NUMA-Aware Scalable and Efficient In-Memory Aggregation on Large Domains

Li Wang, Minqi Zhou, Zhenjie Zhang, Ming-Chien Shan, Aoying Zhou

Abstract—Business Intelligence (BI) is recognized as one of the most important IT applications in the coming big data era. In recent years, Non-Uniform Memory Access (NUMA) has become the de-facto architecture of multiprocessors on the new generation of enterprise servers. Such new architecture brings new challenges to optimization techniques on traditional operators in BI. Aggregation, for example, is one of the basic building blocks of BI, while its processing performance with existing hash-based algorithms scales poorly in terms of the number of cores under NUMA architecture. In this paper, we provide new solutions to tackle the problem of parallel hash-based aggregation, especially targeting at domains of extremely large cardinality. We propose a NUMA-aware radix partitioning (NaRP) method which divides the original huge relation table into subsets, without invoking expensive remote memory access between nodes of the cores. We also present a new efficient aggregation algorithm (EAA), to aggregate the partitioned data in parallel with low cache coherence miss and locking costs. Theoretical analysis as well as empirical study on an IBM X5 server prove that our proposals are at least 2 times faster than existing methods.

Index Terms—Aggregation, radix-partitioning, in-memory databases, cache miss

1 INTRODUCTION

Business Intelligence (BI) is becoming more and more prevailing as it helps enterprises to make valuable decisions based on their archived data sets. It is recognized as one of the most important IT applications in the coming big data era, and consequently establishes a tremendous large business market. As shown in the Gartner’s report, the market revenue for BI had reached $57 Billions in 2010 all over the world, and is projected to reach $81 Billion in 2014 [1]. In this huge market with billion dollar business values, new applications and new requirements for BI are evolving constantly. New applications, such as smart grids, smart traffic, etc., are collecting huge amount of data in every second, and seeking help from BI to analyze those big data in real time to meet their ultimate business goals. Existing systems affiliating such BI applications, including HANA, Greenplum and Aster, are emerging at an amazing pace. On the other hand, the fast advances in the computer hardware (e.g., processors with multiple cores and large capacity of memories) are showing possibilities of supporting such big data analytics from theoretical perspective. However, the architecture of modern multi-core server has evolved so quickly in recent years to deal with the well-known frequency wall, power wall [2] and memory wall [3]. Such architecture changes lead to shift from traditional memory access methods, i.e. uniform memory access (UMA) in the chip multiprocessor systems (CMP) and symmetric multiprocessor systems (SMP), to the non-uniform memory access (NUMA). NUMA architectures have now become the new standard for the processors on the new generation of enterprise servers, such that each processor contains many cores with a shared on-chip cache and an off-chip memory, and has variable memory access costs across different parts of the memory within a server.

Aggregation, such as sum, average, count, etc., is a traditional and fundamental building block in BI systems. The efficiency of aggregation is crucial to systems with critical response time requirement. In practice, aggregation operator is usually invoked on a large dataset containing millions or even billions of records. It usually works with a group-by operator, which divides the dataset into groups based on specified attributes. The cardinality of the result groups could be extremely large. In our real application with Shanghai Stock Exchange, for example, the group-by operator is executed on daily transactions based on the user accounts, involving aggregations over a huge domain with millions of stock trading participants in China.

While the increase on the number of cores on the servers is supposed to scale up the performance of such expensive aggregation operations, the truth is unfortunately depressing. We test the performance of the existing well-known parallel hash-based aggregation algorithms [4] on a NUMA server, with 8 NUMA nodes (10 physical cores per node) by integrating two IBM X5 3950s through QPI cables, under single-node and multi-node settings. In the single-node setting, the data is resident in one NUMA node and the parallel aggregation instances are bound to 10 cores within that NUMA node in round robin. In the multi-node setting, the data is
Fig. 1. The scalability comparison of the parallel aggregation algorithm.

In this paper, we focus on the evaluation of in-memory hash-based aggregation under NUMA architectures, especially for the data with large group-by cardinality. We figure out that the cache coherence miss, cache capacity miss and the locking cost are the three major factors degrading the performance of state-of-the-art aggregation algorithms in NUMA architectures. We propose a novel aggregation algorithm to leverage the high computation capability brought by NUMA architectures, while minimizing the side effects, e.g., cache miss, locking cost, remote memory access, etc., as much as possible. The contributions of this paper are listed as follows:

1) We proposed a new NUMA-affined radix partitioning algorithm which follows the characteristics of NUMA architectures and achieves both intra-node and inter-node load balance.

2) We proposed an efficient aggregation algorithm which effectively avoids the cache coherence miss and reduces the locking cost through elaborate scheduling based on range locking.

3) Theoretical analysis and extensive experimental studies show the efficiency and effectiveness of our newly proposed methods.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 discusses the characteristics of NUMA architectures and defines the problems to tackle in this paper. Section 4 proposes a NUMA-affined radix partitioning algorithm and introduces an efficient aggregation algorithm. Section 5 provides the theoretical study on our proposals. Section 6 presents the experimental results. Section 7 finally concludes the paper.

2 RELATED WORK

The early studies of aggregation focus on disk-based DBMS. [5] discusses the aggregation algorithm on uniprocessor. [6] makes a comprehensive comparison between sort-based aggregation and hash-based aggregation. Proposals in [7], [8], [9], for example, optimize the aggregation algorithm in shared-nothing multiprocessors.

With the popularity of in-memory databases and the emerging of modern many-core processors, the parallel hash-based aggregation algorithms on a single server have to be revisited. One major problem that most parallel aggregation algorithms should solve is the contention to the shared hash table from different processing instances. The state-of-the-art algorithms can be classified into two categories by their means to solve this problem. In the first category, the contention is released by allocating each parallel instance a private hash table that could fully or partially buffer the local aggregation results. One representative work is [4], which allocates a private hash table to each processing instance if the estimated contention exceeds some threshold. Another work is [10], where a generic framework is proposed to remove the bottleneck caused by the contention to the shared memory. In the aggregation evaluation under this framework, a bucket under contention will be automatically duplicated into multiple copies to release the contention. One obvious disadvantage of this kind of algorithms is the overhead of merging the redundant intermediate structure (e.g., private hash table or duplicated buckets) to get the final results, which increases linearly with the number of processing instances as well as the group-by cardinality. In the second category, the contention is reduced through data partitioning. [11] exploits aggregation in Intel Nehalem processor. The data is first partitioned on the group-by attributes(s), and then each partition is aggregated by a single processing instance to avoid the contention. Nevertheless, it fails to take the characteristics of NUMA architectures into considerations, inevitably resulting in a large amount of remote memory access in NUMA architectures. Our work differs from the works stated above in that the data partitioning is NUMA-affined and the contention to the shared hash table is avoided by elaborated scheduling rather than relying on redundant intermediate structures.

There is a large body of research on in-memory data partitioning. Radix partitioning is originally introduced in [12] to reduce the cache miss during the hash partitioning. A parallel radix partitioning algorithm is proposed in [13], where each processing instance creates a histogram to avoid the expensive synchronization overhead. [14] compares several in-memory hash partitioning algorithms in CMP and identifies that the coordinated writing to the shared output buffer is the performance issue. [15] optimizes the parallel radix partition of [13] by leveraging SIMD instruction and software prefetching.
3 Preliminaries

In this section, we introduce the preliminary knowledge used in the rest of the paper, including the NUMA architectures, the problem definition of parallel aggregation, a summary of existing solutions and the new challenges. The notations used in the rest of this paper are summarized in Table 1.

3.1 NUMA architectures and cache coherence miss

Non-Uniform Memory Access (NUMA) architecture solves the performance bottleneck of centralized memory mechanism in SMP (Symmetric Multithreaded Processor) by dividing the memory into a set of components. Each component is attached to one processor, and the components are wired by the interconnect network (e.g., Intel QPI[16]) across the processors, as shown in Fig. 2. One processor associated with its local memory is called a node. Cores within one node access the local memory with larger data transmission bandwidth and lower access latency, significantly better than those of the inter-node remote memory access.

TABLE 1

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>a tuple of the input table</td>
</tr>
<tr>
<td>r,k</td>
<td>the group-by values of tuple r</td>
</tr>
<tr>
<td>r,v</td>
<td>the aggregation values of tuple r</td>
</tr>
<tr>
<td>N</td>
<td>the number of NUMA nodes</td>
</tr>
<tr>
<td>node_i</td>
<td>the i^{th} NUMA node</td>
</tr>
<tr>
<td>M</td>
<td>the number of cores per NUMA node</td>
</tr>
<tr>
<td>T^n</td>
<td>the t^{th} parallel instance running on node^n</td>
</tr>
<tr>
<td>h(·)</td>
<td>the hash function used for the radix partitioning and aggregation</td>
</tr>
<tr>
<td>B</td>
<td>the number of binary bits representing the result of h(·)</td>
</tr>
<tr>
<td>P</td>
<td>the total number of passes in the radix partitioning</td>
</tr>
<tr>
<td>b_p</td>
<td>the number of bits for the p^{th} pass</td>
</tr>
<tr>
<td>K</td>
<td>the total number of bits used for the radix partitioning, e.g., K = \sum_{p=1}^{P} b_p</td>
</tr>
<tr>
<td>s</td>
<td>the number of partitions generated by each node, e.g., s = 2^K</td>
</tr>
<tr>
<td>C^n</td>
<td>the j^{th} partition generated by node^n</td>
</tr>
<tr>
<td>R(C^n_j)</td>
<td>the range of C^n_j</td>
</tr>
<tr>
<td>D^n_j</td>
<td>the set of unprocessed data chunks assigned to parallel instance T^n_j.</td>
</tr>
</tbody>
</table>

TABLE 2

<table>
<thead>
<tr>
<th>Source</th>
<th>The position of valid cache line</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L1 cache</td>
</tr>
<tr>
<td>Local</td>
<td>4</td>
</tr>
<tr>
<td>Local core (on die)</td>
<td>83</td>
</tr>
<tr>
<td>Remote core (QPI 1 hop)</td>
<td>280</td>
</tr>
<tr>
<td>Remote core (QPI 2 hops)</td>
<td>351</td>
</tr>
</tbody>
</table>

In the multicore architecture, two or more cores may have data copies with the same memory location in their private caches. To keep a consistent view on the data, most modern processors apply certain invalidation coherence protocols (e.g., MESI, MESIF, etc. [2]) on the hardware level. Assume that core_i updates the data, the hardware coherence controller notifies core_j to invalidate the in-cache copy of the data to prevent dirty read. Consequently, when core_j tries to read the same piece of memory later, cache coherence miss occurs, which takes additional efforts (cache coherence miss penalty) to retrieve the latest version of the data. The amount of the penalty depends on 1) the NUMA distance between core_i and core_j; and 2) the level of the cache accommodating the latest version. Table 2 summarizes the measured cache coherence miss penalties by using the same methodology as that in [17] on a server with 8 NUMA nodes, which is integrated with two IBM x5 3950s through QPI cables.

Given the numbers in the table, we conclude that the remote memory access is much more expensive than local memory access. This observation motivates us to redesign the parallel aggregation algorithm for NUMA architectures, to minimize the number of synchronization on data copies across different nodes.

3.2 Problem Statement

In aggregation evaluation, tuples are grouped according to the value(s) on a given set of attributes, namely group-by attributes, and for each group a given set of aggregation functions (e.g., sum, max, min, avg, count, etc.) are each evaluated on the specific attributes, namely aggregation attributes. The number of groups is termed as group-by cardinality. In this paper, we focus on hash-based aggregation. The hash function used during the aggregation is denoted as h(·). For each tuple r of the input table, r,k denotes the values on group-by attributes and r,v denotes the values on the aggregation attributes.

On the hardware side, we assume there are N NUMA nodes on the server, named as node_0,..., node_{N-1}. The data for aggregation are evenly placed on these N NUMA nodes. Moreover, the data chunks related to a single tuple are always kept in the memory of a single node. Finally, we also assume that each node_i contains the same number, say M, of physical cores, and each core runs exactly one instance/thread in any of the algorithms.

In the rest of the section, we discuss the major challenges to parallel aggregation and data partitioning,
under the new problem setting of NUMA architectures and large group-by cardinality.

### 3.3 Challenges to Parallel Aggregation

Three parallel algorithms are proposed in [4] for hash-based aggregation. **Shared aggregation** allows parallel instances to share a global hash table, and parallel updates are serialized by executing locks. **Independent aggregation** allocates a private hash table for each parallel instance to store its aggregation results individually, and finally merges all the private hash tables to complete the aggregation. **Hybrid aggregation** assigns a private hash table of specific size (based on the cache size) to each parallel instance, to maintain the partial aggregation results.

When overflow happens to the private hash table, the additional aggregation results are directly flushed into the global hash table.

The performance of all these methods under our problem setting, with NUMA architectures and large group-by domain cardinality, is not promising. Independent aggregation is effective in reducing the thread-level contention on the global hash table. But it results in oversized private hash tables and substantial merging costs on the parallel instances, when the group-by domain is too large. Similarly, hybrid aggregation works well on dataset following skewed distributions, since a small number of buckets are sufficient to aggregate most of the tuples. When the group-by domain grows, more distinct keys appear in the hash tables due to the long-tail effect, leading to lower cache hit rate on the private hash tables. Finally, shared aggregation is more suitable for aggregation on large domains, because there is no overhead incurred by private hash tables and relatively low thread-level contention to the global hash table [4].

However, as shown in Fig. 1, shared aggregation fails to effectively leverage the massive computation capability of NUMA architectures. To gain better insight into the underlying reasons, we further analyze the CPU cycles by testings on group-by domains of two different sizes, i.e. 256K and 1M respectively. The results in Fig. 3 imply that only a very small fraction of the CPU cycles are spent on data reading and hash value calculation, while the others are all linked to locking, cache coherence miss and cache capacity miss. In the following, we provide a detailed analysis on the sources of these overheads.

**Locking**, implemented by “chx” instruction in Spinlock, takes at least 200 cycles. Lock is used to protect any two parallel instances from concurrently updating the same hash bucket. When more parallel instances are running, it will be more likely for an instance to wait for the release of the lock from others and consequently the locking overhead increases.

**Cache coherence miss** is another important reason behind the waste of CPU cycles. When the group-by cardinality is 256K, hash table takes 16MB space and is entirely resident in 30MB L3 cache during the aggregation. Unfortunately, when parallel instances running on all of the 8 NUMA nodes concurrently update the hash table, most accesses to the hash table result in cache coherence misses rather than cache hits. Even worse, the cache coherence miss penalty increases with the number of parallel instances, due to the bottleneck of the snooping protocol [18].

**Cache capacity miss** is unavoidable for aggregation with large group-by cardinality because the hash table is too large to be entirely kept in cache. The hash table to support the maximal cardinality which resides in the 32 KB L1 cache (or 256KB L2 cache, 30MB L3 cache) is 512 (or 4K, 512K). The larger the cardinality is, the more cache capacity misses will be. Also, cache capacity miss penalty increases with the number of parallel instances due to the contention to memory access.

### 3.4 Challenges to Data Partitioning

The cache capacity miss problem stated above can be nicely solved by hash-partitioning the input data on the group-by attributes such that the total size of the hash buckets corresponding to all keys in a single partition is small enough to reside in cache. However, the hash partitioning process becomes the performance issue if the number of partitions exceeds the number of cache-lines and TLB entries. The well-known solution to this problem is the radix partitioning algorithm proposed in [12], which iteratively partitions the dataset rather than in a single pass to retain the good cache behavior.

When running the radix partitioning algorithm in parallel on the modern multicore computers, instances on different cores may insert tuples into the same partition concurrently. This phenomenon introduces thread level contention to the partitions and consequently results in performance degradation. Algorithm proposed in [13] avoids this contention by calculating a histogram before partitioning, by which each parallel instance knows its dedicated position for partitioning, including its start point and its length. Hence, the cost for instance synchronization is effectively reduced. [15] further optimized the parallel radix partitioning by using SIMD and software pre-fetching. However, these algorithms do not consider the characteristics of NUMA architectures, causing unnecessary remote memory accesses with high overhead. Furthermore, none of them has addressed the load balancing problem, which is a key factor to the performance of partitioning with many cores.
4 NUMA-Aware Partitioning and Aggregation

To avoid the cache capacity miss when accessing hash table in aggregation phase, we first split the input tuples into multiple partitions based on the most significant K bits of hash value such that the portion of hash table corresponding a single partition is small enough to reside in cache. Then we aggregate the partitioned data (based on the same hash function as the partitioning phase) using our novel efficient aggregation algorithm, which can effectively reduce the locking cost and cache coherence miss.

4.1 NUMA-Aware Radix Partitioning

Our NUMA-aware radix partitioning (NaRP) extends and improves upon the existing methods proposed in [13] and [15] by including new optimizations based on the characteristics of NUMA architectures. We also maximize the parallelism in partitioning to balance intra-core workload and intra-node workload, which results in a dramatic reduction on partitioning latency.

To limit the negative effects of remote memory access as well as the expensive communication between NUMA nodes, our NaRP method requests each NUMA node to generate $s = 2^K$ radix partitions independently and locally. Local parallel instances running on a NUMA node are responsible for partitioning the local data and maintaining the intermediate partitioning results in local memory. There is no data sharing or communication across NUMA nodes.

Generally speaking, the overall structure of the new parallel partitioning method follows the standard radix partitioning algorithm, as introduced below. The input data kept with $node_n$ are split into $M$ slices, each of which consists of a group of data chunks. Each slice is then taken and processed by one parallel instance at a time. Note that all the parallel instances running on $node_n$ share a common output partition pool, allocated in a consecutive memory block. The partitioning on each instance is accomplished by running $P$ passes on the input data slice, with integers $\{b_1, b_2, \ldots, b_P\}$ as the controlling parameters for the passes. If it is the first pass, the algorithm takes the original data slice as the input, otherwise the outputs of previous pass are fed into the algorithm. In the $p^{th}$ pass, the algorithm generates the results with $2^{bp}$ partitions. Before the start of processing for each pass, each of the instances calculates the memory address of its output pool, by synchronizing with other instances on the same NUMA node. Following the strategy used in [13], to reduce the synchronization cost, each parallel instance $T^n_i$, i.e. the $i$th instance running on $node_n$, first counts the number of new tuples for each partition in the coming pass by scanning the assigned input data and computing the destination partition for each tuple. After receiving the counts from all other instances, every instance is capable of initializing writing cursors in the memory without any conflict with other instances. Instance $T^n_i$, for example, positions its insert cursors, say $I^{T^n_i}$, which is an array with length $2^{bp}$. The $j$th entry of the array indicates the starting position for instance $T^n_i$ to insert the result tuples into the $j$th output partition. After finishing all $P$ passes, each $node_n$ generates $s = 2^K = 2^{\sum_{p=1}^{P} bp}$ partitions, denoted by $C^0_n, \ldots, C^{s-1}_n$. All data in the same partition share exactly the most significant $K$-bits of hash values on their group-by attributes.

While the NUMA-aware radix partitioning can be highly parallelized, the result performance of the method largely depends on the degree of load balancing between the parallel instances. There are two potential problems causing imbalance of the workload, i.e. skewness of the data distribution and performance variance of the cores. Since both problems are difficult and expensive to detect and predict, we believe it is nearly impossible to achieve offline load balancing. Instead, we target to alleviate the problem by adopting online load balancing strategies, based on the characteristics of NUMA architectures.

Intra-node Load Balancing: In the first pass, an important operation for each parallel instance is counting the number of tuples to be inserted into the result partitions and calculating the starting memory address of the insert cursors. Such calculation needs coordination and synchronization among the instances, since the decisions are made based on the volume of data to partition for each parallel instance. As a result, once the calculation of insert cursors is finished, the data slice is assigned to the corresponding instance. For a particular $node_n$, we use $T^n_i$ to denote one of the instances finishing all the data chunks, and use $T^n_u$ to denote an instance still running with unfinished data chunks, which are represented by $D^{T^n_u} = \{D^{T^n_u}_f\}$. The straightforward solution to workload migration is supporting $T^n_i$ to partition the remaining data chunks $D^{T^n_u}$ together. This solution involves sharing and frequent synchronization on the insert cursors $I^{T^n_u}$ of the instances for inserting every tuple, as demonstrated in Fig. 4(a). There are two major disadvantages with this straightforward solution, including (1) the concurrent updates to the insert cursors $I^{T^n_u}$ not only impose additional locking overhead but also lead to high cache coherence penalties; and (2) interleaved insertions from $T^n_i$ and $T^n_u$ on the same cache line of the partition cause false sharing [19] and result in substantial cache misses.

To overcome the limitations of the straightforward solution, we propose the Pre-Allocation Strategy (PAS), with a running example shown in Fig. 4(b). The intuition behind PAS is to allow an instance $T^n_i$ to pre-allocate a block of memory for writing, from the bottom of the result partitions. After the pre-allocation, instance $T^n_i$ partitions $D^{T^n_u}$ independently for instance $T^n_u$. The details of the method is described in Algorithm 1. Particularly, instance $T^n_i$ scans $D^{T^n_u}_f \in D^{T^n_u}$ to calculate the target partition for each tuple and sums up the exact insertion counts $hist_i$, i.e. the number of tuples in $D^{T^n_u}_f$
to be inserted into each partition. Based on the insertion counts, instance \( T^n_u \) pinpoints the block of memory for writing in each pre-assigned partition \( C^n_j \), by calculating the offset from the bottom of the memory pool. If there are \( \text{hist}_i[j] \) tuples, for example, it sets the insert cursor \( I_i[j] = \text{end}[j] - \text{hist}_i[j] \) for \( D^T_u \), where \( \text{end}[j] \) is the bottom of the memory pool for \( C^n_j \). Since the cursor is started to write the bottom part of the pool, it does not affect the data partition processing run by instance \( T^n_u \). The strategy even supports two or more parallel instances to take over the data chunks of \( T^n_u \) to partition, by setting up a lock on the pre-allocation process.

It is worthwhile to emphasize other benefits of PAS to data partitioning. First, \( T^n_j \) and \( T^n_u \) do not need to communicate with each other, or to write to the same part of the memory pool. Consequently, they do not interfere with each other in terms of both workflow and efficiency. Second, if more instances take over the workloads from \( T^n_u \), the partitioning process is independent of each other, after the computation of the insert cursors. Third, the cost of calculating insert counts in \( T^n_j \) for pre-allocation is very efficient, as SIMD instructions are employed and sequential memory access of the original data could leverage hardware cache pre-fetching. Also, if the chunk size is appropriate (e.g., smaller than the private L1 or L2 cache), the data chunk loaded into the private cache during the first scan can be entirely kept in cache and reused by the subsequent partitioning operations.

In the passes except the first one, as the repartitioning of each input partition from previous round is independent, we use task queue model to improve the load balancing between instances [13]. In the task queue model, each input partition is modeled as a job and all input partitions are kept in a job queue. An instance picks up the first partition in the job queue, finishes the partition before his next job pickup at the queue. Unfortunately, the task queue model fails to achieve balance on the workload when the data distribution is highly skewed, e.g., when the input relation follows heavy hitter or Zipf distribution. In such cases, data partition containing the most frequent group-by keys are much more time-consuming than others, and the instance responsible for such a data partition, say \( T^n_u \), is much slower than the other instances. Assume that \( D^T_u \) are the set of unprocessed data chunks in the data partition assigned to \( T^n_u \). We tackle the problem by requesting another idle instance \( T^n_j \) to take over the data chunks from \( D^T_u \), one at a time. As the data chunks from a partition are divided into the result partitions, \( T^n_j \) shares the insert cursors with \( T^n_u \), with PAS employed to reduce the synchronization overhead.

**Inter-node Load Balancing:** The PAS strategy and task queue model can be easily extended to inter-node load balancing by allowing the parallel instances in the NUMA node to help other NUMA nodes after they have finished the processing of all the local data. To minimize the remote memory access and the synchronization overhead, cross-node workload migration is enabled, only when one NUMA node has finished its radix partition for all the passes. The remote memory access penalties during inter-node balancing can be further reduced by the following strategies. First, let the idle NUMA nodes prefer to take over the workload from the nearest nodes (with 1 hop NUMA distance). Second, before \( T^n_u \) partitions the assigned data chunk into the remote pre-allocated partitions, it first feeds the partitions into local cache through efficient sequential scan such that the subsequent writes to those partitions can avoid expensive remote random memory access.

### 4.2 NUMA-Aware Parallel Aggregation

This part of the section presents the first NUMA-aware parallel aggregation algorithm, to maximize the utility of multiple cores under NUMA architectures. To begin with, we first introduce the definition on the range over hash value domain.

In our NaRP method, tuples on each NUMA node are partitioned into \( 2^K \) partitions based on the most significant \( K \) bits of the hash value. Given the number of
bits for hash value representation, $B$, tuples in a single partition contains up to $2^{B-k}$ distinct hash values.

Definition 1: The range of the partition $C_j^n$ on node$_j$, denoted as $R(C_j^n)$, is the set of all the possible hash values that the tuples in $C_j^n$ could have by applying the hash function $h()$ on their group-by attributes.

After running NaRP for partitioning, there are $s = 2^K$ distinct ranges. For $C_j^n$, the $j^{th}$ partition on node$_j$, its partition index $j$ indicates its range. Partitions with the same index have the same range, i.e., $R(C_j^n) = R(C_{j'}^n)$ for any $j = j'$. For instance, let $B = 5$, $K = 3$ and $N = 4$, $R(C_3^0) = R(C_3^1) = R(C_3^3) = R(C_3^4) = \{01100, 01101, 01110, 01111\}$. In contrast, partitions with different partition index have different ranges, i.e. $R(C_j^n) \cap R(C_{j'}^n) = \emptyset$ for any $j \neq j'$.

During the parallel aggregation, when two or more parallel instances running on different NUMA nodes are aggregating the partitions of the same range, they are supposed to process tuples with the same group-by values and hence update the same hash buckets of the global hash table. This not only leads to cache coherence miss but also requires locking on corresponding hash bucket during the aggregation processing on each individual tuple.

We first consider two straightforward baseline solutions to avoid the cache coherence miss and locking cost as following. The first method, namely range based aggregation (RBA), forces the partitions with overlapping ranges to be aggregated by a single instance, which is similar to the method proposed in [11]. As all partitions with the same range will be processed by a single parallel instance, locking is not necessary. Unfortunately, this method introduces expensive remote memory accesses in NUMA architectures. Another method, namely node independent aggregation (NIA), allocates each NUMA node a private hash table and lets each parallel instance aggregate on the local private hash table, which will be merged together finally. As the ranges within each NUMA node do not overlap, providing that an entire partition will be aggregated by a single instance, the cache coherence miss is avoided and locking is not needed. However, the overhead of merging private hash tables is very expensive for large group-by cardinality.

To discount the disadvantages of RBA and NIA, we propose Efficient Aggregation Algorithm (EAA), based on global hash table but without partition movement between NUMA nodes. In the following, we summarize the strategies used behind our Efficient Aggregation Algorithm (EAA).

Adoption of Global Hash Table: Our EAA only uses a single global hash table, since the merging of private hash tables under our problem setting is too expensive to afford. But we have new optimization techniques to minimize the cache coherence misses and locking cost.

Range Locking: EAA applies range locking, in which there are $2^K$ global mutex locks, namely range locks, shared by all the parallel instances. The $j^{th}$ range lock corresponds to the range that the $j^{th}$ partition on any node has.

Before any instance aggregates a partition $C_j^n$ into the global hash table, it should first obtain the $j^{th}$ range lock to prevent other parallel instances from aggregating the partitions with the same range. This strategy avoids the chances of cache coherence misses when accessing the hash table. Furthermore, once the range lock for the partition is obtained, no other locking is needed during aggregating the entire partition and hence the locking cost is significantly reduced.

Reduction on Remote Memory Access: The parallel instances only process the local partitions, which are output from the data partitioning. To further reduce the cost of remote memory access when updating the global hash table, before aggregating a partition, our method fetches the involved hash buckets into CPU cache through sequential read, and then consecutively aggregates the entire partition (all the local data corresponding to these buckets) before these buckets are swapped out of cache.

Non-Blocking Instances. In range locking, locks on a large range probably degrade the performance on concurrency. EAA does not block a parallel instance when it fails to acquire the lock for a the range of a partition. Instead, the parallel instance moves to another partition immediately. Also, a local buffer is used to retain the non-blocking property when the data is highly skewed.

Algorithm 2: Efficient aggregation

Input: $ht_g$, $C^n = \{C_0^n, ..., C_s^n\}$, $rl_g = \{rl_0, ..., rl_{s-1}\}$.

$LB \leftarrow \emptyset$, randomly reorder $C^n$;

while $C^n \neq \emptyset$ do

foreach $C_j^n$ in $C^n$ do

if $rl_j$.try_lock() is successful then

$C_j^n = C_j^n / C_j^s$;

warm up the cache;

aggregate $C_j^n$ into the in-cache buckets;

rl$_j$.release_lock();

f_count $\leftarrow 0$;

else

f_count $\leftarrow f_count + 1$

if f_count $\geq \delta$ then

aggregate $C_j^n$ into $LB$;

f$\leftarrow 0$;

flush $LB$ to $ht_g$;

Based on the strategies above, we hereby introduce the complete working mechanism of EAA. The pseudocodes for every parallel instance, say $T_i^n$ on node$_i$, are presented in Algorithm 2. The inputs are the global hash table (shared by all the parallel instances) and $C^n$, a set of local radix partitions (shared by local instances on node$_i$), as well as an array of globally shared mutex locks, rl$_g$, in which each $l_j$ in the array corresponds to the range of the $j^{th}$ partition. $T_i^n$ keeps aggregating the local partitions in $C^n$ in a random order, until all partitions in $C^n$ are completed. $T_i^n$ acquires a lock to the
range of \( C^n \) before aggregating any \( C^n_j \in C^n \) to prevent other parallel instances from aggregating the partitions with the same range. If the range lock is successfully acquired, \( T^n \) removes \( C^n_j \) from \( C^n \), warms up the local CPU cache by reading the hash buckets corresponding to the range, and aggregates \( C^n_j \) by updating the in-cache hash buckets. When the range lock is obtained, other parallel instances are not allowed to access the same buckets of the global hash table until the the range lock is released. By choosing a proper number of bits for radix partition, we can guarantee that the total size of the hash buckets for a single range is small enough to load into the cache. Note that the warmup is extremely efficient by simply arranging the hash buckets in consecutive memory space. If the instance fails to acquire the lock for a particular range, \( T^n \) immediately switches to aggregate next partition in \( C^n \), again by random pickup.

In most of the cases, a parallel instance always finds an unlocked partition to aggregate (See Section 5 for theoretical analysis). However, if the data are highly skewed, the size of the “fat” partitions that contain the most frequent keys may be very large. The more skewed the input data is, the more likely that the “fat” partitions are the last few (fewer than the number of parallel instances) unprocessed partitions on every NUMA node. As a result, some parallel instances may suffer starving problem, unable to acquire the lock to any range of the remaining partitions waiting for processing. To detect such event without significant overhead, a variable \( f\_count \) is maintained to count the number of failures of \( T^n \) on acquiring locks. If \( f\_count \) reaches a pre-defined threshold \( \delta \), e.g., \( \delta = |C^n| \), the algorithm removes \( C^n_i \) for \( T^n_i \) from \( C^n \) and aggregates \( C^n_j \) into the local buffer \( LB \), which is finally merged into the global hash table with locking applied on the hash table buckets to guarantee the thread-safety. Note that our algorithm is different essentially from the NIA in that only a very small fraction of the partitions are aggregated into the local buffer. Our theoretical and empirical evaluations in Section 5 and Section 6.3 verify this claim.

4.3 Parameter Selection

To avoid the performance degradation caused by hash conflicts in the aggregation phase, the number of binary bits \( B \), which is used to represent the hash function results and in turn determines the number of hash table buckets, is selected such that \( 2^B \) is no smaller than the upper bound of the estimated group-by cardinality. The estimation can either be obtained from the query optimizer or through sophisticated sampling techniques, e.g., [20]. The number of bits \( K \) used for radix partition guarantees that the total size of hash buckets corresponding to a partition (e.g., \( 2^{B-K} \) buckets) is smaller than L1 cache size for cache miss avoidance. If \( 2^K \) is too large for one pass partitioning, multiple passes partitioning is called to ensure that the “fanout” for each pass does not exceed the number of L1 cachelines and TLB entries.

Note that when the group-by cardinality of the input data is very low (e.g., several thousand or less), the entire hash table can reside in cache and radix partitioning phase is omitted. Thus, in EAA, the input data on each NUMA node is considered to be of the same range covering all the hash value domains, and all parallel instances but one will fail to obtain the range lock and aggregate the results into their respective local buffers. In such cases, our EAA degenerates and becomes identical to private aggregation approach, which is reported as the most efficient method for low group-by cardinality[4].

5 Theoretical Analysis

In this section, we mathematically model the performance of the NUMA-aware algorithms proposed in previous section. The theory derived in this section helps us to verify the superiority of our proposals. In particular, we aim to show that our proposed aggregation algorithms quantitatively reduce the cache coherence miss and locking cost as well as the overhead of the local buffer.

Theorem 1: In EAA, no cache coherence miss and cache capacity miss will be introduced when aggregating the partitions into the global hash table.

Proof: First, after radix partitioning, it is guaranteed that hash buckets corresponding to a single partition can be entirely resident in cache, and hence no cache capacity miss will happen. Second, because of the range exclusivity property in EAA, any two parallel instances never concurrently update the same hash buckets. The algorithm thus does not incur any cache coherence miss when accessing the global hash table.

In our efficient aggregation algorithm, the number of partitions aggregated into the local buffer decides the performance, as initialization and the flushing of the local buffer cost additional overheads and will introduce cache miss and locking cost. While it is unlikely to provide a completely accurate estimation on the number of such partitions, we conduct some upper bound analysis on the expected number. The upper bound is derived based on the necessary but not sufficient condition on aggregating a partition into local buffer, i.e., a partition is aggregated into the local buffer only when a parallel instances fills to acquire the range lock on the partition. Such failure happens when the ranges crash occurs on at least two parallel instances.

Definition 2: A range crash happens when two instances try to aggregate the partitions with the same range at the same time.

Based on the definitions, we aim to estimate the upper bound on the expected number of range crashes. To simplify the theoretical analysis, we make the following assumptions on the behavior of the NUMA nodes. Firstly, each parallel instance takes exactly the same time to finish a partition, regardless of the host NUMA node. Second, NUMA node accesses their partitions in completely random and independent orders. In the
Lemma 1: A range crash happens to a NUMA node node\(_n\) only if there are \(MN\) partitions waiting for aggregation, in which \(M\) is the number of instances and \(N\) is the number of nodes.

Proof: Each NUMA node node\(_n\) tries to aggregate to at most \(M\) partitions at any time. Consider all the other nodes except node\(_n\). They are writing to \(M(N-1)\) partitions at the same time. It means that when there are more than \(M + M(N-1) = MN\) partitions to write, node\(_n\) can always find \(M\) partitions without range conflict to write. This completes the proof of the lemma.

The lemma above leads us to the main theorem on the estimation.

Theorem 2: The expected number of range crashes is no larger than \(\frac{2M^2N^2}{s}\), in which \(s\) is the total number of partitions generated on each node during data partitioning.

Proof: To prove the theorem, we first estimate the number of range crashes happening on each NUMA node. Based on Lemma 1, we only need to consider the last \(MN\) partitions, based on the random order on the partitions generated for the particular node. Without loss of generality, we assume these partitions are \(\{C_1^n, C_2^n, \ldots, C_{MN}^n\}\). If we use \(F_j\) to denote the event of range crash occurring to partition \(C_j^n\), the expected number of range crashes on node\(_n\) is equivalent to \(\sum_{i=1}^{MN} Pr(F_j)\), in which \(Pr(F_j)\) is the probability of the event \(F_j\). Because of the symmetry property such that all partitions are exchangeable in the probability space, we only need to estimate the probability of any single event.

For a particular \(F_j\), range crash happens between \(n\) and \(n'\), only if \(C_j^n\) is scheduled close to \(C_j^{n'}\), such that the scheduled order on \(n\) and \(n'\) is no larger than \(2M\). The probability of such scheduling is no larger than \(\frac{2M}{s}\). Since there are \(N - 1\) candidate nodes for range crashes against node \(n\), the total probability of range crash with respect to \(C_j^n\) is no larger than \(\frac{2M(N-1)}{s}\).

Finally, we combine all the partitions and all the nodes, by summing up all the probabilities. The final upper bound on the expectation of range crash is upper bounded by \(\frac{2M^2N^2}{s}\).

In the rest of this section, we analyze the efficiency of range locking. In EAA, a parallel instance does not wait if it fails to obtain a range lock but continues to try another partition. When there are no range locks available, it will aggregate the partition into the local buffer. Consequently, range locking will not block any parallel instance and hence does not decrease the concurrency. This means the overhead of range locking only depends on the total number of attempts to acquire the range locks.

Theorem 3: Let \(L\) be the total times of attempts made by all the parallel instances in EAA to acquire the range locks, then \(E(L) \leq N \times s + \frac{2M^2N^2}{s}\).

Proof: In \(L\) attempts to acquire the range lockings, we denote \(L_s\) to be the times of successes and \(L_f\) to be the times of failures. Apparently, \(L = L_s + L_f\). For \(L_s\), consider that a parallel instance will fail to obtain the range lock if and only if range crash happens, hence \(L_f\) is equal to the number of range crashes and \(E(L_f) \leq \frac{2M^2N^2}{s}\) according to Theorem 2. Once a parallel instance succeeds to obtain the range lock, a partition will be aggregated into the global hash table. Also, there are totally \(N \times s\) partitions on all the NUMA nodes and some partitions might be aggregated into local buffer. Thus, \(L_s \leq N \times s\) holds. Finally, \(E(L) = E(L_s + L_f) \leq E(N \times s + L_f) = N \times s + E(L_f) \leq N \times s + \frac{2M^2N^2}{s}\).

Comparing with the algorithms (e.g., shared aggregation) which employ locking on aggregating every individual tuple and hence call lock() as many times as the number of input tuples, EAA only calls lock() for much fewer times.

6 EXPERIMENT AND EVALUATION

6.1 Experimental Setup

The evaluation is on a server equipped with two IBM X5 3950s connected by QPI cables. The details of the hardware specifications are provided in Table 3.

We use both real dataset and synthetic datasets in the experimental evaluations. The real dataset includes transactional records from Shanghai Stock Exchange. We extract the buyer IDs (4-byte-integer) and stock IDs (4-byte-integer) from the transactions as the group-by attributes, and the transaction volumes (8 byte long integer) as the summation/maximum aggregation attribute. Such aggregation tasks are commonly used in the trading analysis. The real dataset has over 800 million records and 200 million distinct groups, roughly following power-law distribution. To verify the scalability of the algorithms, we sample the transaction table under different sampling rate to generate variant datasets with desirable group-by cardinalities. We also generate synthetic datasets in our experiments. The synthetic dataset is based on a schema with an 8-byte integer as the group-by column and an 8-byte integer as the aggregation target. We employ 3 different distributions when generating the group-by attributes of the synthetic records, including uniform, Zipf and heavy hitter. In Zipf distribution, the exponent is set to be 0.5. In heavy hitter distribution, half of the tuples share the same value in the group-by attribute, while the values of other half follow uniform distribution. For both the real data and synthetic data, each table contains at least 1 trillion

<table>
<thead>
<tr>
<th>TABLE 3</th>
<th>Hardware specifications of the server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of NUMA nodes</td>
<td>8</td>
</tr>
<tr>
<td>Number of Cores per NUMA node</td>
<td>10</td>
</tr>
<tr>
<td>Size of DRAM per NUMA node</td>
<td>32 GB</td>
</tr>
<tr>
<td>L1 cache size(private)</td>
<td>32 KB</td>
</tr>
<tr>
<td>L2 cache size(private)</td>
<td>256 KB</td>
</tr>
<tr>
<td>L3 cache size(shared within a NUMA node)</td>
<td>30 MB</td>
</tr>
</tbody>
</table>
records. Tables are evenly and horizontally partitioned and loaded into the memory on each NUMA node. The columns of each table are separately stored and split into data chunks of 16KB. The aggregation query used in our experiments is formulated as follows.

\[
\text{SELECT key, SUM(value), MAX(value) FROM T \text{ GROUP BY key;}}
\]

The experimental programs are written in C++, compiled by gcc with -O3 optimization flag and run on Linux SUSE 11 system. CPU cache miss rate and QPI traffic are measured by using Intel Performance Counter Monitor. We use libnuma to allocate memory on specific NUMA nodes and employ APIs in "sched.h" to bind the parallel instances with specific cores. Without otherwise clarification, the algorithms are run with 8 NUMA nodes, and each parallel instance is bound exclusively with a unique core.

We use modulus operator in our hash function implementation. The modulus base is a power of 2, such that the efficient bit shift operations could be employed to calculate the hash value. The buckets of the hash table are 64-byte-aligned to avoid false sharing and are allocated in consecutive memory to reduce TLB misses. We assume that the estimated group-by cardinality is known to the query optimizer. The domain of the hash function is thus adjusted to be larger than the known group-by cardinality, to avoid the hash table overflow. SIMD instructions are used to optimize the radix partitioning. We employ spin-lock based on "chx" instruction as lock implementation, as we find spin-lock is more efficient than other locks in the scenario of in-memory aggregation, where lock is used frequently and is held for a short period of time (several CPU cycles).

6.2 NUMA-aware Radix Partitioning

We first evaluate the performance of the load balancing mechanism in our NUMA-aware radix partitioning (NaRP), and compare it against two state-of-the-art parallel radix partitioning algorithms.

The radix partitioning algorithm is insensitive to the radix bits \( b_p \) for each pass [15], as long as the radix bits is smaller than \( 8 \) bits (the fanout of 8 bits is 256, which is smaller than the number of cache lines in L1 cache and the number of second level TLB entries). In the rest of the experiments, we adopt \( b_1, b_2, \ldots, b_p \leq 8 \) and pick up them appropriately to minimize the number of passes.

**Load Balancing Evaluation:** Fig. 5(a) shows the performance of NaRP without load balancing strategy, running on the dataset under uniform distribution and the data set under heavy hitter distribution respectively. For graphic reasons, we only use 4 NUMA nodes and run 16 parallel instances (4 instances per node) in this group of experiments. In Fig. 5, each parallel instance with index \( i \) is bound to the \( [i/4] \)-th NUMA node. The results in Fig. 5(a) show that the workload is imbalanced on both of the distributions, although each parallel instance processes the same amount of data. As we expected, this is due to the competition on the CPU resources between our parallel instances and other system-level processes. It is interesting to find that the workload in the second pass is much more skewed in the heavy hitter distribution. That is because the parallel instances 2, 6, 10, and 14 are responsible for handling “hot” values generated by the heavy hitter distribution.

We run NaRP again with our dynamic load balance strategy (Pre-Allocation Strategy and task queue model) on the same data. Fig. 5(b) demonstrates a much balanced performance between the instances. It shows that our workload migration mechanism works well, so that instances finishing earlier effectively help the other lagging instances.

We also evaluate the throughput of a single parallel instance during both intra-node and inter-node load balancing phases. The results are shown in Table 4. Note that the control overhead in PAS is also included. In both intra-node phase and inter-node phase, \( T_u \) refers to the parallel instance lagging behind, while \( T_f \) refers to the instance finishing earlier and acquiring additional workload from \( T_u \). \( T_f \) and \( T_u \) are running on the same NUMA node in intra-node phase and on different NUMA nodes in inter-node phase. To make better comparison, a normal phase is added in the table where a single parallel instance is partitioning its own data alone. Table 4 clearly shows that the performance of both \( T_f \) and \( T_u \) degrades dramatically by a margin larger than 26M tuples per second. When using the naive strategy (sharing the insert cursors), we find that the concurrent updates to the insert cursors and the output buffer introduce expensive locking costs and cache coherence miss penalties. In PAS, in contrast, due to our elaborate design, \( T_u \) is not aware of the existence of \( T_f \) in terms of both working flow and efficiency. This helps \( T_u \) to keep the same
performance as the normal phase. Note that $T_f$ is only slightly slower than $T_u$, although $T_f$ needs to scan the input data chunk once more for pre-allocation. This is because the data chunks entirely loaded into L1 cache in the first scan can be reused without cache refreshing in the subsequent scan, if the chunk size is smaller than the L1 cache. Benefit from the tricks of reducing remote memory access penalty, the throughput of $T_f$ in intra-node load balancing is only 4.2M tuples/s slower than that in intra-node load balancing.

**Comparison with State-of-the-art Algorithms:** We compare our NUMA-aware radix partitioning (NaRP) with two state-of-the-art parallel radix partitioning algorithms: concurrent radix partitioning (CRP)[14] and histogram-based radix partitioning (HbRP) [13]. In CRP, the parallel instances share the same memory for output and hence do not need to calculate histograms before partitioning but have to coordinate with each other when inserting every tuple. HbRP is similar to CRP, except that insert cursors are calculated in advance to reduce the synchronization cost.

We run the 2-pass radix partitioning with the three algorithms on the uniform distributed data with 1M unique group-by attributes. The throughput comparison of the three algorithms in Fig. 6(a) shows that our NaRP is at least 2 times faster than the other two algorithms. To better illustrate the reason behind the advantage, we present the breakdown of CPU cycles when partitioning each tuple in the 2nd radix pass in Fig. 6(b). The consumption of CPU cycles is categorized into three parts: histogram calculation (optional), coordinating with other parallel instances, and partitioning the input tuples to the output buffers. Clearly, CRP is inefficient as too many cycles are wasted on coordination. In contrast, beneficial from the strategy of insert cursors, the coordinating cost in HbRP and NaRP is ignorable. Histogram calculation, involving scan on the input data, in NaRP is nearly 2 times faster than BbRP, since the scan in NaRP is completely done on the local memory. Note that although the remote memory scan latency is largely reduced by hardware-level prefetching, the remote memory access demand from a large number of instances makes the bottleneck in QPI. Moreover, memory writing in the partitioning is mainly done in a random fashion. Therefore, the remote memory access latency cannot be effectively subsidized by prefetching. On the other hand, NaRP optimally skips the remote memory writing, thus saving 50% of cycles spent on writing.

### 6.3 Aggregation

This part of the section evaluates the aggregation algorithms on the partitioned data, generated by the NaRP algorithm on the data with 256K group-by cardinality with $b_1 = 8$.

**Multi-core Aggregation:** We evaluate our aggregation algorithm (EAA) against three other algorithms, including random ordering algorithm (ROA), range based aggregation (RBA), and node independent aggregation (NIA). In ROA, each parallel instance aggregates the radix partitions in a random order with spin-lock on the granularity of the hash table bucket. The performance comparison is presented in Fig. 7 and the measured QPI traffic and cache miss rate during the running of aggregation algorithms are shown in Table 5. RBA performs poorly due to the heavy QPI traffic, although the lock cost and the cache coherence miss are avoided. The QPI traffic is costly in RBA due to two major reasons: (1) the remote memory access introduces delay in QPI; and (2) any parallel instance may read the partitions stored on any NUMA node at any time. Hence, the memory controller becomes the bottleneck when the read demands are not evenly distributed on the NUMA nodes. In contrast, parallel instance in ROA only reads local partition and hence the QPI traffic is effectively reduced. However, ROA suffers from the cache coherence miss penalty caused by range crash and the locking cost. If the input data follows a skewed distribution, ROA is much more likely to incur range crashes. This results in poor performance of ROA on skewed distributions. In NIA, the cache coherence miss is reduced by aggregating the partitions using the private hash table. Flushing these private hash tables brings QPI traffic, which is a huge overhead to NIA. EAA overcomes all the difficulties of the other three algorithms, presenting much superior performance, since remote memory access is rare and the cache coherence miss and the locking cost are avoided with almost no side effect.

We also report the size of local buffer used by EAA, to verify our theoretical studies in Sec.5. It is tested by varying the number of partitions per node. The results summarized in Fig. 8 show that the local buffer adoption is much less frequent than our theoretical bound. Such adoption is further reduced by increasing the number of partitions, which is consistent with our conclusion in the theoretical analysis. Since the cost of local buffer is proportional to the number of partitions aggregated into the local buffer, we hereby claim that our algorithm is essentially different from independent aggregation algorithm.

<table>
<thead>
<tr>
<th>Load balancing</th>
<th>Normal</th>
<th>Intra-node</th>
<th>Inter-node</th>
</tr>
</thead>
<tbody>
<tr>
<td>w.o. PAS</td>
<td>31.6</td>
<td>5.6</td>
<td>5.6</td>
</tr>
<tr>
<td>w. PAS</td>
<td>31.6</td>
<td>31.6</td>
<td>31.6</td>
</tr>
</tbody>
</table>

**Table 4**

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Phases</th>
<th>Normal</th>
<th>Intra-node</th>
<th>Inter-node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load balancing</td>
<td>w.o. PAS</td>
<td>31.6</td>
<td>5.6</td>
<td>5.4</td>
</tr>
<tr>
<td>Load balancing</td>
<td>w. PAS</td>
<td>31.6</td>
<td>31.6</td>
<td>31.6</td>
</tr>
</tbody>
</table>

**Table 5**

<table>
<thead>
<tr>
<th>Methods</th>
<th>QPI Traffic (GB/s)</th>
<th>Cache miss ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBA</td>
<td>14.1</td>
<td>9</td>
</tr>
<tr>
<td>ROA</td>
<td>5.6</td>
<td>90</td>
</tr>
<tr>
<td>NIA</td>
<td>3.4</td>
<td>74</td>
</tr>
<tr>
<td>EAA</td>
<td>0.2</td>
<td>24</td>
</tr>
</tbody>
</table>

**Table 5**

<table>
<thead>
<tr>
<th>Methods</th>
<th>QPI Traffic (GB/s)</th>
<th>Cache miss ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBA</td>
<td>14.1</td>
<td>9</td>
</tr>
<tr>
<td>ROA</td>
<td>5.6</td>
<td>90</td>
</tr>
<tr>
<td>NIA</td>
<td>3.4</td>
<td>74</td>
</tr>
<tr>
<td>EAA</td>
<td>0.2</td>
<td>24</td>
</tr>
</tbody>
</table>
6.4 Overall Performance Comparison

In this part of the section, we compare our complete aggregation algorithm with both NaRP and EEA against other baseline approaches, including shared aggregation (SA), hybrid aggregation (HA), independent aggregation (IA), shared aggregation with parallel radix partition (SAPRP), and independent aggregation with independent radix partition (IAPRP). Parallel instances in SA directly aggregate the input data into the globally shared hash table. A small hash table (e.g., no larger than L1 cache) is allocated for each parallel instance in HA, and the overflow results of the private hash table are merged into the shared hash table. In IA, each parallel instance uses a private hash table, which is merged together finally. IAPRP extends IA such that each parallel independently partitions its assigned data before aggregation. In SAPRP, the input data is first partitioned based on histogram-based parallel partitioning algorithm, then shared aggregation is employed on the partitioned data. To avoid hash table contention, each partition is processed by only one parallel instance. Note that we include our load balancing strategies in the implementation of all these baseline approaches. The group-by cardinality ranges from 256 to 256M in the experiments. For the radix partitioning, the partition bits $K$ follows $B - K \leq 8$, where $B$ is the number of binary bits representing hash value, such that the hash buckets corresponding to one partition fit in L1 cache. Note that when the group-by cardinality is smaller than 4K, SAPRP and IAPRP are equivalent to SA and IA respectively, as the radix partitioning phase in such cases is omitted.

**Throughput Evaluation:** The throughput of the four aggregation algorithms with 80 parallel instances is shown in Fig. 9. Due to space limitation, we only show the results for uniform data and real data. To better understand underlying reasons for the performance of each algorithm, we show the time breakdown for the evaluations on the real dataset in Fig. 10. Among all the algorithms, SA is extremely inefficient for small group-by cardinality, because it severely suffers from locking cost and cache coherence miss caused by the parallel updates on the shared hash table. The overhead decreases with the increase of group-by cardinality, but the size of hash table grows accordingly and an increasing number of cache capacity misses are introduced when accessing the hash table. In HA, the small private hash tables effectively reduce the contention to the shared hash table. Consequently, HA outperforms SA for small cardinality. However, the cache hit rate of private hash tables drops with the increase of group-by cardinality, and thus private hash tables cannot release the contention of the shared hash table but consume additional CPU cycles for swap in and out. As a result, HA performs poorly when the group-by cardinality is larger than $2^{14}$. In contrast, IA always reduces the locking cost and avoids
the cache coherence miss through private hash tables. However, hash table size grows with the increase of group-by cardinality, so the cache swap becomes more frequent when accessing the hash table and the cost of private hash table merging grows linearly. Therefore, the performance is better than SA for cardinality smaller than 1 million but degrades greatly when the cardinality increases. In IAIRP, the cache capacity miss is effectively reduced by radix-partitioning the data. Consequently, IAIRP outperforms IA. However, the performance is still significantly limited by the overhead of private hash tables. Moreover, as each instance partitions the data independently and individually, the size of each radix partition is relatively small, and consequently the advantage of cache capacity miss reduction degrades. Benefiting from the data locality enhancement by the radix partitioning and the avoidance of cache coherence miss and locking cost, SAPRP is much more efficient than SA in most cases. However, when the cardinality is very small, the exclusive processing of each partition reduces the concurrency and hence harms the performance significantly. The performance reduction is more noticeable on the real dataset, because the skewed data distribution introduces load balance problem, which further decreases the concurrency. Also, SAPRP suffers from remote memory access latency during both the radix partitioning phase and aggregation phase. In contrast, our algorithm (NaRP+EAA) shows much better performance than others. There are two major reasons, including: (1) the radix partitioning in NaRP not only improves the data locality but also considers the characteristics of NUMA architectures; and (2) EAA avoids using private hash tables, minimizes the remote memory access and effectively reduces the overhead of cache coherence miss and locking by the elaborated scheduling.

**Numa-awareness:** To gain better insight into the contributions of NUMA-awareness, e.g., avoiding remote memory access and across node synchronization, to the aggregation performance, we run IAIRP and NaRP+EAA under three settings: 1-node, 4-node, and 8-node settings. In 1-node setting, the algorithms are running on 1 NUMA node and hence memory access and synchronization are conducted locally. In 4-node setting, the algorithms are running on 4 fully interconnected nodes and all the across node communications are through 1-hop QPI. In contrast, in 8-node setting, two fully interconnected nodes are integrated by an external QPI and the communications between some nodes cause 2-hop QPI traffic. In each settings, the number of parallel instances in the algorithms is as many as the number of physical cores in the involved node(s). Table 6 shows the aggregation throughput per parallel instance in the two algorithms under the three settings, respectively. Comparing with the 1-node setting, a single parallel instance in SAPRP is inefficient in 4-node setting and the decrease is further in 8-node setting. That is because SAPRP fails to consider the characteristic of NUMA architectures. As a result, the more expensive the remote memory access is, the more inefficient SAPRP will be. In contrast, benefit from NUMA-awareness, the efficiency of NaRP+EAA does not show a noticeable performance degradation with the increase of remote memory access penalty.

**Scalability:** We compare the scalability of the five aggregation algorithms on the real data set with 1 million group-by cardinality, and show the results in Fig. 11. Among the five algorithms, SA scales the worst, as the locking cost and the cache coherence miss increases with the number of parallel instances. The scalability of IA is better than SA because the locking and cache coherence miss is avoided, but it is limited by the overhead of private hash table, which increases linearly with the number of parallel instances. In contrast, SAPRP and IAIRP has better performance because of the data locality enhancement by the radix partitioning. However, the cost of remote memory access and cache coherence in SAPRP and the overhead of private hash table in IAIRP increases with the number of instances. Consequently, SAPRP and IAIRP do not scale well, while our proposal NaRP+EAA is almost linearly scalable.

### 7 CONCLUSIONS

In this paper, we discuss the in-memory aggregation computation of big data with large group-by cardinality on modern servers under NUMA architectures. We point out the limitations of the state-of-the-art solutions on poor performances and scalability, due to the expensive costs of cache coherence miss and locking in NUMA architectures. We thus contribute a new NUMA-aware radix partitioning algorithm to improve the data partitioning efficiency, by optimizing the operations based on the characteristics of NUMA and including new load balancing strategies. We also design a novel efficient parallel aggregation algorithm, featuring in avoiding the cache coherence miss and minimizing the locking cost. We provide theoretical analysis on the performance of these proposals. The experiment results confirm that our new aggregation algorithm is at least 2 times faster than other state-of-the-art algorithms by testing on real and synthetic datasets on an IBM X5 server.

---

**Table 6**

<table>
<thead>
<tr>
<th></th>
<th>1-node</th>
<th>4-node</th>
<th>8-node</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAPRP</td>
<td>19.32</td>
<td>15.71</td>
<td>12.93</td>
</tr>
<tr>
<td>NaRP+EAA</td>
<td>20.14</td>
<td>22.02</td>
<td>19.80</td>
</tr>
</tbody>
</table>

**Fig. 11.** Scalability comparison on the real dataset.
We believe that the key ideas of avoiding cache coherence miss and synchronization in this paper can be extended to processing other database operations as well as many real-time analysis applications in NUMA architectures. Online aggregation of fast data stream, for example, is one of the possible extensions. By applying our implementation strategies appropriately, the efficiency of the existing methods could be dramatically improved.

Acknowledgment: This work is partially supported by National Science Foundation of China under grant No.61332006 and No.61232002.

References


