PolyHJ: A Polymorphic Main-Memory Hash Join Paradigm for Multi-Core Machines

Omar Khattab
Carnegie Mellon University in Qatar
okhattab@cmu.edu

Mohammad Hammoud
Carnegie Mellon University in Qatar
mhhamoud@cmu.edu

Omar Shekfeh
ICT Labs Australia
o.shekfeh@gmail.com

ABSTRACT

Relational join is a central data management operation that influences the performance of almost every database query. In this paper, we show that different input features and hardware settings necessitate different main-memory hash join models. Subsequently, we identify four particular models by which hash-based join algorithms can be executed and propose a novel polymorphic paradigm that dynamically subscribes to the best model given workload and hardware characteristics. We refer to our polymorphic paradigm as PolyHJ and suggest a corresponding implementation, which consists of two mechanisms, namely, in-place, cache-aware partitioning (ICP) and collaborative building and probing (ColBP). ICP and ColBP serve substantially in reducing multi-core cache misses, memory bandwidth usage, and cross-socket traffic. Our experimental results demonstrate that PolyHJ can successfully select the right models for the tested workloads and significantly outperform the current state-of-the-art hash-based join schemes.

ACM Reference Format:
Omar Khattab, Mohammad Hammoud, and Omar Shekfeh. 2018. PolyHJ: A Polymorphic Main-Memory Hash Join Paradigm for Multi-Core Machines. In *2018 ACM Conference on Information and Knowledge Management (CIKM'18), October 22–26, 2018, Torino, Italy.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3269206.3271680

1 INTRODUCTION

Recent advancements in hardware technology have revealed remarkable architectural trends. For instance, semiconductor fabrications enable nowadays constructing integrated circuits with thousands of processors [13]. As a response, the industry proceeded rapidly with packing tens and hundreds of cores on a single chip [3]. In addition, cache and memory capacities with tens of megabytes and hundreds of gigabytes, respectively, are increasingly becoming affordable and prevalent [1, 16].

However, under the hood, memory speed is still growing at a lower rate compared to CPU, widening the memory-CPU gap and gradually imposing memory latency as a major bottleneck for performance. Besides, as the number of cores is scaled up, the memory bandwidth available to each core is rather decreased, leading to what is commonly known as the *bandwidth wall problem* [27]. Likewise, although the total size of on-chip cache is increasing, *per-core* (or per-thread) cache of just a couple of megabytes is rendering an average norm across chip generations, resulting in what we denote as the *cache wall problem* (see Fig. 1)[1].

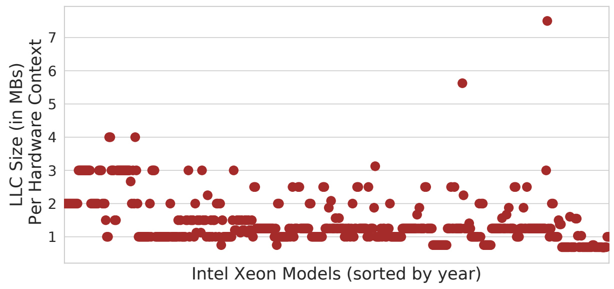

Figure 1: Last-level cache (LLC) size in MBs per hardware context/thread for 481 Intel Xeon models from 2006 to 2018, sorted by year. The trend demonstrates what we refer to as the cache wall problem.

The above trends and bottlenecks lead to the following natural question: how can fundamental and highly influential data management operations harness the increasing power of modern hardware, while overcoming obstacles like the bandwidth and cache wall problems? As an attempt to answer this query, various re-designs of data management operations have been pursued over the last few years [17, 28]. Among these operations, main-memory join, a cornerstone algorithm in database management systems, has constantly attracted attention and gone through various recasts after almost every major architectural innovation [8, 14, 22].

Main-memory joins are typically classified into two major categories, *hash* and *sort-merge joins*. Recent studies show that hash joins are superior in performance to sort-merge ones, especially on large-scale multi-core machines [7, 20, 25]. Consequently, this paper focuses on hash joins, which are themselves categorized into *No-Partitioning* (NOP) and *Partitioned Hash Join* (PHJ) paradigms. NOP and PHJ employ different approaches. In particular, NOP applies a *hardware-oblivious* strategy, in which it entirely precludes partitioning input relations so as to maintain design simplicity and invariability against the characteristics of the hardware. More precisely, it assumes that modern hardware is already good enough in hiding cache and Translation Lookaside Buffer (TLB) misses, which might be caused by its simple design. In contrast, PHJ promotes a *hardware-conscious* approach, whereby it partitions input relations so as to judiciously exploit cache and TLB localities.

[1]Fig. 1 shows only results for Intel Xeon. However, we observed comparable trends for other server CPU models (e.g., AMD Opteron and IBM Power).

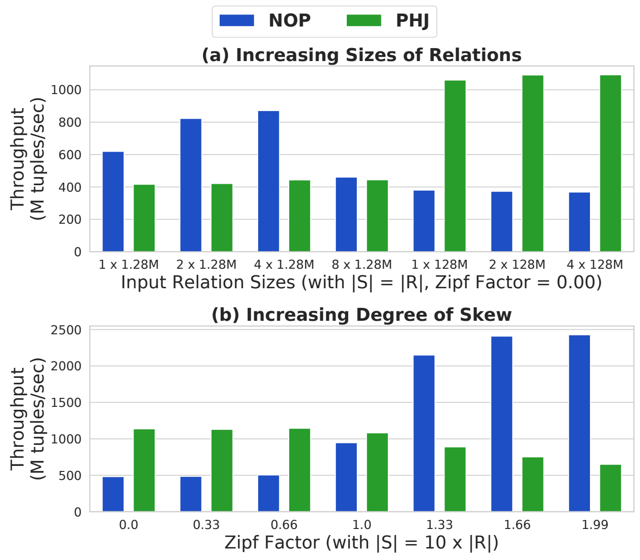

Figure 2: The performance of NOP versus PHJ under: (a) no skew and increasing dataset sizes, with the inner and outer relations being always equal, and (b) increasing skew and a fixed dataset size, with the inner and outer relations consisting always of 128 million tuples (or 977 MiB) and 1280 million tuples (or 9766 MiB), respectively. The figure clearly exhibits the size-skew dichotomy.

As a result, NOP and PHJ demonstrate striking differences in how they handle varying dataset sizes and distributions. Specifically, NOP provides high throughput with small inner relations and/or substantially skewed key distributions [8, 12]. In fact, the performance of NOP improves significantly as skew increases [12]. Conversely, with large inner relations and low-to-moderate skewness, PHJ flourishes and, considerably, outperforms NOP [8, 9, 25]. We studied the behaviors of NOP and PHJ using their state-of-the-art representative schemes, namely, NOPA and CPRA from [25]. Figures 2 (a) and 2 (b) illustrate their opposing behaviors, which we refer to as the *size-skew dichotomy*.

In an attempt to address the size-skew dichotomy and tackle the bandwidth and cache wall problems, this paper proposes *Polymorphic Hash Join* (PolyHJ). PolyHJ adjusts its behavior according to the characteristics of input relations and hardware. Generalizing beyond NOP and PHJ, it decouples the partitioning of relations via partitioning none, one, or both, yet with equal or unequal numbers of partitions. As such, NOP can be pursued by partitioning none of the relations and PHJ by *equally* partitioning both (i.e., generating the same number of partitions for both), alongside other alternatives. Of course, the selection of any alternative will depend on the sizes and distributions of relations as well as the capacities of the Last-Level Cache (LLC) and TLB.

Besides its unique aspect of polymorphism, PolyHJ incorporates re-designed join phases so as to efficiently deal with the limited per-core bandwidth and cache capacity. In particular, PolyHJ adopts a *bandwidth-aware* partitioning mechanism, which interestingly converges towards just one LLC miss per a number of tuples that can fit in one cache line. This results in a maximum of one access to main-memory for a cache line of tuples, greatly alleviating the pressure on the memory bandwidth. PolyHJ accomplishes this via re-ordering input relations (whenever possible), *in-place* and within small and contiguous cached blocks. Alongside, PolyHJ suggests *aggregate LLC-aware* building and probing phases, which seek to effectively utilize logically-shared LLCs. To elucidate, modern high-end CPUs usually adopt a shared LLC among cores on a chip [28]. Consequently, even with a fixed or slightly decreasing per-core cache size, the aggregate LLC capacity increases as the number of cores on a chip is increased. While classical join paradigms overlook this characteristic, PolyHJ leverages it via *efficiently* allowing each built hash table to be as large as the total size of the shared LLC, thus mitigating the cache wall problem.

The main contributions of this paper can be summarized as follows:

- We investigate multiple possible models of hash joins, generalizing them beyond NOP and PHJ. This is achieved through considering various workload and hardware characteristics, including key distributions of input relations, difference in their sizes, and available LLC capacity.
- We present PolyHJ, an open-source[2] polymorphic hash join paradigm, which tackles the size-skew dichotomy via dynamically executing different hash join models for different relations and hardware.
- We propose new bandwidth- and cache-aware implementations of the join phases (i.e., the partitioning, building and probing phases) for multi-core machines.
- We thoroughly evaluate PolyHJ and show that it successfully selects the best hash join model for nearly every tested workload. As an outcome, when all schemes utilize all CPUs, PolyHJ outperforms the state-of-the-art NOP and PHJ schemes in [25] by averages of 2X and 2.4X, and up to 3.7X and 5.3X (excluding few edge cases, which result in 91X speedup), respectively.

The rest of the paper is organized as follows. We present a brief survey of main-memory hash joins in Section 2. Details of PolyHJ are provided in Section 3. We discuss our experimentation methodology and results in Section 4. Finally, Section 5 concludes with main remarks and future directions.

2 RELATED WORK

We now discuss the No-Partitioning (NOP) and Partitioned Hash Join (PHJ) paradigms in more detail and present some of their schemes (or implementations) that are closely related to PolyHJ's.

2.1 No-Partitioning Join

The NOP Paradigm: At its core, the NOP paradigm is a parallelization of the canonical hash join model [8], which suggests a simple, two-phase organization, including a *building* and a *probing* phases. In particular, NOP divides each input relation into sections and assigns a different thread to each section. In the building phase, all threads build concurrently a *single, shared* hash table out of the inner relation (say, *R*), using a specific hash function (say, *h*). To guarantee that parallel writes on shared buckets are mutually

[2]http://github.com/cmuq-ccl/PolyHJ

exclusive, some synchronization mechanism is involved. After the hash table is constructed, the probing phase is triggered, whereby the threads at different sections in the outer relation (say, S) hash tuples using h, locate buckets in the built hash table accordingly, and match R's and S's tuples, before outputting join results.

As implied by its simple design, the phases of NOP are not crafted based on the features of any specific hardware component like (multi-core) cache organization or TLB. Hence, schemes implementing NOP are deemed to be hardware-oblivious. Nonetheless, they are interestingly capable of *naturally* exploiting cache locality when S is highly skewed [8]. Under such workload characteristics, the hash table can be probed efficiently since high skew results in high temporal locality over a small, frequently-accessed subset of buckets in the table.

NOP Schemes: As suggested by the NOP paradigm, any NOP scheme needs to employ a single, shared hash table and usually a mutual exclusion mechanism that synchronizes conflicting writes (or collisions). For instance, Lang et *al.* [21] proposed incorporating a linear-probing technique to handle collisions and a lock-free mechanism to synchronize parallel writes during building. More precisely, they suggested writing tuples that hash to fully occupied buckets into succeeding vacant buckets using an atomic Compare-And-Swap (CAS) operation. More recently, Schuh et *al.* [25] introduced a No-Partition Array (NOPA) scheme, which averts synchronization altogether. In particular, NOPA taps into the fact that the inner input relation is typically joined on a primary key. Consequently, it leverages the uniqueness of the values of this key to store each value at a distinct, unshared bucket in what is referred to as an *array-based* hash table. We compare PolyHJ against NOPA in Section 4.

2.2 Partitioned Hash Join

The PHJ Paradigm: As opposed to NOP, the PHJ paradigm relies on a partitioning strategy to controllably exploit cache locality, irrespective of the key distributions in input relations. As in NOP, each input relation is split into sections and each section is assigned a unique thread. Contrary to NOP, a *partitioning phase* is incorporated before proceeding with a building and a probing phases. In the partitioning phase, each thread partitions its section into an equal number of partitions, referred to as *partitioning fanout*. This is done for both input relations, R and S. Afterwards, a hash table is built for each R's partition by a single thread during the building phase, and probed immediately by (typically) the same thread from the corresponding S's partition during the probing phase.

By partitioning R and S, the PHJ paradigm aims at enhancing locality and diminishing synchronization overhead during building and probing. Typically, the partitioning fanout is selected differently for different workloads so as each hash table can fit into either a private L2 [8, 20, 25] or a portion of a shared Last-Level-Cache (LLC), divided equally among threads [25]. In doing so, PHJ enforces the random accesses caused by hashing to occur on small, cache-resident rather than large, memory-resident hash tables. This has been shown to noticeably reduce cache misses and improve query performance [6]. Besides increasing locality, partitioning entails *isolated* building and probing tasks, whereby each thread builds and probes a pair of partitions from R and S independently. Consequently, all such isolated tasks can be executed in an embarrassingly parallel fashion, with no need for any inter-task synchronization mechanism whatsoever.

PHJ Schemes: The partitioning strategy of PHJ has evolved much since firstly introduced by Shatdal et *al.* [26]. For instance, Manegold et *al.* [22] observed that simply scattering tuples to their respective partitions renders infeasible with large input relations. This is because these relations necessitate large partitioning fanouts in order to guarantee that each hash table can still fit in cache. Unfortunately, while large fanouts can maintain high cache effectiveness, they inversely increase TLB misses during partitioning itself. Consequently, Manegold et *al.* proposed a TLB-aware *multi-pass* partitioning mechanism, wherein an input relation is partitioned over multiple passes, with each pass producing a number of partitions that does not exceed the number of TLB entries.

The work by Kim et *al.* [20], Blanas et *al.* [12], and Balkesen et *al.* [8] contributed towards efficient parallel versions of the multi-pass partitioning mechanism for multi-core machines. Later, Balkesen et *al.* [9] considered using a buffering-based technique (which was proposed earlier for radix sorting by Satish et *al.* [24]) to employ larger partitioning fanouts, yet without involving more than a *single* pass.

More recently, Schuh et *al.* [25] tuned various PHJ implementations for NUMA architectures and evaluated them using different hash table implementations. They observed that existing PHJ schemes ineffectively scatter tuples to NUMA-remote locations over multi-socket machines. Relying on the fact that NUMA-remote writes are usually expensive, they proposed the Chunked Parallel Radix (CPR) join algorithm, in which each thread separately creates its partitions on its local socket. They also suggested CPRA, an extended variant of CPR, which adopts array-based hash tables. In their comprehensive evaluation of thirteen different main-memory join schemes, CPRA demonstrated the highest throughput among the PHJ-based ones. We compare PolyHJ versus CPRA in Section 4.

The main-memory join schemes considered so far are all based on the assumption that both input relations completely fit in memory. Barber et *al.* [10] relaxed this assumption via exploring joins in which only the (smaller) inner relation must fully reside in memory. Subsequently, they proposed a memory-efficient scheme, referred to as *Concise Hash Table Join* (CHTJ), which partitions only the inner relation, but not the outer one. While in fact CHTJ illustrates high performance when the size of the outer relation surpasses memory capacity, later experiments by Schuh et *al.* [25] show that it can be greatly outperformed by NOPA and CPRA (described earlier) otherwise.

Lastly, while this paper focuses on the hash join for mainstream multi-core architectures, we note there is a recent interest in tuning the join for specialized processing units like Intel Xeon Phi [14, 18, 23] and GPUs [15, 19].

3 POLYHJ

3.1 Join Models

PolyHJ is a polymorphic join paradigm, which gracefully adapts to varying workload characteristics and hardware configurations. More precisely, it employs a hash-based strategy, with two *or* three

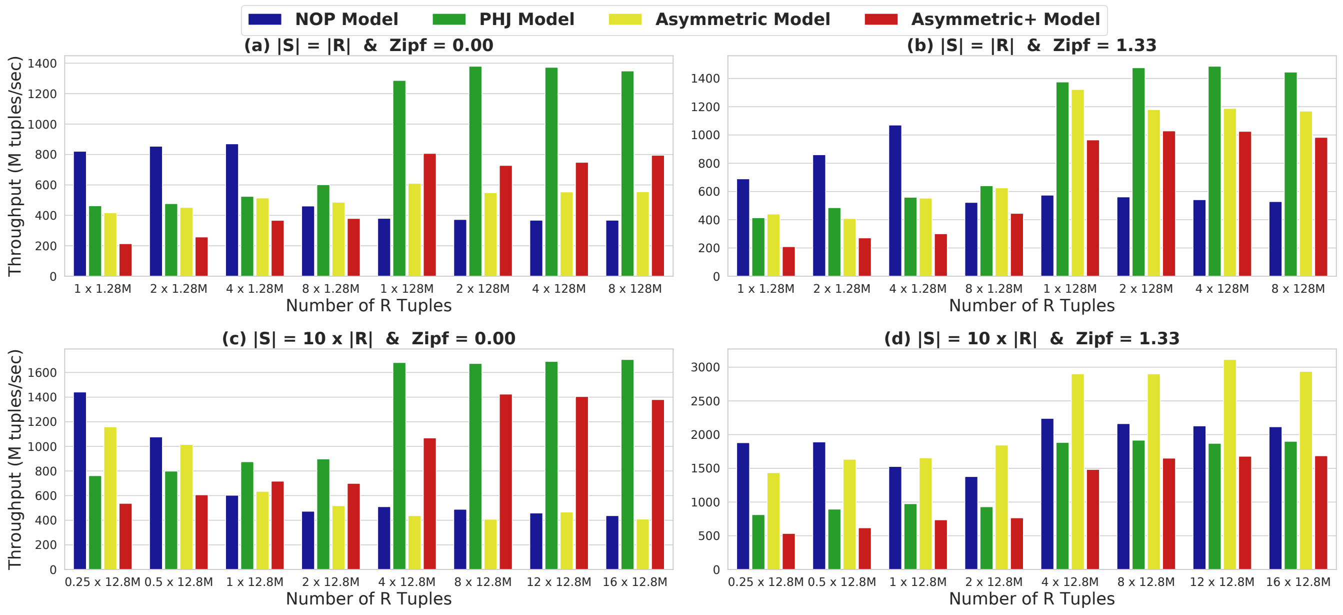

Figure 3: The runtimes of the NOP, PHJ, Asymmetric and Asymmetric+ Models, implemented using the ICP and ColBP techniques, under various dataset sizes and skewness (M = Millions, R = Inner Relation, and S = Outer Relation).

phases, namely (*i*) a *potential* partitioning phase, (*ii*) a building phase, and (*iii*) a probing phase. As discussed in Section 1, PHJ and NOP schemes demonstrate a size-skew dichotomy, whereby PHJ thrives with increasing dataset sizes, but falters in exploiting moderate-to-high skewness. On the flip side, NOP excels naturally with highly-skewed datasets, but greatly degrades under large dataset sizes (see Fig. 2). It is this critical gap between PHJ and NOP that PolyHJ attempts to close, no matter what workload characteristics (i.e., sizes and content distributions of relations) and hardware configurations (i.e., sizes of LLCs and TLBs)[3] are given.

With this goal in mind, we observe that every hash join scheme can be described via two parameters $f_R \geq 1$ and $f_S \geq 1$, which denote the fanouts (or the numbers of partitions) produced by the scheme for the inner and outer relations, R and S, respectively. As such, by applying equal partitioning onto both relations, the PHJ paradigm effectively sets $f_R = f_S > 1$. In contrast, by skipping partitioning altogether, the NOP paradigm essentially sets $f_R = f_S = 1$, treating thereby each relation as a single partition. Henceforth, we refer to the model applied by PHJ as the ***PHJ Model*** and to the one applied by NOP as the ***NOP Model***.

At first glance, the size-skew dichotomy may seem to suggest adopting the PHJ Model if R is large and S has low-to-moderate skew, and the NOP Model otherwise[4]. However, we find that some input characteristics may encourage *decoupling* f_R from f_S (i.e., having $f_R \neq f_S$) as we describe next.

To begin with, recall that the building phase may incur many LLC misses if R is larger than LLC. Thus, it is generally reasonable to partition a large R. In contrast, the probing phase may exhibit high locality if S is sufficiently skewed, resulting in the majority of its tuples getting matched against only a few keys in the hash table. In this case, generating a great deal of partitions for S may be unnecessary, especially that locality will be exploited naturally. This turns even more true when S is significantly larger than R, rendering the partitioning of S quite expensive. Clearly, under these conditions, it becomes more practical to partition R, yet keep the fanout (and, thus, the partitioning cost) of S to a minimum (i.e., setting $f_R > f_S \geq 1$).

Based on the above discussion, we can set the fanout of S to the minimum (i.e., $f_R > f_S = 1$), hence, effectively partitioning only R but not S, then probing all hash tables at once from undivided S. We denote this partitioning model as the ***Asymmetric Model***. Evidently, the Asymmetric Model trades away some locality during the probing phase to gain a faster partitioning phase. For many partitioning techniques (e.g., multi-pass partitioning in [8, 22]), the total cost of partitioning can also be reduced by generating fewer partitions. Consequently, we can also set the fanout of S to a value larger than the minimum, yet smaller than the fanout used for R (i.e., setting $f_R > f_S > 1$). In general, this would result in higher locality during the probing phase compared to the Asymmetric Model, yet in lower locality compared to the PHJ Model. We refer to this model as the ***Asymmetric+ Model***[5]. In the Asymmetric+ Model, each S partition will probe some, rather than all or one, hash tables at once.

We conducted a set of experiments to evaluate the above stated premises of the NOP, PHJ, Asymmetric and Asymmetric+ Models. Fig. 3 shows our results using *static* PolyHJ over varied dataset

[3]On modern systems, TLB entries usually span more pages than what an LLC can equivalently fit, particularly with 2MiB page sizes and above (as proposed in [9, 25]). Hence, in the rest of this paper we assume TLB-awareness when suggesting LLC-awareness.

[4]We assume (as related work [8, 12, 25]) the inner and outer relations R and S are joined on primary and foreign keys, respectively. Thus, skewness only applies to S.

[5]To the contrary of these four models, the cases where $f_S > f_R \geq 1$ would be typically undesirable. Given that R is partitioned so that each hash table can completely reside in cache, partitioning S onto (just) an equal fanout would be sufficient to probe cache-contained hash tables.

sizes. In particular, we disabled PolyHJ's dynamic model selection for the given input and manually instructed it to run over each input dataset using all possible models. Moreover, we considered equal and unequal R and S (see Figures 3 (a) and (b)) as well as uniform and non-uniform S (see Figures 3 (c) and (d)). Our software and hardware configurations are discussed in Section 4.1. Based on these configurations, the datasets with $|R| \in \{1 \times 1.28M, 2 \times 1.28M, 4 \times 1.28M, 0.25 \times 12.8M, 0.5 \times 12.8M\}$ tuples fit either entirely or almost entirely in LLC, hence, do not necessitate partitioning R. As expected and depicted in Figures 3 (a), (b), (c), and (d), the NOP Model outperforms all other models under these datasets since it avoids partitioning R, let alone S.

Reciprocally, when the size of R exceeds the size of LLC, partitioning R becomes crucial. This is demonstrated in Figures 3 (a), (b), (c), and (d) with all the tested datasets, except the smaller ones mentioned above. Under this setting, if $|R|$ and $|S|$ are equal, the PHJ Model appears superlative as illustrated in Figures 3 (a) and (b). Unlike the NOP Model, it is noticeable that the Asymmetric Model is competitive in Fig. 3 (b). On the other hand, if S is uniform and significantly larger than R, and R is larger than LLC, the competition ensues between the PHJ Model and the Asymmetric+ Model since partitioning S remains advantageous (see Fig. 3 (c)). However, if S is highly skewed and greatly larger than R, which in turn is larger than LLC, partitioning S becomes less effective and the Asymmetric Model dominates (see Fig. 3 (d)).

To conclude, Fig. 3 clearly shows that there is no *one-size-fits-all* model for all types of workloads and that different workloads usually favor different models. PolyHJ attempts to address this problem via choosing the best model for any given workload (based on techniques described in Section 3.4). We next discuss PolyHJ's internal engine.

3.2 In-Place, Cache-Aware Partitioning

As discussed in Section 1, the bandwidth- and cache-wall problems pose major scalability issues for in-memory joins. Hence, besides its polymorphic behavior, PolyHJ directly addresses these two problems via incorporating an *in-place, cache-aware partitioning* (ICP) and a *collaborative building and probing* (ColBP) techniques. In doing so, PolyHJ seeks to accomplish high scalability with various dataset sizes and numbers of cores. We discuss ICP in this section and ColBP in Section 3.3.

As indicated by its name, ICP is capable of partitioning an input relation by re-ordering its tuples *in-place*. In doing so, ICP only scatters tuples over small, contiguous, and cache-contained memory areas. Contrary to typical partitioning implementations, ICP does not need separate destination partitions, but rather re-orders the relation(s) on-the-fly so as to avoid wasting memory bandwidth.

Similar to other multi-threaded join schemes, R (similarly, S) is divided into equal *sub-relations*, each assigned to a single thread. Independently, each thread splits its sub-relation into small *blocks* of equal size, chosen so that at least two blocks fit in the thread's share of LLC (see Section 3.4 for more details on threads' shares of LLC). For each block b_j (and not for the entire sub-relation), the thread applies traditional single-pass partitioning (refer to [8, 22]). Importantly, since two blocks fit in cache, the thread scatters the partitioning output of block b_{j+1} onto the space originally occupied by the preceding block b_j. As b_0 does not have a preceding block,

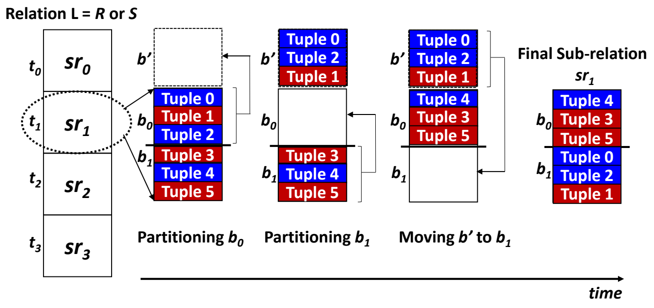

Figure 4: An illustration of how ICP is executed over sub-relation sr_1 by thread t_1, which divides sr_1 into two blocks, b_0 and b_1. The block b' is a temporary block for holding the tuples of only the first block (i.e., b_0) during partitioning. The partitioning fanout is assumed to be 2, hence the usage of two colors in each block.

it is temporarily partitioned into an external buffer b'. It is finally restored to the bottom of the sub-relation, once all the other blocks have been scatterd, thus preserving the integrity of the full relation[6].

To exemplify, Fig. 4 illustrates how ICP is carried over the sub-relation sr_1 by thread t_1. In parallel, all other sub-relations are partitioned in the same manner. We assume a fanout equal to 2 (hence, the usage of two different colors) and two blocks per sub-relation (say, b_0 and b_1). Evidently, after partitioning all the sub-relations of the input relation, L (which could be R or S), tuples will be organized in a way that a partition of L will be *distributed* over all blocks across the sub-relations, yet will be placed *contiguously* within each block in a given sub-relation.

The design of ICP emphasizes a tradeoff between better locality during scattering versus higher contiguity among the tuples of each output partition in memory. To explain this, the smaller the block size is, the more cache-friendly the hash-based scatter accesses are during partitioning. In contrast, the larger the block size is, the more contiguous the tuples of each produced partition become. For instance, a sub-relation with only one block (i.e., similar to more traditional partitioning) will have all the tuples of the same partition co-located together. Obviously, this would entail more sequential and, hence, more efficient scans of each partition during the building and probing phases, but more random scatters over a large non-cached region during the partitioning phase.

To achieve high locality and decent contiguity, PolyHJ sets the block size to be a fraction ($< 1/2$) of the size of LLC per thread. Thus, each block b_j is expected to remain in cache after initially scanned. Subsequently, the region originally occupied by b_j is re-used, after reading and partitioning its tuples, for writing (or scattering) b_{j+1}'s tuples to it. Since the aggregate size of b_j and b_{j+1} has been set to be less than the size of LLC per thread, all accesses to main-memory in ICP are sequential scans (when firstly reading a block), and virtually all random scatterings occur within cache.

Lastly, the other PolyHJ mechanism, namely, ColBP, works in tandem with ICP to reduce the number of generated partitions. This increases the number of tuples in each sub-partition produced

[6]If, for any reason, the input relation cannot be re-ordered, PolyHJ can always fall back to a classical buffering-based partitioning.

out of a sub-relation. Consequently, contiguity is enhanced while locality is still exploited, thus maximizing the benefit of ICP. We next discuss ColBP.

3.3 Collaborative Building and Probing

Besides ICP, PolyHJ suggests a collaborative building and probing (ColBP) mechanism. Once a thread has executed ICP on its assigned sub-relation, it can trivially locate the beginning of each (now-partitioned) block b_j, as all but the last block are equal-sized. By maintaining a pointer to the current position in each block, the thread can scan over each partition P_k, across all of its blocks, before proceeding with building/probing partition P_{k+1}.

While scanning each partition, a thread scatters its R tuples (i.e., those in its sub-relation), and performs probing using its S tuples, to/from their respective hash table(s). Consequently, all threads *collaborate* in the construction and probing of each hash table, rather than being assigned mutually-exclusive subsets of partitions as is the case with common PHJ schemes. As discussed in Section 4.4, this aspect is critical to avoid load-imbalance under skew.

As opposed to traditional techniques, the behavior of ColBP depends on the number of LLCs, denoted as llc_{num}[7]. Specifically, given the availability of sufficiently many R partitions, ColBP guarantees that, at any given time, each hash table is shared for writes (i.e., during the building phase) among *only* the cores on a single LLC[8]. To elaborate, ColBP starts by dividing the available threads into *groups*, whereby the threads in each group share an independent LLC. During the building phase, while threads within a group concurrently build a shared hash table (using an NOP-like implemenation per partition), threads *across* groups do not scatter tuples to the same hash table simultaneously. Instead, groups take turns in constructing hash tables. Once a group is done building its share of a hash table, it swaps with another finished group to proceed with building its part of another hash table.

The above strategy serves in alleviating coherence traffic and potentially NUMA latency across LLCs and sockets. More precisely, it allows each hash table to reside at a single LLC for the whole duration while it is being built by a group at that LLC. Clearly, this increases LLC hits. If, instead, the table was to be constructed simultaneously by multiple groups, its constituent shared cache lines would keep flip-flopping between different LLCs whenever they are altered by threads from different groups. This is typically necessitated and enforced by a coherence protocol (e.g., MESI [27]) at the architecture level so as to protect cache lines from write-write conflicts during execution.

Finally, we note that the design of ColBP is more scalable than existing approaches. In particular, as larger input relations are to be joined in main-memory, typical PHJ schemes will likely react by increasingly applying more expensive pass(es) during partitioning, so as to fit each hash table in the available cache capacity per thread. As discussed in Section 1, the cache-wall problem manifests itself noticeably in modern architecture, wherein the per-core cache capacity shows little or no growth even as the number of cores per chip is increased (see Fig. 1). However, we observe that many modern high-end CPU models have a single, logically-shared LLC per chip (or socket). The capacity of this LLC is growing as more cores are packed on a chip, despite the cache-wall problem.

To this end, we designed ColBP to mitigate the effects of the cache-wall problem via allowing each hash table to be as large as the total size of each available LLC. Thus, ColBP requires a fanout that is much smaller than those needed by typical PHJ schemes. Interestingly, this keeps the cost of partitioning manageable for the ever-growing dataset sizes.

3.4 Model Selection

PolyHJ seeks a favorable tradeoff between the costs of ICP and ColBP. Given a workload and hardware, it dynamically configures the fanouts f_R and f_S as follows, in an attempt to select the most suitable join model.

To allow for an efficient building phase, PolyHJ partitions relation R, if necessary, so that each partition can be scattered over a hash table that can fit in LLC. Let each LLC have capacity to hold C tuples, and let $|R|$ be the number of tuples in relation R. PolyHJ sets $f_R = 2^r$ for $r = \max(\left\lceil \log_2(\frac{|R|}{C}) \right\rceil, 0)$[9].

Likewise, to allow for an efficient probing phase, PolyHJ partitions relation S, if necessary, so that the locality of probing accesses to the hash tables is sufficiently high. In principle, we want to estimate the skewness in S for comparison against predefined thresholds when dynamically configuring f_S. Observe that if, after partitioning S, the majority of its tuples would belong to only few partitions, then partitioning S would not be very effective, as the largest partition(s) will be close in size to S itself. In this case, partitioning will contribute little extra locality to the probing phase (i.e., little beyond the locality induced by skew itself) and would be expensive if S is significantly larger than R, hence, favoring the Asymmetric Model. In contrast, if partitioning S would evenly distribute its tuples across most or all partitions, substantial scattering must happen during partitioning, resulting thereby in better locality during probing, thus suggesting the PHJ Model.

Based on this observation, whenever S is significantly larger than R (i.e., $|S| > k \times |R|$ for some pre-determined constant $k > 1$), PolyHJ decides between the PHJ Model and the Asymmetric Model by initially and temporarily setting $f_S = f_R$. Afterwards, PolyHJ performs sampling via starting ICP at different offsets throughout S, using all the threads concurrently. Since partitioning internally begins by counting the number of tuples that will belong to each partition (i.e., building a *histogram* [8]), threads can start doing so before finalizing f_S. This process is applied over a given sample size of tuples (say, a block per sub-relation, assuming sufficiently many sub-relations). If, overall, threads report high skew among partition sizes (i.e., that the m largest partitions will contain more than an α fraction of the sampled S tuples, for thresholds $m \geq 1$ and $0 < \alpha < 1$), ICP is terminated on S, and the Asymmetric Model is pursued. Otherwise, PolyHJ simply continues ICP normally, effectively choosing the PHJ Model[10]. This sampling-based heuristic

[7]For instance, a machine with two sockets and a last-level L3 cache per each socket *shared* among the socket's cores entails an llc_{num} of 2. In contrast, a machine with one socket of N cores, but with only a *private* LLC per core, implies an llc_{num} of N.
[8]Of course, this is not applicable to the NOP Model, which builds a single hash table using all threads.

[9]In practice, only a constant fraction (e.g., 1/2) of the total LLC capacity is used as C. This takes into account cache pollution by other processes that may be running concurrently on the server.
[10]Since PolyHJ relies on an efficient partitioning mechanism (i.e., ICP), the gain from partitioning S on a fanout smaller than (an already-small) f_R (i.e., as in the Asymmetric+

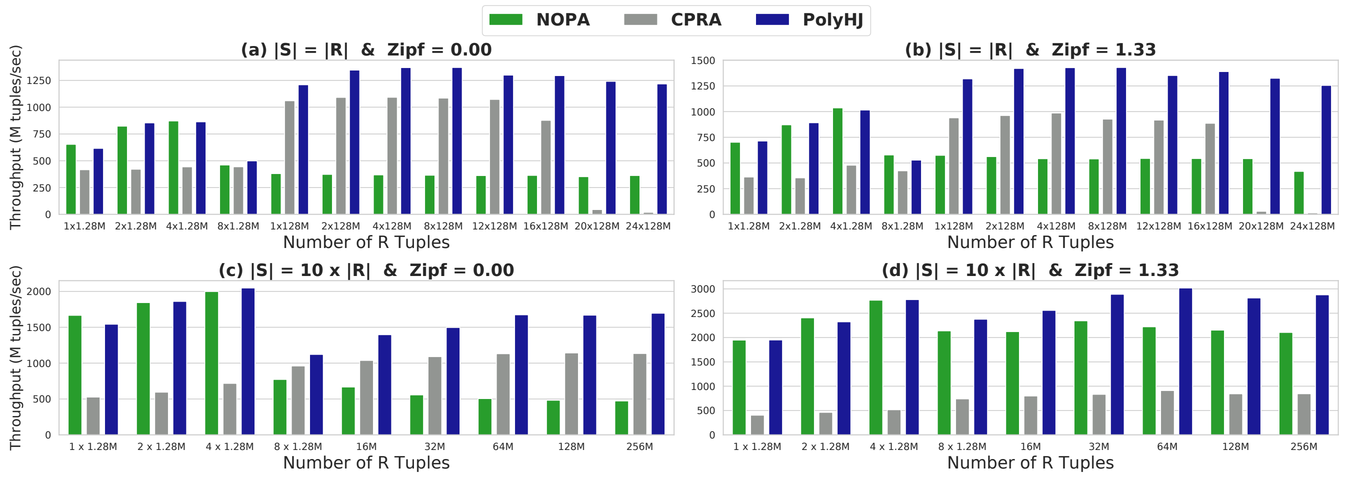

Figure 5: Scalability results for NOPA, CPRA, and PolyHJ under different workload types.

has a minimal overhead on the runtime of PolyHJ, and is included in the results shown in Section 4.

4 EXPERIMENTS

4.1 Methodolgy

In this section, we evaluate the performance of PolyHJ against the state-of-the-art NOP and PHJ schemes, namely, NOPA and CPRA (see Section 2 for details on both schemes). We implemented and verified PolyHJ in C. The authors of NOPA and CPRA shared with us their experimental testbed, which included implementations for thirteen join variants studied in their paper. We carefully extracted and integrated NOPA and CPRA into our testbed along with PolyHJ. All code was compiled using GCC, version 5.4.0, with the optimization flag -O3.

We conducted all our experiments on a dual-socket server, with two Intel Xeon E5-2650 v3 CPUs. Each CPU is comprised of 10 physical cores, each encompassing 2 hardware threads (i.e., the machine has 40 hardware contexts in total). The threads of each socket share a 25MiB L3 LLC cache, and the server includes a total of 64GB of DRAM, distributed equally among its two sockets. As for the OS, we utilized 64-bit Ubuntu 16.04 (kernel version 4.4.0-45) and a page size of 2MiB as suggested in [9, 25].

Similar to related work (e.g., [8, 14, 25]), we adopt 8-byte tuples as (*key*, *payload*) pairs, consisting of two 4-byte integers[11]. As is the case with NOPA and CPRA, we use array-based hash tables and assume joins are pursued between primary keys at inner input relation, R, and corresponding foreign keys at outer input relation, S. For each workload, we run CPRA with an appropriate range of radix bits and select the ones that provide the best performance. In PolyHJ, the fanouts f_R and f_S are rather chosen dynamically as discussed in Section 3.4. Of course, the cost of estimating skew to select fanouts for any given workload is part of the PolyHJ scheme and, hence, is included in our reported runtime results.

We investigate various combinations of workloads in terms of input relation sizes and key distributions. In particular, we consider four types of workloads, namely: (1) WT_1, where input relations, R and S, are of equal sizes and exhibit uniform key distributions, (2) WT_2, where R and S are of equal sizes, but S demonstrates high non-uniform key distribution, (3) WT_3, where R and S are unequal, but uniform, and (4) WT_4, where R and S are unequal, and S is highly skewed.

We first study the scalability of NOPA, CPRA, and PolyHJ via varying the input relation (or dataset) sizes per each workload type. This allows us also to scrutinize the polymorphic behavior of PolyHJ under different datasets, and, accordingly, observe its effectiveness in handling the size-skew dichotomy. Second, we examine the scalability of the three schemes with respect to thread-level parallelism, using solo and dual sockets. This provides us with insights into how NOPA, CPRA, and PolyHJ can perform under various architectural settings, including concurrency, simultaneous multithreading (SMT), and non-uniform memory (NUMA) accesses, let alone efficacy in addressing the bandwidth-wall problem. Third, we explore the effects of varied key distributions on the three schemes and show how PolyHJ can efficiently run under any skewness level. This also demonstrates PolyHJ's capability in solving the size-skew dichotomy. We also present LLC-related statistics so as to illustrate the cache effectiveness of PolyHJ versus NOPA and CPRA, and correlate that with our reported runtime/throughput results.

4.2 Scalability with Dataset Sizes

We now compare the performances of NOPA, CPRA, and PolyHJ as we scale dataset sizes, using our four types of workloads outlined in Section 4.1. Specifically, we vary dataset sizes between 1×1.28M and 24×128M tuples as shown in Fig 5. Whenever the total number of tuples ($|R|+|S|$) of any workload is within a small multiple of a single LLC capacity, we run the workload using all hardware contexts of a solo socket (i.e., using 20 threads). The workloads that satisfy this condition are the ones that consist of $|R| = |S| = \{1 \times 1.28\text{M}, 2 \times$

Model) does not usually offset the loss from probing hash tables that are larger than LLC(s). Hence, PolyHJ does not automatically select the Asymmetric+ Model. This observation has been corroborated in Fig. 3.

[11] In database systems, dictionary encoding [5, 11] is often employed to map long or variable-length values (e.g., strings) to fixed-length integer codes.

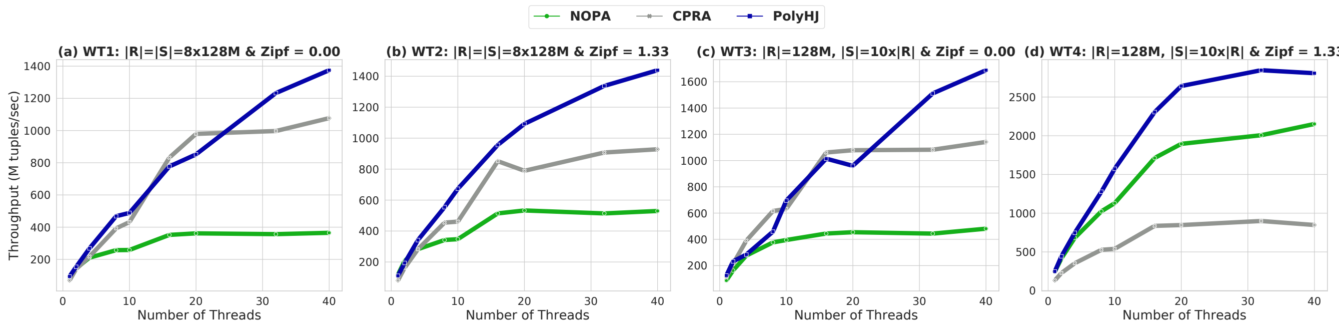

Figure 6: Thread scalability results for NOPA, CPRA, and PolyHJ under different workloads.

1.28M, 4×1.28M, 8×1.28M} tuples. The remaining workloads dwarf a single LLC, hence, are executed using all hardware contexts of dual sockets (i.e., using 40 threads). Evidently, by using a single socket for small datasets, all schemes (especially NOPA) can maximally leverage a shared LLC and preclude unnecessary NUMA accesses.

Prior to analyzing our results, it is worth noting that the single hash table of NOPA completely resides in one LLC with dataset sizes of $\{1 \times 1.28\text{M}, 2 \times 1.28M, 4 \times 1.28\text{M}\}$ tuples. Figures 5 (a), (b), (c), and (d) depict our scalability results for workload types, WT_1, WT_2, WT_3, and WT_4, respectively. With workloads including up to $|R| = 4 \times 1.28$M tuples, PolyHJ adopted the NOP Model, just like NOPA. With larger R, PolyHJ embraced the PHJ Model under WT_1, WT_2, and WT_3, and the Asymmetric Model under WT_4, just as expected. As a first observation, PolyHJ almost always selects a join model that is at least as good as the best amongst NOPA and CPRA. More often than not, we can even see that PolyHJ strictly outperforms both join schemes, and not just one.

Let us delve deeper into the behaviors of NOPA, CPRA, and PolyHJ. In Figures 5 (a), (b), and (c), we observe a clear trend. That is, under a small relation R, NOPA and PolyHJ perform very closely, both outpacing CPRA. This occurs irrespective of skew or the ratio between $|S|$ and $|R|$. On larger R, the competition ensues between CPRA and PolyHJ, whereby both start demonstrating an edge over NOPA (by not partitioning large R, NOPA incurs very high LLC misses- see Table 1). In this case, both CPRA and PolyHJ represent the PHJ Model. However, the re-designed phases of PolyHJ grant it an overall lead.

In Fig. 5 (a) and (b), the above trend abruptly stops on very large input (i.e., 20×128M and 24×128M tuples), wherein the performance of CPRA falters tremendously. This is because CPRA replicates input relations during partitioning, causing its working set size to exceed the DRAM capacity when such relations are quite large (which could still fit in DRAM if not replicated). Consequently, the kernel is forced to swap several GBs of data in and out the disk, degrading thereby CPRA's performance. Interestingly, PolyHJ circumvents this slippery slope when executing large-scale workloads via adopting an in-place mechanism (i.e., ICP), which avoids replicating relations during partitioning.

Finally, we glean two more observations from Figures 5 (c) and (d). First, the performance of CPRA does not change tremendously across WT_3 and WT_4. In particular, CPRA shows an average degradation of 24% under WT_4 versus WT_3 due to high skew (more on this in Section 4.4). Second, the throughput of NOPA improves by an average of 167% under WT_4 as opposed to WT_3. Noticeably, this confirms the strength of NOPA, which naturally picks up under unequal and highly skewed datasets (as discussed in Section 2). In contrast, PolyHJ synergistically adapts to workload characteristics and considerably outperforms NOPA and CPRA. For instance, with sufficiently large and highly skewed S (i.e., under WT_4), PolyHJ averted partitioning S via choosing the Asymmetric Model, tapping (at least) into the natural locality that can be exploited under high skew. Besides locality, this enabled PolyHJ to save the cost of partitioning S, thus obtaining better results under WT_4 than WT_3.

In short, Fig. 5 shows how PolyHJ can effectively handle the size-skew dichotomy. PolyHJ was able to provide average speedups versus NOPA by 2.3X, 1.8X, 1.9X and 1.15X, and up to 3.7X, 3X, 3.5X and 1.35X under WT_1, WT_2, WT_3 and WT_4, respectively. Alongside, it outperformed CPRA by averages of 2.45X, 3X, 1.8X and 3.8X, and up to 64.5X, 91X, 3.1X and 5.3X under WT_1, WT_2, WT_3 and WT_4, respectively[12].

4.3 Scalability with Numbers of Threads

In this subsection, we compare the scalabilities of NOPA, CPRA, and PolyHJ with respect to thread-level parallelism under our four workload types. In particular, per each workload type, we select a representative workload, namely, A, B, C, and D of types WT_1, WT_2, WT_3, and WT_4, respectively (see Fig. 6). For each workload, we vary the number of threads, starting with 1 thread and scaling it up to 40 threads. As long as the number of threads is less than or equal to 10 (i.e., the number of cores on each socket), we schedule all the threads on distinct cores on a single socket. This entails local memory allocations, local NUMA accesses, and one shared LLC for all schemes. As we move up to 16 and 20 threads, we schedule the threads equally across two sockets, resulting in remote NUMA accesses, and two shared LLCs for all schemes. Nonetheless, no SMT is utilized yet (i.e., no two threads are scheduled on a single core). With 32 and 40 threads, we apply equal SMT across sockets.

[12]The effectiveness of PolyHJ's redesigned implementation can be isolated by studying the experiments in which PolyHJ subscribes to the PHJ model like CPRA. In such experiments that utilize all available CPUs (i.e., over Sections 4.2 and 4.4), PolyHJ outperforms CPRA by 1.4X on average (geometric mean), even after excluding experiments where CPRA is forced to swap to disk. This is a conservative estimate; we manually tuned the radix bits for CPRA, while PolyHJ automatically selects its fanouts.

Clearly, the above scheduling criteria allows us to test and evaluate NOPA, CPRA, and PolyHJ under various architectural conditions. Nevertheless, we note that CPRA is sensitive to all these conditions due to being hardware-conscious. As such, we had to run it with various fanouts per each number of threads and report only the best performing one. Figures 6 (a), (b), (c), and (d) display our results. To start with, we observe an asymmetry in the scalabilities of NOPA and CPRA. In particular, CPRA scales better under workloads *A*, *B*, and *C* as opposed to *D*. On the flip side, NOPA scales better under *D* as compared to *A*, *B*, and *C*. Interestingly, PolyHJ demonstrates steady scalability across all workloads. Under *A*, *B*, and *C*, PolyHJ selected the PHJ Model, while it adopted the Asymmetric Model under *D*. We next elaborate on the reasons behind such behaviors of NOPA, CPRA, and PolyHJ.

The limited scalability of NOPA under workloads *A*, *B*, and *C* but not *D* is mainly due to a bandwidth issue. Specifically, in NOPA, when *R* and *S* are equal and large, or *S* is larger than *R* but uniform, almost no locality is exploited during building/probing the (larger than LLC) hash table. This was the case for *A*, *B*, and *C*, whereby near 100% LLC miss rates were incurred. Per each LLC miss, DRAM was accessed and a cache line consisting of 16 buckets was placed at LLC. Due to poor locality, very often, only one bucket was actually used from the cache line before it was evicted. As we scaled up the number of threads, the available memory bandwidth got rapidly saturated[13]. To this end, NOPA did not scale well under these three workloads (especially under *A*, wherein both *R* and *S* are large and uniform). Conversely, under *D*, *S* is much larger than *R* and subsumes a highly skewed distribution. This yielded a significantly better LLC hit rate during the probing phase (see Table 1). As more threads were added, NOPA maintained good scalability under *D*, leveraging higher LLC locality and fewer accesses to DRAM.

PolyHJ and NOPA scale quite similarly under workload *D*, but PolyHJ provides higher throughputs. This is because PolyHJ chose the Asymmetric Model under *D*, thus partitioned *R* and subsequently exploited *more* locality versus NOPA (which does not either relation; see Table 1 for LLC statistics). As for CPRA against PolyHJ, from 1 to 20 threads they demonstrate comparable scalability under workloads *A*, *B*, and *C* (see Figures 6 (a), (b), and (c)). With 32 and 40 threads (i.e., when SMT is enabled), PolyHJ significantly outpaces CPRA. To explain this, we observed that the partitioning phases of CPRA and PolyHJ exhibit similar scalability under SMT (though result in different throughputs). The ColBP phase of PolyHJ, however, scales very well under SMT, contrasting the building and probing phases of CPRA. This is because PolyHJ demonstrates higher (yet reasonable) LLC misses during ColBP, which enables SMT to hide more latency and, accordingly, further expedite PolyHJ's performance. To the contrary of PolyHJ, under workload *D*, CPRA reveals limited scalability due to staggering with high skew (see reasons in Section 4.4).

Overall, the results in Fig. 6 shed light on the importance of designing highly parallel join algorithms that are both bandwidth- and skew-aware. For instance, while NOPA performs well under high skew, its scalability is substantially hindered under low or no skew due to the bandwidth-wall problem. On the other hand, while CPRA addresses effectively the bandwidth-wall problem via applying partitioning, its scalability is hampered under high skew. Remarkably, PolyHJ is bandwidth- and skew-aware, hence, performs and scales well under low-to-high parallelism and with all sorts of workloads.

[13]In our setup, each NUMA node has peak bandwidth of about 24 GB/sec for NUMA-local accesses and 11.5 GB/sec for NUMA-remote ones. These values are based on 1:1 reads/writes ratio, obtained using Intel Memory Latency Checker [2]. This suggests a peak per-core NUMA-local bandwidth of just a couple of GB/sec when all cores are competing for bandwidth.

Workload	Scheme	Building/Partitioning Phase		Probing/Joining Phase	
		L3 Misses [M]	L3 Hit Rate [%]	L3 Misses [M]	L3 Hit Rate [%]
A	NOPA	1146	1.7	1147	0.9
	CPRA	280	85.9	257	86.1
	PolyHJ	53	89.0	455	80.5
B	NOPA	1147	1.6	100	44.1
	CPRA	239	80.7	165	82.4
	PolyHJ	47	88.8	357	72.1
C	NOPA	138	3.9	1382	4.0
	CPRA	193	82.1	94	74.9
	PolyHJ	31	87.6	206	86.9
D	NOPA	138	3.9	121	44.2
	CPRA	106	71.6	20	72.0
	PolyHJ	3	90.1	153	57.2

Table 1: LLC misses and hit rates for NOPA, CPRA, and PolyHJ under workloads shown in Fig. 6.

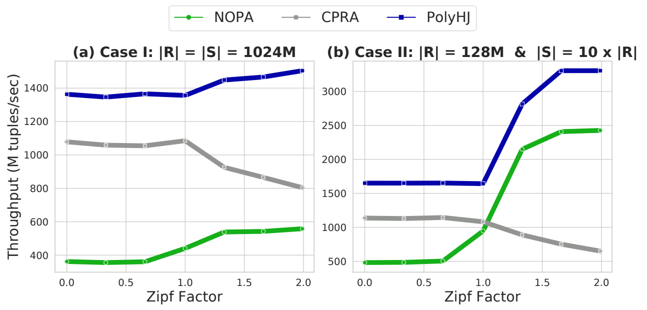

Figure 7: Throughput results of NOPA, CPRA, and PolyHJ under equal and unequal input relations, and varied skew.

4.4 Effect of Skew

We now study the effect of non-uniform key distributions on the throughputs of NOPA, CPRA, and PolyHJ. In particular, we test the robustness of PolyHJ under a wide range of values by varying skewness on a Zipfian scale from 0.0 (completely uniform) to 1.99 (extremely high skew) similar to the work at [8]. As we focus on skewed distributions, we consider two cases in terms of input relations, *R* and *S*, namely, (1) $|S| = |R|$ (***Case I***) and (2) $|S| = 10 \times |R|$ (***Case II***). Alongside, we utilize all the hardware contexts of our dual-socket server (i.e., 40 threads) to evaluate the cases. Fig. 7 demonstrates our results.

Under Case I, we observe that the impact of skew is generally slim on NOPA, CPRA, and PolyHJ (see Fig. 7 (a)). Specifically, as we increase skewness, the throughputs of NOPA and PolyHJ are slightly improved, while that of CPRA is somewhat degraded. In all such runs, PolyHJ subscribed to the PHJ Model. As described in Section 2, higher skew enhances locality during probing and, subsequently, serves in expediting performance. This explains the improvement in NOPA's performance, starting from Zipf = 1.00 onwards, let alone PolyHJ which leverages even more locality as a fact of applying partitioning. In contrast, under high skew, CPRA suffers from load imbalance during probing, manifested in particular partitions, which render (significantly) larger than others [25]. Consequently, the threads assigned to large partitions run much longer than others. As is typically the case in systems, the performance of CPRA becomes bound to the slowest thread.

Under Case II, we notice a two-stage trend. The first stage covers Zipf values from 0.00 to 1.00, whereby the relative performance of NOPA, CPRA, and PolyHJ is dictated by the dataset size. To begin with, CPRA outperforms NOPA because NOPA's single hash table is sufficiently larger than LLC. Similarly to CPRA, PolyHJ partitioned *R* via embracing the PHJ Model and, accordingly, surpassed NOPA. Moreover, it outstripped CPRA (which also employs the PHJ Model) due to its superlative ICP and ColBP mechanisms. The second stage spans Zipf values from 1.33 and beyond, wherein we spot much higher throughputs for NOPA and PolyHJ, but not CPRA. As discussed earlier, higher skew provides NOPA with greater locality during the probing phase, especially if *S* is substantially larger than *R*. This is precisely the situation under Case II, hence, NOPA shows better corresponding results as opposed to Case I. Besides NOPA, increased skew allows CPRA to exploit higher locality as well, but reversely causes load imbalance, degrading thereby its overall performance.

As compared to NOPA and CPRA, PolyHJ exploits higher locality and inherently avoids load imbalance as skewness is escalated. In particular, in the second stage of Case II, PolyHJ selected the Asymmetric Model, tapping into the natural locality that can be obtained with higher skew and saving the cost of partitioning *S*. Hence, as all threads collaboratively probed *R*, load imbalance was effectively precluded. As a result, PolyHJ maintained a strong edge over CPRA and NOPA.

5 CONCLUSIONS

In this paper, we presented PolyHJ, a novel polymorphic main-memory, hash-based join paradigm, which dynamically selects the best join model for any given workload and hardware characteristics. In addition, we proposed a full-fledged scheme that implements PolyHJ using two highly parallel and scalable techniques, namely, in-place, cache-aware partitioning (ICP) and collaborative building and probing (ColBP). ICP and ColBP effectively address the bandwidth and cache wall problems via judiciously mitigating random writes to memory and controllably producing hash tables that are as large as each aggregate LLC. To encourage reproducibility and extensibility, we make our PolyHJ code publicly available at [4]. Our experimentation results show that PolyHJ can effectively meet its design goals and greatly surpass the two state-of-the-art hash-based join schemes, NOPA and CPRA [25].

Finally, we set forth two main future directions. Firstly, we utilized ICP and ColBP as implementations for the partitioning and building/probing phases in PolyHJ, respectively. In principle, other implementations of the PolyHJ paradigm can be pursued, entailing different model selection strategies. Consequently, we plan to develop a generic model selection strategy, which can adapt and work with any implementation of PolyHJ. Secondly, akin to NOPA and CPRA, ColBP relied on array-based hash tables. However, it would be useful to consider our presented four join models under other joining options (e.g., joining inner relation on a non-primary key), which may introduce auxiliary costs and/or benefits for some of the models.

ACKNOWLEDGMENTS

This publication was made possible by NPRP grant # 7-1330-2-483 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

REFERENCES

[1] [n. d.]. IBM Power. http://www-03.ibm.com/systems/power/.
[2] [n. d.]. Intel© Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.
[3] [n. d.]. Intel© Xeon Phi. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.
[4] [n. d.]. PolyHJ Source Code. https://github.com/cmuq-ccl/PolyHJ.
[5] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD*. ACM, 671–682.
[6] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. 1999. DBMSs on a modern processor: Where does time go?. In *PVLDB*.
[7] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. In *PVLDB*.
[8] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*.
[9] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2015. Main-memory hash joins on modern processor architectures. In *TKDE*.
[10] Ronald Barber, G Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, G Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-efficient hash joins. In *PVLDB*.
[11] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*. ACM, 283–296.
[12] Spyros Blanas, Yinan Li, and Jignesh M Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*.
[13] Shekhar Borkar. 2007. Thousand core chips: a technology perspective. In *DAC*.
[14] Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. 2017. A study of main-memory hash joins on many-core processor: A case with intel knights landing architecture. In *CIKM*. ACM, 657–666.
[15] Jiong He, Shuhao Zhang, and Bingsheng He. 2014. In-cache query co-processing on coupled CPU-GPU architectures. *Proceedings of the VLDB Endowment* 8, 4 (2014), 329–340.
[16] John L Hennessy and David A Patterson. 2011. *Computer Architecture: a Quantitative Approach.* Elsevier.
[17] Jian Huang, Karsten Schwan, and Moinuddin K Qureshi. 2014. NVRAM-aware logging in transaction systems. In *PVLDB*.
[18] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. 2015. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *PVLDB* 8, 6 (2015), 642–653.
[19] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. ACM, 55–62.
[20] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. In *PVLDB*.
[21] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2015. Massively parallel NUMA-aware hash joins. In *IMDM*.
[22] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing main-memory join on modern hardware. In *TKDE*.
[23] Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. 2015. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*. ACM, 1493–1508.
[24] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D Nguyen, Victor W Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*.
[25] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD*.
[26] Ambuj Shatdal, Chander Kant, and Jeffrey F Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *PVLDB*.
[27] Yan Solihin. 2015. Fundamentals of Parallel Multicore Architecture.
[28] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores. In *PVLDB*.