parallel sort-merge join and the Non-Partitioned Join implementation that was used for comparison by Balkesen et al. [1] (labeled "NPJ" in the figure).

---

[5] We considered breaking it down by abort code but found no useful correlation (see Appendix A)
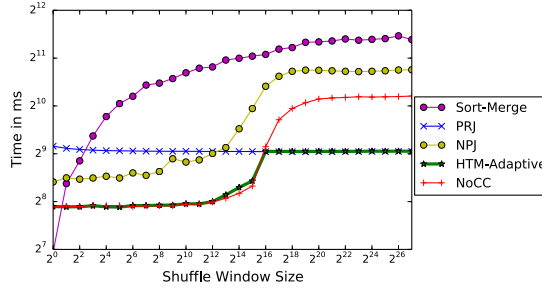
Fig. 7: Full Hash Join (Build and Probe)

The sort-phase of the sort-merge join is based on Timsort [10] which is designed to work well an almost-sorted data. We observe that, only for data that is perfectly sorted (shuffle window size equal to 1) does the sort-merge join outperform our adaptive implementation.

The Non-Partitioned Join was implemented (by Balkesen et al.) using per-bucket spinlocks. As apparent in Figure 7, NPJ performs about 42% worse than HTM for sorted data but degrades in performance once randomness (and thus contention) increases. The reason is two-fold: firstly, the use of spinlocks instead of HTM leads to a 14% slowdown. Second, in NPJ, the locks are co-located with the tuples in the buckets which increases the memory footprint of the resulting table. The HTM approach does not use locks, hence is able to fit 3 tuples per 32 byte bucket compared to 2 tuples in the case of NPJ, which results in another 25% speed-up. Finally, and most apparently, the implementation is not adaptive which can be seen by the performance for inputs with low-locality.

5   Conclusion

Locality in input data is an important, yet often underutilized, factor when developing and selecting appropriate implementations of data management operators. We demonstrated how the state-of-the-art parallel hash join implementations fail to recognize and exploit locality of the input data. To mitigate that problem, we developed an adaptive hash join implementation that uses hardware transactional memory to protect inserts into a shared hash table. We recognized the number of inserts per transaction as the most important performance factor and adaptively tune this parameter at runtime. In addition, our implementation recognizes input data with poor locality and automatically falls back to the current state-of-the-art: parallel radix-partitioned hash joins. The result is a hash join implementation that is more than 3 times faster than the state-of-the-art on high-locality data and never more than 1% slower.

References

1. BALKESEN, C., ET AL. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE* (2013).
2. BLANAS, S., LI, Y., AND PATEL, J. M. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD* (2011).
3. HENNESSY, J. L., AND PATTERSON, D. A. *Computer architecture: a quantitative approach*. Elsevier, 2011.
4. HERLIHY, M., AND MOSS, J. E. B. *Transactional memory: Architectural support for lock-free data structures*. ACM, 1993.
5. JACOBI, C., SLEGEL, T., AND GREINER, D. Transactional memory architecture and implementation for ibm system z. In *MICRO* (2012).
6. KIM, C., ET AL. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *VLDB* (2009).
7. LEIS, V., KEMPER, A., AND NEUMANN, T. Exploiting hardware transactional memory in main-memory databases. In *ICDE* (2014).
8. MAKRESHANSKI, D., LEVANDOSKI, J., AND STUTSMAN, R. To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1298–1309.
9. MANEGOLD, S., BONCZ, P., AND KERSTEN, M. Optimizing main-memory join on modern hardware. *TKDE* (2002).
10. PETERS, T. Description of timsort. http://bugs.python.org/file4451/timsort.txt.
11. ROGAWAY, P., AND SHRIMPTON, T. Cryptographic hash-function basics: Definitions, implications, and separations for collision resistance. In *International Workshop on Fast Software Encryption* (2004).
12. SHAVIT, N., AND TOUITOU, D. Software transactional memory. *Distributed Computing* (1997).
13. TRAN, K. Q., BLANAS, S., AND NAUGHTON, J. F. On transactional memory, spinlocks, and database transactions. In *ADMS* (2010).
14. YOO, R. M., ET AL. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *SC* (2013).

A   Transaction Abort Breakdown

When studying the pseudocode of our approach in Section 4.6, a reader may note that one might use the status code to determine the reason for aborted transactions and use this as runtime feedback. Figure 8 shows a breakdown of the reason for aborts observed with when running the *TSize-Adaptive* implementation (when varying the shuffle window size). Capacity aborts occur if transaction working set exceeds L1 cache size or if more than A cache lines of the same cache set are accessed, where A is the L1 cache associativity. Conflict abort happens if two transactions read/write sets overlap. The main reason we cannot use this information as runtime feedback is that most transaction aborts have return code set to 0, i.e., giving no information about the reason for abort ($RC = 0$ line) and degree of noise is high for the other return codes.
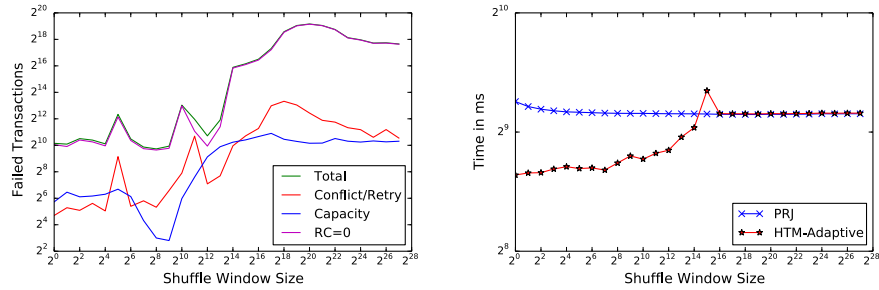

Fig. 8: Reason for Transaction Abort   Fig. 9: Processing uniform random data

B   Non-Unique Inputs

While we consider the problem of efficient conflict handling out of scope of this paper, we still consider it important to establish that the presented techniques do not prevent conflict handling. To illustrate this, consider Figure 9 which corresponds to Figure 7 run on uniform randomly generated integers in the domain 1 to n (the size of the input) which, naturally, includes duplicate values. As in Figure 7, the experiment is to perform the full join but only counting the number of matches. As stated in Section 4, we use bucket-chaining to handle overflows of the buckets and re-inserting to handle aborted transaction. The figure shows a similar pattern to Figure 7 but exposes a suboptimal configuration of the threshold for switching to the radix-partitioned implementation. This indicates that a conflict-aware fallback strategy may be worthwile.