A Six-dimensional Analysis of In-memory Aggregation

Puya Memarzia [1], Suprio Ray [2], and Virendra C. Bhavsar [3]

Faculty of Computer Science, University of New Brunswick, Fredericton, Canada {[1] pmemarzi, [2] sray, [3] bhavsar}@unb.ca

ABSTRACT

Aggregation is widely used to extract useful information from large volumes of data. In-memory databases are rising in popularity due to the demands of big data analytics applications. Many different algorithms and data structures can be used for in-memory aggregation, but their relative performance characteristics are inadequately studied. Prior studies in aggregation primarily focused on small selections of query workloads and data structures, or I/O performance. We present a comprehensive analysis of in-memory aggregation that covers baseline and state-of-the-art algorithms and data structures. We describe 6 analysis dimensions which represent the independent variables in our experiments: (1) algorithm and data structure, (2) query and aggregate function, (3) key distribution and skew, (4) group by cardinality, (5) dataset size and memory usage, and (6) concurrency and multithread scaling. We conduct extensive evaluations with the goal of identifying the trade-offs of each algorithm and offering insights to practitioners. We also provide a glimpse on how the CPU cache and TLB are affected by these dimensions. We show that some persisting notions about aggregation, such as the relative performance of hashing and sorting, do not necessarily apply to modern platforms and state-of-the-art implementations. Our results show that the ideal approach in a given situation depends on the input and the workload. For instance, sorting algorithms are faster in holistic aggregate queries, whereas hash tables perform better in distributive queries.

1 INTRODUCTION

Despite recent advances in processing power, storage capacity, and transfer speeds, our need for greater query efficiency continues to grow at a rapid pace. Aggregation is an essential and ubiquitous data operation used in many database and query processing systems. It is considered to be the most expensive operation after joins, and is an essential component in analytical queries. All 21 queries in the popular TPC-H benchmark include aggregate functions [12]. A common aggregation workload involves grouping the dataset tuples by their key and then applying an aggregate function to each group.

Many prior studies on in-memory aggregation limited the scope of their research to a narrow set of algorithms, datasets, and queries. For example, many studies do not evaluate holistic aggregate functions [11, 32, 49]. The datasets used in most studies are based on a methodology proposed by Grey et al. [18]. These datasets do not evaluate the impact of data shuffling, or enforce deterministic group-by cardinality where possible. Some studies only evaluate a proposed algorithm against a naive implementation, rather than comparing it with other state-of-the-art implementations [20, 45]. Other studies have focused on secondary aspects, such as optimizing query planning for aggregations [48], distributed and parallel algorithms [11, 20, 40, 49, 50],

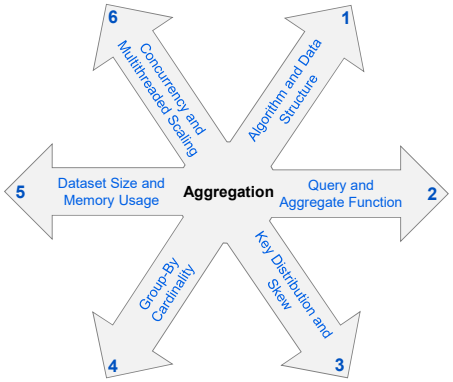


Figure 1: An overview of the analysis dimensions

and iceberg queries [45]. Additionally, some data structures have been proposed for in-memory query processing or as drop-in replacements for other popular data structures, but have not been extensively studied in the context of aggregation workloads [4, 26, 29]. Real-world applications cover a much more diverse set of scenarios, and understanding them requires a broader and more fundamental view. Due to these limitations, it is difficult to gauge the usefulness of these studies in other scenarios.

Different combinations of methodologies and evaluation parameters can produce very different conclusions. Applying the results from an isolated study to a general case may result in poor performance. For example, methods and optimizations for distributive aggregation are not necessarily ideal for holistic workloads. Our goal is to conduct a comprehensive study on the fundamental algorithms and data structures used for aggregation. This paper examines six fundamental dimensions that affect main memory aggregation. These dimensions represent well-known parameters which can be used as independent variables in our experiments. Figure 1 depicts an overview of the analysis dimensions. Figure 12 depicts a decision flow chart that summarizes our observations.

Dimension 1: Algorithm and Data Structure. In recent years, there have been many studies on main-memory data structures, such as tree-based indexes and hash tables. Many of these data structures can be used for in-memory aggregation. Aggregation algorithms can be categorized by the data structure used to store the data. Based on this we divide the algorithms into three main categories: sort-based, hash-based, and tree-based algorithms. We propose a framework that aims to cover many of the scenarios that could be encountered in real workloads. Over the course of this paper, we evaluate and discuss implementations from all three categories.

Dimension 2: Query and Aggregate Function. An aggregation query is primarily defined by its aggregate function. These functions are typically organized into three categories: distributive, algebraic, and holistic [17]. Distributive aggregate functions, such as *Count*, can be independently computed in a distributed manner. Algebraic aggregates are constructed by combining several distributive aggregates. For example, the *Average* function is a combination of *Sum* and *Count*. Holistic aggregate functions, such as *Median*, cannot be distributed in the same way as the two

previous categories because they are sensitive to the sort order of the data. Aggregation queries are also categorized based on whether their output is a single value (scalar), or a series of rows (vector). We evaluate a set of queries that cover both distributive and holistic, and vector and scalar categories.

Dimension 3: Key Distribution and Skew. The skew and distribution of the data can have a major impact on algorithm performance. Popular relational database benchmarks, such as TPC-H [12], generally focus on querying data that is non-skewed and uniformly distributed. However, it has been shown that these cases are not necessarily representative of real-world applications [13, 22]. Recently, researchers have proposed a skewed variant of the TPC-H benchmark [10]. The sizes of cities and the length and frequency of words can be modeled with Zipfian distributions, and measurement errors often follow Gaussian distributions [18]. Furthermore, skewed keys can be encountered as a result of joins [6] and composite queries. Our datasets are based on the specifications defined by [18] with a few additions. We cover the impact of both skew and ordering.

Dimension 4: Group-by Cardinality. Group-by cardinality is related to skew in the sense that both dimensions affect the number of duplicate keys. However, the group-by cardinality of a dataset directly determines the size (number of groups) of the aggregation result set. Prior studies have indicated that group-by cardinality has a major impact on the relative performance of different aggregation methods [2, 20, 24, 32]. These studies claim that hashing performs faster than sorting when the group-by cardinality is low relative to the dataset size, and that this performance advantage is reversed when the cardinality is high. We find that the accuracy of this claim depends on the implementation. We evaluate the performance impact of group-by cardinality, as well as its relationship with CPU cache and TLB misses.

Dimension 5: Dataset Size and Memory Usage. Recent advances in computer hardware have encouraged the use of main-memory database systems. These systems often focus on analytical queries, where aggregation is a key operation. Although memory has become cheaper and denser, this is offset by the increasing demands of the industry. Our goal is to shed some light on the trade-off between memory efficiency and performance.

Dimension 6: Concurrency and Multithreaded Scaling. Nowadays, query processing systems are expected to support intra-query parallelism in addition to interquery parallelism. Concurrency imposes additional challenges, such as reducing synchronization overhead, eliminating race conditions, and multithreaded scaling. We explore the viability and scalability of several multi-threaded implementations.

The key contributions of this paper are:

- Evaluation of aggregation queries using sort-based, hash-based, and tree-based implementations
- Methodology to generate synthetic datasets that expands on prior work
- Extensive experiments that include comparison of distributive and holistic aggregate functions, vector and scalar aggregates, range searches, evaluation of memory efficiency and TLB and cache misses, and multithreaded scaling
- Insights on performance trends and suggestions for practitioners

The remainder of this paper is organized as follows: We describe the queries in Section 2. We elaborate on the algorithms and data structures in Section 3. In Section 4 we specify the characteristics of our synthetic datasets. We present and discuss our experimental setup and evaluation results in Section 5, and summarize our findings in Section 6. In Section 7 we categorize and explore the related work. Finally, we conclude the paper in Section 8.

2 QUERIES

In this section, we describe the queries used for our experiments. In Table 1 we describe each query along with a simple example. Our goal is to evaluate and compare different aggregation variants. There are three main categories of aggregate functions: distributive, algebraic, and holistic. Distributive functions, such as *Count* and *Sum*, can be processed in a distributed manner. This means that the input can be split and processed in multiple partitions, and then the intermediate results can be combined to produce the final result. Algebraic functions consist of two or more Distributive functions. For instance, *Average* can be broken down into two distributive functions: *Count* and *Sum*. Holistic aggregate functions cannot be decomposed into multiple distributive functions, and require all of the input data to be processed together. For example, if an input is split into two partitions and the *Mode* is calculated for each partition, it is impossible to accurately determine the Mode for the total dataset. Other examples of Holistic functions include *Rank*, *Median*, and *Quantile*.

The output of an aggregation can be either in *Vector* or *Scalar* format. In Vector aggregates, a row is returned in the output for each unique key in the designated column(s). These columns are commonly specified using the *group-by* or *having* keywords. The output value is returned as a new column next to the group-by column. Scalar aggregates process all the input rows and produce a single scalar value as the result.

Sometimes it is desirable to filter the aggregation output based on user defined thresholds or ranges. We study an example of a range search combined with a vector aggregate function in *Q7*. In a real-world environment, it may be possible to push the range conditions to an earlier point in the query plan, but if several different range scans are desired, early filtering may not be possible. The main purpose of this query is to evaluate each data structure's efficiency at performing a range search in addition to the aggregation.

3 DATA STRUCTURES AND ALGORITHMS

In this section, we introduce the data structures and algorithms that we use to implement aggregate queries. We divide these algorithms into three categories: sort-based, hash-based, and tree-based. In order to facilitate reproducibility, we have selected open-source data structures and sort algorithms where possible. We also consider several state-of-the-art data structures, such as ART[26], HOT[8], and Libcuckoo[29]. Since the performance of algorithms can shift with hardware architectures, we also consider some of the more fundamental algorithms and data structures, such as a B+Tree [7]. Throughout this section we will state theoretical time complexities using n as the number of elements and k as the number of bits per key.

The implementation of an aggregate operator can be broken down into two main phases: the *build* phase and the *iterate* phase. Consider this example using a hash table and a vector aggregate function (refer to Q1 in Table 1). During the build phase, each key (created from the group-by attribute or attributes) is looked up in the hash table. If it does not exist, it is inserted with a starting value of one. Otherwise, the value for the existing key is incremented. Once the build phase is complete the iterate phase reads the key-value pairs from the hash table and writes

Table 1: Aggregation Queries

Query	SQL Representation (example)	Aggregate Function	Category	Output Format
Q1	`SELECT product_id, COUNT(*) FROM sales GROUP BY product_id`	Count	Distributive	Vector
Q2	`SELECT student_id, AVG(grade) FROM grades GROUP BY student_id`	Average	Algebraic	Vector
Q3	`SELECT product_id, MEDIAN(amount) FROM products GROUP BY product_id`	Median	Holistic	Vector
Q4	`SELECT COUNT(sale_id) FROM sales`	Count	Distributive	Scalar
Q5	`SELECT AVG(grade) FROM grades`	Average	Algebraic	Scalar
Q6	`SELECT MEDIAN(part_id) FROM parts`	Median	Holistic	Scalar
Q7	`SELECT product_id, COUNT(*) FROM sales WHERE product_id BETWEEN 500 AND 1000 GROUP BY product_id`	Count with Range Condition	Distributive	Vector

the resulting items to the output. A similar procedure is used for tree data structures. The calculation of the aggregate value during the build phase (early aggregation) is only possible when the aggregate function is distributive or algebraic. As a result, holistic aggregate values cannot be calculated until all records have been inserted. Sort-based approaches "build" a sorted array using the group-by attributes. As a result, all the values for each group are placed in consecutive locations. The aggregate values are calculated by iterating through the groups.

3.1 Sort-based Aggregation Algorithms

Sorting algorithms are a crucial building block in any query processing system. Many popular database systems, such as Microsoft SQL and Oracle, employ both sort-based and hash-based algorithms. We examine several algorithms designed for sorting arrays of fixed length integers, although some of the approaches could be adapted to variable length strings.

3.1.1 ***Quicksort.*** Quicksort is a sorting method based on the concept of divide and conquer that was invented by Tony Hoare [19] and remains very popular to this day. The average time complexity of Quicksort is $O(n \log(n))$. The worst case time complexity is considerably worse at $O(n^2)$, but this is rare, and is mitigated on modern implementations [21, 35] .

3.1.2 ***Introsort.*** Introspective sort (Introsort) is a hybrid sorting algorithm that was proposed by David Musser [33]. Introsort can be regarded as an algorithm that builds on Quicksort and improves its worst case performance. This sorting algorithm starts by sorting the dataset with Quicksort. When the recursion depth passes a certain threshold, the algorithm switches to Heapsort. This threshold is defined as the logarithm of the number of elements being sorted. This algorithm guarantees a worst case time complexity of $O(n \log(n))$.

The GCC variant of Introsort [21] differs from the original algorithm in two ways. Firstly, the recursion depth is set to $2 * \log(n)$. Secondly, the algorithm switches to Insertion sort, which is fast on small data chunk, but has a time complexity of $O(n^2)$.

3.1.3 ***Radix Sort (MSB and LSB).*** Radix sorting works by sorting the data one bit (binary digit) at a time. There are two variants of Radix Sort, based on the order in which the bits are processed: Most Significant Bit (MSB) Radix Sort, and Least Significant Bit (LSB) Radix Sort. As the names suggest, MSB sorts the data starting from the top (leftmost) bits, and works its way down. In comparison, LSB starts from the bottom bits. The time complexity of Radix sort is $O(k * n)$ where k is the key width (number of bits in the key), and n is the number of elements.

3.1.4 ***Spreadsort.*** Spreadsort is a hybrid sorting algorithm that combines the best traits of Radix and comparison-based sorting. This algorithm was invented by Steven J. Ross in 2002 [37]. Spreadsort uses MSB Radix partitioning until the size of the partitions reaches a predefined threshold, at which point it switches to comparison-based sorting using Introsort. Comparison-based sorting is more efficient on small sequences of data compared to Radix partitioning. The time complexity of the MSB Radix phase is $O(n \log(k/s + s))$ where k is the key width, and s is the maximum number of splits (default is 11 for 32 bit integers). As mentioned, time complexity of Introsort is $O(n \log(n))$.

3.1.5 ***Sorting Microbenchmarks.*** In order to obtain a basic understanding of the performance of these algorithms and how they compare, we evaluate five integer sorting algorithms on a variety of datasets. The tested algorithms are: Quicksort, Introsort, MSB Radix Sort, LSB Radix Sort, and Spreadsort. We test each algorithm on five data distributions: random integers between one and five, random integers between one and one million, random integers between one thousand and one million, presorted sequential integers, and reverse sorted sequential integers. We measure the time to sort ten million integers from each distribution. The results, depicted in Figure 2, show that Introsort and Spreadsort generally outperform the other sorting algorithms.

3.2 Hash-based Aggregation Algorithms

Hash tables are particularly efficient in workloads that require fast random lookups, which they perform in constant time. A hash function transforms a key into an address within the table. However, hash tables do not generally guarantee any ordering of the keys (lexicographical or chronological). It is possible to pre-sort the data and construct a hash function that guarantees

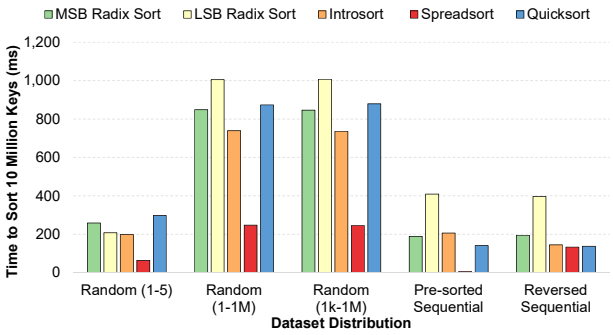


Figure 2: Sort Algorithm Microbenchmark

ordered keys (minimal perfect hashing [15, 30]). However, the impact on query execution time would be quite severe.

Hash tables are not well-suited to gradual dynamic growth, as growing the table may entail rehashing all existing elements as well. In principle, a hash table's size could be tuned to anticipate the dataset group-by cardinality. However, in practice it is difficult to estimate the cardinality, particularly when there are several group-by columns. Cardinality estimation errors result in excessive memory usage if too high, and costly rehash operations if too low. In our experiments we assume that only the size of the dataset is known, hence we set the initial size of the hash tables accordingly.

Hash tables can be categorized based on their collision resolution scheme. Collision resolution defines how a hash table resolves conflicts caused by multiple keys hashing to the same location. We now describe four collision resolution schemes and the implementations that use them: linear probing, quadratic probing, separate chaining, and cuckoo hashing.

3.2.1 ***Linear probing.*** Linear probing is part of the family of collision resolution techniques called open addressing. Open addressing hash tables typically store all the items in one contiguous array. They do not use pointers to link data items. Linear probing specifies the method used to search the hash table. An insertion begins from the hash index and probes forward in increments of one until the first empty bucket is found. Linear probing hash tables do not need to allocate extra memory to store new items as long as the table has empty slots. However, they may encounter an issue called primary clustering, where colliding records form long sequences of occupied slots. These sequences displace incoming keys, and they grow each time they do so, resulting in the high number of displacement of records.

We implement a custom linear probing hash table using several industry best practices, such as maintaining a power of two table size. If the desired size is not a power of two then the nearest greater power of two is chosen. This is a popular optimization that allows the table modulo operation to be replaced with a much faster bitwise AND. The downside to this policy is that is easier to overshoot the available memory. In order to resolve this, our implementation falls back to the modulo operation and the table size is set to the nearest prime number if possible, and the exact size parameter is used as the final fallback.

3.2.2 ***Quadratic probing.*** Quadratic probing is an open addressing scheme that is very similar to linear probing. Like linear probing it calculates a hash index and searches the table until a match is found. Rather than probing in increments of one, a quadratic function is used to determine each successive probe index. For example, with an arbitrary hash function $h(x)$ and quadratic function $f(x) = x^2$, the algorithm probes $h(x)$, $h(x)+1$, $h(x)+4$, $h(x)+9$ instead of a linear probe sequence of $h(x)$, $h(x)+1$, $h(x)+2$, $h(x)+3$. This approach greatly reduces the likelihood of clustering, but it does so at the cost of reducing data locality.

Google **Sparse Hash** and **Dense Hash** [41] are based on open addressing with quadratic probing. Sparse Hash favors memory efficiency over speed, whereas Dense Hash targets faster speed at the expense of higher memory usage.

3.2.3 ***Separate chaining.*** Separate chaining is a way of resolving collisions by chaining key-value pairs to each other with pointers. Buckets with colliding items resemble a single linked list. The main advantages of separate chaining include fast insert performance, and relatively versatile growth. The use of pointer-linked buckets reduces data locality, which is important for lookups and updates. However, unlike linear probing, separate chaining hash tables do not suffer from primary clustering.

Separate chaining hash tables remain popular in recent works [2, 3, 9, 39]. Templated separate chaining hash tables are included as part of the Boost and standard C++ libraries. Additionally, the Intel TBB library provides versatile hash tables that support concurrent insertion and iteration.

3.2.4 ***Cuckoo Hashing.*** Cuckoo hashing was originally proposed by Pagh et al. [34]. Its core concept is to store items in one of two tables, each with a corresponding hash function (this can be extended to additional tables). If a bucket is occupied by another item, the existing item is displaced and reinserted into the other table. This process continues until all items stabilize, or the number of displacements exceeds an arbitrary threshold. Cuckoo hashing provides a guarantee that reads take no more than two lookups. Its main drawback is relatively slower and less predictable insert operations, and the possibility of failed insertions.

In [29], researchers from Intel labs presented a concurrent cuckoo hashing technique **Libcuckoo**. Libcuckoo introduces improvements to the insertion algorithm by leveraging hardware transactional memory (HTM). This hardware feature allows concurrent modifications of shared data structures to be atomic. Their experimental results indicate that Libcuckoo outperforms MemC3 [14], and Intel TBB [35].

3.3 Tree-based Aggregation Algorithms

Hash-based and sort-based aggregation approaches are very popular, mainly due to a heavy focus of past studies on "write once read once" (WORO) aggregation workloads, as opposed to "write once read many" (WORM). We consider several tree data structures, and assess their viability for aggregation.

Trees are commonly used to evaluate range conditions. However, aggregation benchmarks, such as TPC-H, do not include range queries. Tree data structures are well suited to incremental dynamic growth. The trade-off is higher time complexities for both insert and lookup operations, compared to hash tables.

We divide the tree data structures into *comparison trees* and *radix trees*. Comparison trees have traditionally served as indexing structures, but Radix trees are being increasing adopted in recent main memory databases, such as HyPer [23] and Silo [44]. The *Btree* family and *Ttree* are comparison trees, and *ART* and *Judy* are Radix trees.

3.3.1 ***Btree.*** The *B-tree* is a popular tree data structure that was initially invented in 1971 by Bayer et al. [5], and forms the basis for many modern variants [7, 28]. A B-tree is a balanced *m-way* tree where m is the maximum number of children per

Table 2: Data Structure Time Complexity

Data Structure	Average Case Insert	Worst Case Insert	Average Case Search	Worst Case Search
ART	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Judy	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Btree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Ttree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Separate Chaining	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Linear Probing	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Quadratic Probing	$O(1)$	$O(\log(n))$	$O(1)$	$O(\log(n))$
Cuckoo Hashing	$O(1)$ (amortized)	$O(n)$ (rehash)	$O(1)$	$O(1)$

node. The B-tree is perhaps best recognized as a popular disk-based data structure used for database indexing, although they are also used for in-memory indexes. The defining characteristics of B-trees are that they are shallow and wide due to using a high fanout. This reduces the number of node lookups as each node contains multiple data items. B-trees may also include pointers between leaf nodes to facilitate more efficient range scans. The time complexity for inserting n items into a B-tree is $O(n \log(n))$. We use a cache-optimized implementation based on the *STX B+tree* [7] which we will henceforth refer to as *Btree*.

3.3.2 ***Ttree.*** *Ttree* (spelled "T-tree" in the literature) was originally proposed in 1986 by Lehman et al. [25]. Its intended purpose was to provide an index that could outperform and replace the disk-oriented B-tree for in-memory operations. Although the Ttree showed a lot of promise when it was first introduced, we show in section 3.4 that advancements in hardware design have rendered it obsolete on modern processors.

3.3.3 ***ART.*** ART (Adaptive Radix Tree) [26] is a Radix tree variant with a variable fan-out. Its inventors present it as a data structure that is as fast as a hash table, with the added bonus of sorted output, range queries, and prefix queries. ART uses SIMD instructions to concurrently compare multiple keys in parallel. ART saves on memory consumption by using dynamic node sizes and merging inner nodes when possible. Radix trees have several key advantages compared to comparison trees. The height of a radix tree depends on the length of the keys, rather than the number of keys. Additionally, in contrast with comparison trees, they do not need to perform re-balancing operations.

We also considered *HOT* [8], which builds on the same principles as ART. However, we found its performance with integer keys to be noticeably worse, as its main focus is string keys.

3.3.4 ***Judy Arrays.*** Judy Arrays (henceforth referred to as Judy) were invented by Doug Baskins [4], and defined as a type of sparse dynamic array designed for sorting, counting, and searching. They are intended to replace common data structures such as hash tables and trees. Judy is implemented as a 256-way Radix tree that uses variable fan out and a total of 20 compression techniques to reduce memory consumption and improve cache efficiency [1]. Judy is fine-tuned to minimize cache misses on 64 byte cache-lines. Like many other tree data structures, the size of a Judy array dynamically grows with the data and does not need to be pre-allocated.

3.4 Data Structure Microbenchmarks

We use a microbenchmark to evaluate each data structure's efficiency in a store and lookup workload. We separately measure the time it takes to build the data structure (build phase), and the time to read all the items in the data structure (iterate phase). All hash tables are sized to the number of elements. The results are depicted in Figure 3 using the abbreviations outlined in Table 3. With the exception of Hash_LC, the build phase is faster on all the hash tables due $O(1)$ insert complexity. Hash_LC performs poorly in the build phase because it is designed as a concurrent data structure. We evaluate its concurrent scalability in Section 5.8. Hash_LP and Hash_Dense provide the fastest overall times. Btree is noticeably faster in the iterate phase, but it takes a relatively long time to build due to the cost of balancing the tree. Due to the relatively poor performance exhibited by Ttree in both phases, we opt to omit it from subsequent experiments.

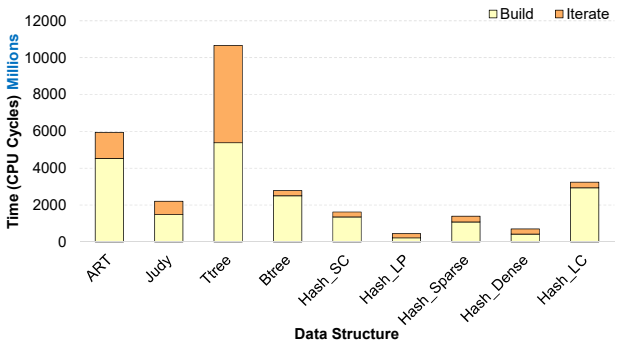


Figure 3: Data Structure Microbenchmark

3.5 Time Complexity

It is a well known fact that time complexities for algorithms are not always the best predictors of real-world performance. This is due to a number of factors, including hidden constants and overheads arising from the implementation, hardware characteristics such as CPU architecture cache and TLB, compiler optimizations, and operating systems. On modern systems, cache misses are particularly expensive. Nevertheless, time complexity is widely used as a mean to understand and compare the relative performance of different algorithms.

Table 2 provides an overview of the known time complexities for each of the data structures that we evaluate. Here n denotes the number of elements, and k the number of bits in the key.

4 DATASETS

In order to effectively evaluate the algorithms, we generate a set of synthetic datasets that vary in terms of input size, group-by cardinality, key distribution, and key range. Our datasets are based on the highly popular input distributions described in prior works [11, 18, 20]. We employ several modifications to these datasets, with the goal of expanding the data characteristics

Table 3: Algorithms and Data Structures

Label	Type	Description
ART	Tree	Adaptive Radix Tree [26]
Judy	Tree	Judy Array [4]
Btree	Tree	STX B+Tree [7]
Hash_SC	Hash	std::unordered_map [21] (Separate Chaining)
Hash_LP	Hash	Linear Probing (Custom)
Hash_Sparse	Hash	Google Sparse Hash [41]
Hash_Dense	Hash	Google Dense Hash [41]
Hash_LC	Hash	Intel libcuckoo [29]
Introsort	Sort	std::sort (Introsort) [21]
Spreadsort	Sort	Boost Spreadsort [42]

Table 4: Dataset Distributions

Abbreviation	Description	Cardinality
Rseq	Repeating Sequential	Deterministic
Rseq-Shf	Rseq Uniform Shuffled	Deterministic
Hhit	Heavy Hitter	Deterministic
Hhit-Shf	Hhit Uniformly Shuffled	Deterministic
Zipf	Zipfian	Probabilistic
MovC	Moving Cluster	Probabilistic

that we evaluate. Some datasets, such as the sequential dataset, produce very predictable patterns. For such datasets, we generate an additional variant with uniform random shuffling. In [11] it is mentioned that the group-by cardinality is often probabilistic. We enforce deterministic group-by cardinality in cases where the target distribution of the dataset would not be affected.

Throughout this paper we use *random* to refer to a uniform random function with a fixed seed, and *shuffling* refers to the use of the aforementioned function to shuffle all the records in a dataset. The number of records in the dataset is denoted as n records and the group-by cardinality is c.

In the repeating sequential dataset (*RSeq*), we generate a series of segments that contain multiple number sequences. The number of segments is equal to the group-by cardinality, and the number of records in each segment is equal to the dataset size divided by the cardinality. A shuffled variant of the repeating sequential dataset (*RSeq-Shf*) is also generated. This dataset mimics transactional data where the key incrementally increases.

In the heavy hitter dataset (*HHit*), a random key from the key range accounts for 50% of the total keys. The remaining keys are produced at least once to satisfy the group-by cardinality, and then chosen on a random basis. In a variant of this dataset, the resulting records are shuffled so that the heavy hitters are not concentrated in the first half of the dataset. Real-world examples of heavy hitters include top selling products, and network nodes with the highest traffic.

In the Zipfian dataset (*Zipf*), the distribution of the keys is skewed using Zipf's law [36]. According to Zipf's law, the frequency of each key is inversely proportional to its rank. We first generate a Zipfian sequence with the desired cardinality c and Zipf exponent $e = 0.5$. Then we take n random samples from this sequence to build n records. The final group-by cardinality is non-deterministic and may drift away from the target cardinality as c approaches n. The Zipf distribution is used to model many big data phenomena, such as word frequency, website traffic, and city population.

Table 5: Experiment Parameters

Dataset	Repeating Sequential, Heavy Hitter, Moving Cluster, Zipfian
Dataset Size	100M, 10M, 1M, 100k
Group-by Cardinality	100, 1000, 10000, 100000, 1000000, 10000000
Algorithm	Hash_LP, Hash_SC, Hash_LC, Hash_Sparse, Hash_Dense, ART, Judy, Btree, Introsort, Spreadsort, Hash_TBBSC, Sort_BI, Sort_QSLB
Thread Count	1, 2, 3, 4, 5, 6, 7, 8 (logical core count)
Query	Q1 (Vector Distributive), Q3 (Vector Holistic), Q6 (Scalar Distributive), Q7 (Vector Distributive with Range)

In the moving cluster dataset (MovC), the keys are chosen from a window that gradually slides. The i^{th} key is randomly selected from the range $\lfloor (c - W)i/n \rfloor$ to $\lfloor (c - W)i/n + W \rfloor$, where the window size $W = 64$ and the cardinality c is greater than the window size ($c >= W$). The moving cluster dataset provides a gradual shift in data locality and is similar to workloads encountered in streaming or spatial applications.

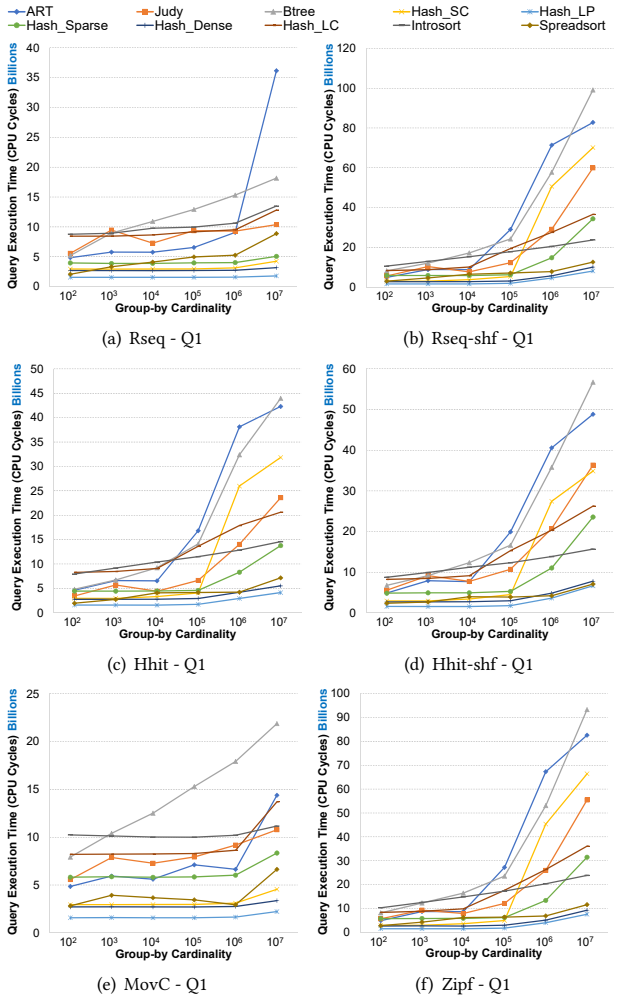
5 RESULTS AND ANALYSIS

In this section we evaluate the efficiency of the aggregation algorithms. We examine and compare the performance impact of dataset size, group-by cardinality, key skew and distribution, data structure algorithm, and the query and aggregation functions. We also evaluate peak memory usage as a measure of each algorithm's memory efficiency. The experimental parameters are outlined in Table 5.

For each experiment the input dataset is preloaded into main memory. We do not measure the time to read the input from disk. Throughout this paper we aim to understand how each of these dimensions can affect main memory aggregation workloads. Due to space constraints we only show the results for Q1, Q3, Q6 and Q7. We start the experiments with two common vector aggregation queries (Q1 and Q3) in Section 5.2. Due to the popularity of these queries, we further analyze these queries by evaluating cache and TLB misses in Section 5.3, memory usage for different dataset sizes in Section 5.4, and data distributions in Section 5.5. Additionally, we evaluate range searches (Q7) in Section 5.6 and scalar aggregation queries (Q6) in Section 5.7. We examine multithreaded scaling in Section 5.8. Finally, we summarize our findings in Section 6. We now outline our experimental setup.

5.1 Platform Specifications

The experiments are evaluated on a machine with an Intel Core i7 6700HQ processor at 3.5GHz, 16GB of DDR4 RAM at 2133MHz, and a 512GB SSD. The CPU is a quad core based on the Skylake microarchitecture, with hyper-threading (8 logical cores), 256KB of L1 cache, 1MB of L2 cache, and 6MB of L3 cache. The TLB can hold 64 entries in the L1 Data TLB, and 1536 entries in the L2 Shared TLB (4KB pages). The code is compiled and run on Ubuntu Linux 16.04 LTS, using the GCC 7.2.0 compiler, and the -O3 and -march=native optimization flags. We now present and discuss the experimental results.

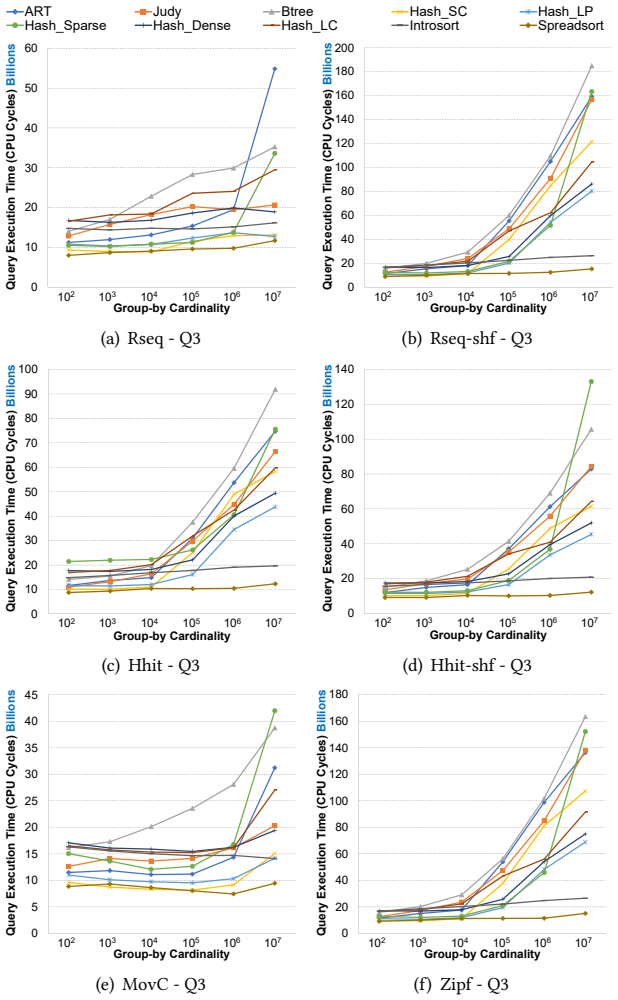


(a) Rseq - Q1 (b) Rseq-shf - Q1

(c) Hhit - Q1 (d) Hhit-shf - Q1

(e) MovC - Q1 (f) Zipf - Q1

Figure 4: Vector Aggregation Q1 - 100M Records



(a) Rseq - Q3 (b) Rseq-shf - Q3

(c) Hhit - Q3 (d) Hhit-shf - Q3

(e) MovC - Q3 (f) Zipf - Q3

Figure 5: Vector Aggregation Q3 - 100M records

5.2 Results - Vector Aggregation

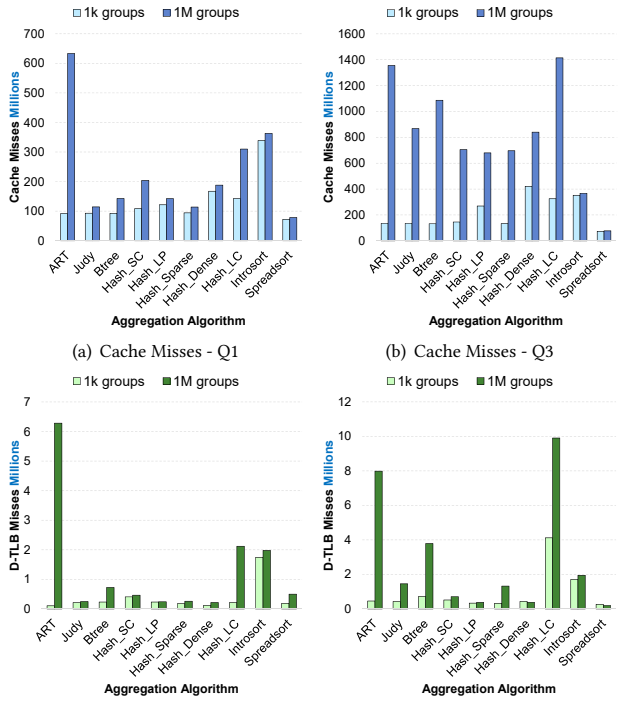
We begin our experiments by evaluating Q1 and Q3 (see Table 1), which are based on commonly used aggregate functions. Due to space constraints and the similarity between Algebraic and Distributive functions, we do not show results for Q2. In these experiments, we keep the dataset size at a constant 100M and vary the group-by cardinality from 10^2 to 10^7. In each chart we measure the query execution time for a given query and dataset distribution, and the group-by cardinality increases from left to right. The results for Q1 (Vector COUNT) and Q3 (Vector MEDIAN) are shown in Figures 4 and 5 respectively. A larger group-by cardinality means more unique keys, and fewer duplicates. In tree-based algorithms the data structure dynamically grows to accommodate the group-by cardinality. This is reflected in the gradual increase in query execution time. The insert performance of ART and Judy depends on the length of the keys, which increases with cardinality. Additionally, the compression employed by ART and Judy are more heavily used at high cardinality.

The results for Q3 show that Spreadsort is the fastest algorithm across the board. The overall trend shows that hash-based algorithms, such as Hash_SC and Hash_LP, are competitive with Spreadsort until the group-by cardinality exceeds 10^4. The execution times for both Spreadsort and Introsort show considerably less variance, whereas the worsening of data locality results in sharp declines in performance for the hash-based and tree-based implementations. The performance of Hash_Sparse dramatically worsens at 10^7 groups, suggesting that the combination of Hash_Sparse's gradual growth policy and the extra space needed by this query result in a much steeper decline in performance compared to Q1.

In order to understand why Hash_LP outperforms all the other algorithms in Q1, we need to consider several factors. Firstly, the average insert time complexity (as shown in Table 2) is unaffected by group-by cardinality. Secondly, the cache-friendly layout of Hash_LP takes great advantage of data locality compared to the other hash tables. Lastly, compared to Q3, Q1 does not require additional memory to store the values associated with each key. This reduces the pressure on the cache and TLB, and allows Hash_LP to compete with memory efficient approaches such as Spreadsort. We further explore cache and TLB behavior in Section 5.3 and memory consumption in Section 5.4.

5.3 Results - Cache and TLB misses

Cache and TLB behavior are metrics of algorithm efficiency. Together with runtime and memory efficiency, they paint a picture of how different algorithms compare with each other. Processing large volumes of data in main memory often leads to many cache and TLB misses, which can hinder performance. A cache miss can sometimes be satisfied by a TLB hit, but a TLB miss incurs a page table lookup, which is considerably more expensive. Using



(a) Cache Misses - Q1 (b) Cache Misses - Q3

(c) TLB misses - Q1 (d) TLB misses - Q3

Figure 6: Cache and TLB misses - Rseq 100M Dataset

the *perf* tool, we measure the CPU cache misses and data-TLB (D-TLB) misses of Q1 and Q3 with low cardinality (10^3 groups) and high cardinality (10^6 groups) datasets. The results are depicted in Figure 6.

It is interesting to compare the results in Figure 6(c) with the performance discrepancy between Hash_LP and Spreadsort in Q1. At low cardinality, the number of TLB misses is relatively close between the two algorithms. However, at high cardinality, Spreadsort exhibits considerably higher TLB misses. Similarly, in Figure 4(a) we see the runtime gap between the two algorithms widen in Hash_LP's favor as the cardinality increases to 10^7.

Although this metric is not a guaranteed way to predict the relative performance of the algorithms, it is a fairly reliable measure of scalability and overall efficiency. The cache behavior of Spreadsort is consistently good. Other algorithms, such as ART, exhibit large jumps in both cache and TLB misses. This correlates with similar gaps in the query runtimes, and it is noted in [8] that ART's memory efficiency and performance may degrade if it creates many small nodes due to the dataset distribution.

5.4 Results - Memory Efficiency

Memory efficiency is a performance metric that is arguably as important as runtime speed. We now measure the peak memory consumption at various dataset sizes.

To do so we lock the group-by cardinality at 10^3 and vary the dataset size from 10^5 up to 10^8. These measurements are taken by using the Linux */usr/bin/time -v* tool to acquire the *maximum resident set size* for each configuration. The results for Q1 are depicted in Table 6 and the memory usage of Q3 is shown in Table 7. The results show that the hash tables consume the most memory, followed by the tree data structures. The sort algorithms are the most memory efficient because they sort the data in-place. In order to maintain good insert performance, most hash tables consume more memory than they need to store the items, and some will only resize to powers of two. Hash_Dense's memory usage is particularly high because it uses 6× the size of the entries in the hash table when performing a resize. After the resize is completed, the memory usage shrinks down to 4× the previous size. Comparing Tables 6 and 7, we see a jump in memory usage from Q1 to Q3. This is due to the fact that Q3 requires the data structures to store the keys and all associated values in main memory, whereas Q1 only requires the keys and a count value. Consequently, holistic queries like Q3 will generally consume more memory.

Table 6: Peak Memory Usage (MB) - Q1 on Rseq 10^3 Groups

Dataset Size / Algorithm	10^5	10^6	10^7	10^8
ART	4.45	11.61	131.61	1027.44
Judy	4.31	11.53	131.49	1027.45
Btree	4.54	11.79	131.66	1027.60
Hash_SC	5.45	19.41	159.07	1540.95
Hash_LP	5.23	18.67	156.29	1529.44
Hash_Sparse	4.61	11.94	131.68	1027.58
Hash_Dense	6.42	27.33	336.02	2814.70
Hash_LC	26.95	44.44	263.14	2069.90
Introsort	4.50	11.74	131.66	1027.59
Spreadsort	4.53	11.55	131.65	1027.44

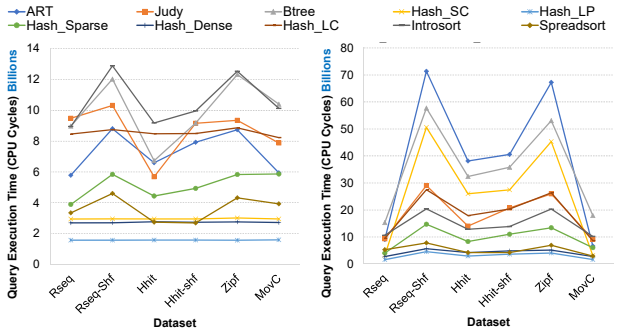
Table 7: Peak Memory Usage (MB) - Q3 on Rseq 10^3 Groups

Dataset Size / Algorithm	10^5	10^6	10^7	10^8
ART	5.07	15.26	132.57	1212.46
Judy	4.87	15.37	132.68	1212.82
Btree	5.14	15.33	132.78	1212.64
Hash_SC	5.88	23.28	211.39	1986.78
Hash_LP	7.80	45.55	437.76	4264.07
Hash_Sparse	5.11	15.92	137.78	1255.51
Hash_Dense	12.74	79.16	1156.55	9404.48
Hash_LC	30.01	68.44	686.95	5575.09
Introsort	4.45	11.61	131.57	1027.50
Spreadsort	4.31	11.66	131.59	1027.48

5.5 Results - Dataset Distribution

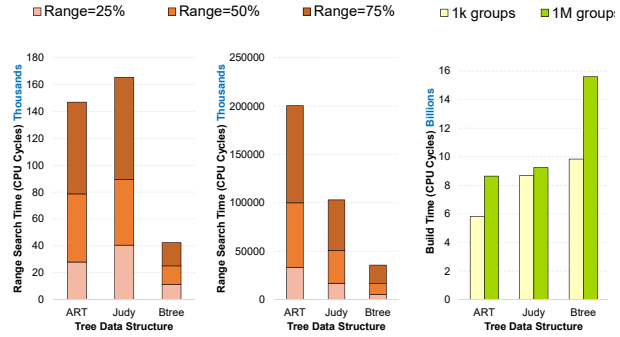
These experiments show the performance impact of the data key distribution. The results are presented in Figures 7(a) and 7(b). In each figure, we vary the key distribution while keeping the dataset size at a constant 100 million records. To get a better understanding of how this factor ties in with cardinality, we show results for both low and high cardinality (10^3 and 10^6 groups).

The results point out that Zipf and Rseq-Shf are generally the most performance-sensitive datasets. The shuffled variants of Rseq and HHit take longer to run due to a loss of memory locality. By comparing the two figures we can see that this effect is amplified by group-by cardinality, as it increases the range of keys that must be looked up in the cache. In the low cardinality dataset, the number of unique keys is small compared to the dataset size. Introsort is the overall slowest algorithm at low cardinality and its performance is around the middle of the pack at high cardinality. This is in line with prior works suggesting that sort-based aggregation is faster when the group-by cardinality is high [32]. However, as we can see in the results produced by Spreadsort, the algorithm also performs well at low cardinality which contradicts earlier claims. Due to this, it may be worth revisiting hybrid sort-hash aggregation algorithms in the future.



(a) 10^3 groups (low cardinality) (b) 10^6 groups (high cardinality)

Figure 7: Vector Q1 - Variable Key Distributions - 100M records



(a) Range Search Time - 10^3 groups (low cardinality) (b) Range Search Time - 10^6 groups (high cardinality) (c) Build time by dataset cardinality

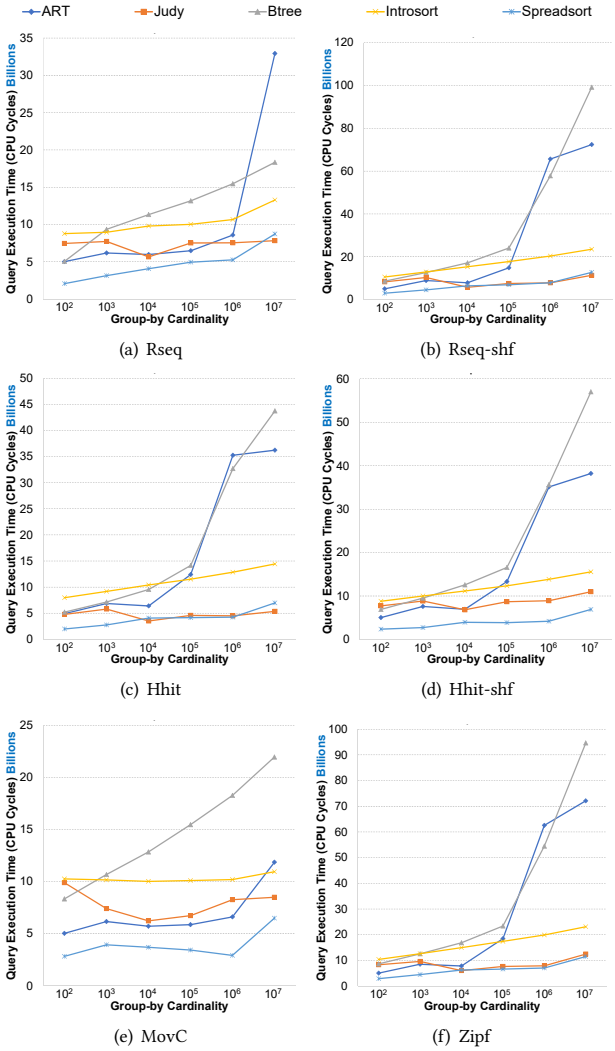
Figure 8: Range Search Aggregation Q7 - 100M records

The results also highlight an interesting trend when it comes to shuffled/unordered data. Observe that ART's performance in Figure 7(b) significantly worsens when going from Rseq to Rseq-Shf or indeed any unordered distribution. The combination of high cardinality and unordered data increase pressure on the cache and TLB. If we consider how well Spreadsort performs in these situations, then the results indicate that presorting the data before invoking the ART-based aggregate operator could significantly improve performance. However, careful consideration of the algorithm and dataset is required to avoid increasing the runtime.

5.6 Results - Range Search

The goal of this experiment is to evaluate algorithms that provide a native range search feature, and combine this with a typical aggregation query. Although it is possible to implement an integer range search on a hash table, this would not work for strings and other keys with non-discrete domains. Consequently, we focus on the tree-based aggregation algorithms. Q7 calculates the vector count aggregates for a range of keys. The tuples that do not satisfy the range condition could be filtered out before building the index (if the range is known in advance). We assume that (a) the data has already been loaded into the data structure, and (b) this is a a Write Once Read Many (WORM) workload, and multiple range searches will be satisfied by the same index.

We evaluate the time it takes to perform a range search on each of the tree-based data structures for ranges that cover 25%, 50% and 75% of the group-by cardinality (the smaller ranges are run first). The results are shown in Figure 8. In Figure 8(c), we see that the time to build the tree dominates the runtime. The search times



(a) Rseq (b) Rseq-shf (c) Hhit (d) Hhit-shf (e) MovC (f) Zipf

Figure 9: Scalar Aggregation Q6 - 100M records

shown in Figures 8(a) and 8(b) indicate that Btree significantly outperforms the other algorithms if the tree is prebuilt. This is likely due to the pointers that link each leaf node, resulting in one $O(\log(n))$ lookup operation to find the lower bound of the search, and a series of pointer lookups to complete the search. At low cardinality (10^3 groups), the range search time for ART is 12% lower than Judy, but it is 94% higher at high cardinality (10^6 groups). If we factor in the build time and consider the workload WORO, then ART is the fastest algorithm.

5.7 Results - Scalar Aggregation

Unlike Vector aggregate functions, which output a row for each unique key, Scalar aggregate functions return a single row. We evaluate Q6 on the tree-based and sort-based aggregation algorithms. Hash tables are unsuitable for this query because the keys need to be in lexicographical order to calculate the median. Figure 9 shows the query execution time for Q6 with different datasets. The overall winner of this workload is the Spreadsort algorithm. In the case of a dynamic or WORM workload, a tree-based algorithm would have two advantages of faster lookups and requiring considerably less computation for new inserts. A good candidate for tree-based scalar aggregation is Judy, as it outperforms Introsort on all the datasets, and comes close to the

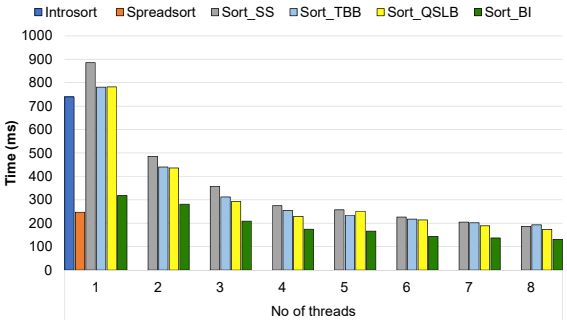
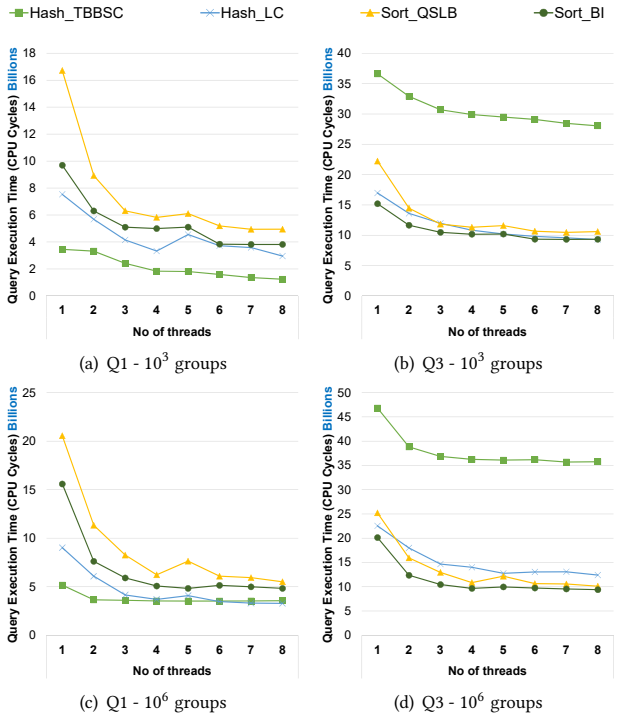


Figure 10: Parallel Sort Algorithm Microbenchmark



(a) Q1 - 10^3 groups (b) Q3 - 10^3 groups

(c) Q1 - 10^6 groups (d) Q3 - 10^6 groups

Figure 11: Multithreaded Scaling - Rseq 100M

performance of Spreadsort in three out of the six datasets. Although ART wins over Judy in some cases, it is inconsistent and its worse case performance is significantly worse, rendering it a poor candidate for this workload. The conclusion is in line with our expectation. To calculate the scalar median of a set of keys, Spreadsort is the fastest algorithm. If an index has already been built then Judy is usually the quickest in producing the answer.

5.8 Multithreaded Scalability

A concurrent algorithm's ability to provide a performance advantage over a serial implementation depends on two main factors: the problem size, and the algorithmic efficiency. Considerations pertaining to algorithmic efficiency include various overheads associated with concurrency, such as contention and synchronization. In order to implement concurrent aggregate operators, a suitable data structure must fulfill three requirements. Firstly, they must be designed for data-level parallelism that can scale with an increasing number of threads. Secondly, they must support thread-safe insert and update operations. It is not uncommon to encounter data structures that support concurrent *put* and *get* operations, but provide no way to safely modify existing values. Lastly, they must provide a means to iterate through their

Table 8: Concurrent Algorithms and Data Structures

Label	Type	Description
Hash_TBBSC	Hash	TBB Separate Chaining (Concurrent Unordered Map [35])
Hash_LC	Hash	Intel Libcuckoo [29]
Sort_BI	Sort	Block Indirect Sort [42]
Sort_QSLB	Sort	Quicksort with Load Balancing (GCC Parallel Sort [43])

content, preferably without requiring prior knowledge of the range of values. In this section, we evaluate the performance and scalability of concurrent data structures and algorithms which fulfill all three criteria. We considered and ultimately rejected several candidate tree data structures. HOT [8] does not support concurrent incrementing of values (needed by Q1) or multiple values per key (needed by Q3). BwTree [28, 46] is a concurrent B+Tree originally proposed by Microsoft. However, our preliminary experiments found its performance to be very poor, as limitations in its API prevent efficient update operations. These characteristics have been discovered by other researchers as well [47]. The concurrent variant of ART [27] currently lacks any form of iterator, which is essential to our workloads.

We use a microbenchmark to select two parallel sorting algorithms from among four candidates. We vary the number of threads from one to eight, and include the two fastest single-threaded sorting algorithms for comparison. The workload consists of sorting random integers between 1-1M, similar to the microbenchmark presented in Section 3. The results are shown in Figure 10. Sort_BI is a novel sorting algorithm, based on the concept of dividing the data into many parts, sorting them in parallel, and then merging them [42]. Sort_TBB is a Quicksort variant that uses TBB task groups to create worker threads as needed (up to number of threads specified). Sort_SS (Samplesort) [42] is a generalization of Quicksort that splits the data into multiple buckets, instead of dividing the data into two partitions using a pivot. Lastly, Sort_QSLB [43] is a parallel Quicksort with load balancing. Considering the performance and scalability at 8 threads, we select the Sort_BI and Sort_QSLB algorithms to implement sort-based aggregate operators.

We selected four concurrent algorithms and algorithms, listed in Table 8, all of which are actively maintained open-source projects. We introduced Hash_LC in Section 3, and Hash_TBBSC is a concurrent separate chaining hash table that is similar to Hash_SC. We evaluate the multithreaded scaling for Q1 and Q3, on both low and high dataset cardinality. The results are depicted in Figure 11. We observe that both hash tables are faster in Q1, and Hash_TBBSC outperforms Hash_LC regardless of the cardinality. Sort-based approaches take the lead in Q3. The gap between sorting and hashing increases at higher cardinalities. This echoes our previous single-threaded results. The performance of Hash_TBBSC degrades significantly in Q3, because storing the values requires the use of a concurrent data structure (in this case a concurrent vector) as the value type. This is a limitation of the hash table implementation, which results in additional overhead due to synchronization and fragmentation [35]. We also considered implementing Q3 using TBB's concurrent multimap, but the performance was significantly worse. Hash_LC does not suffer from these issues, as it provides an interface for user-defined *upsert* functions. We observe similar trends with other data distributions, which we omit here due to space constraints.

6 SUMMARY AND DISCUSSION

Based on the insights we gained from our experiments, we present a decision flow chart in Figure 12 that summarizes our main observations with regards to the algorithms and data structures. We acknowledge that our experiments do not cover all possible situations and configurations, and our conclusions are based on these computational results and observations.

We start by picking a branch depending on the output format of the aggregation query. If the query is scalar, the workload determines the best algorithm. If query workload is "Write Once Read Once" (WORO) then the Spreadsort algorithm provides the fastest overall runtimes. If we require a reusable data structure that can satisfy multiple queries of this category, then Judy is a more suitable option. Going back to the start node, if the aggregation query is vector, our decision is determined by the aggregate function category. Holistic aggregates (such as Q3) are considerably faster, and more memory efficient with the sorting algorithms, particularly Spreadsort (single-threaded) and Sort_BI (multithreaded). This advantage is more noticeable at high group-by cardinality. If the query is distributive (such as Q1) then our experiments show that Hash_LP (single-threaded) and Hash_TBBSC (multithreaded) are the fastest algorithms. For aggregate queries that include a range condition, we found that Btree greatly outperformed the other algorithms in terms of search times. This advantage is only relevant if we assume that the tree has been prebuilt. Otherwise, ART is the best performer in this category, due to its advantage in build times.

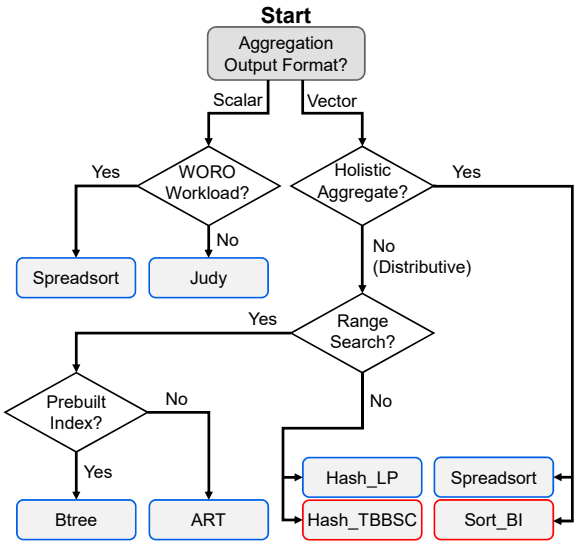


Figure 12: Decision Flow Chart

7 RELATED WORK

There have been a broad range of studies on the topic of aggregation. With the growing popularity of in-memory analytics in recent years, memory-based algorithms have gained a lot of attention. We explore some of the work that is thematically close to our research.

Some studies have proposed novel index structures for database operations. Notably, recent studies have looked into replacing comparison trees with radix trees. In [26] Leis et al. proposed an adaptive radix tree (ART) designed for in-memory query processing. The authors evaluated their data structure with the TPC-C benchmark, which does not focus on analytical queries or aggregation. Based on a similar concept Binna et al. propose HOT [8] (Height Optimized Trie). The core concepts behind this approach are reducing the height of the tree on sparsely distributed keys, and the use of AVX2 SIMD instructions for intra-cycle parallelism. The authors demonstrate that HOT significantly outperforms other indexes, such as ART [26], STX B+tree [7], and Masstree [31], on insert/read workloads with string keys. However, for integer keys, ART maintains a notable performance advantage in insert performance, and is competitive in read performance.

The duality of hashing and sorting for database operations is a topic that continues to generate interest. The preferred approach has changed many times as hardware, algorithms, and data have evolved over the years. Early database systems relied heavily on sort-based algorithms. As memory capacity increased, hash-based algorithms started to gain traction [16]. In 2009 Kim et al. [24] compared cache-optimized sorting and hashing algorithms and concluded that hashing is still superior. In [38] Satish et al. compared several sorting algorithms on CPUs and GPUs. Their experiments found that Radix Sort is faster on small keys, and Merge Sort with SIMD optimizations is faster on large keys. They predicted that Merge Sort would become the preferred sorting method in future database systems. It can be argued that this prediction has yet to materialize.

Müller et al. proposed an approach to hashing and sorting with an algorithm that can switch between them in real time [32]. The authors modeled their algorithm based on their observation that hashing performs better on datasets with low cardinality, but sorting is faster when there is high data cardinality. This holds true with some basic hashing or sorting algorithms, but there are algorithms for which this model does not apply. Their approach adjusts to the cardinality and skew of the dataset by setting a threshold on the number of groups found in each cache-sized chunk of the data. This approach cannot be used for holistic aggregation queries, as the data is divided into chunks.

Balkesen et al. [2] compared the performance of highly optimized sort-merge and radix-hashing algorithms for joins. Their implementations leveraged the extra SIMD width and processing cores found in modern processors. They found that hashing outperformed sorting, although the gap got much smaller for very large datasets. The authors predicted that sorting may eventually outperform hashing in the future, if the SIMD registers and data key sizes continue to expand.

Parallel aggregation algorithms focus on determining efficient concurrent designs for shared data structures. A key question in parallel aggregation is whether threads should be allowed to work independently, or to work on a shared data structure. Cieslewicz et al. [11] present a framework to select a parallel strategy based on a sample from the dataset. Surprisingly, the authors claim that sort-based aggregation can only be faster than hash-based aggregation if the input is presorted. We found that in the context of single threaded algorithms, sort-based aggregation is quite competitive with hash-based.

In [49], the authors examined several previously proposed parallel algorithms, and propose a new algorithm called PLAT based on the concept of partitioning, and a combination of local and global hash tables. Most of these algorithms do not support holistic aggregation, because they split the data into multiple hash tables in order to reduce contention. Furthermore, none of the algorithms are ideal for scalar aggregation as they do not guarantee lexicographical ordering of the keys.

8 CONCLUSION

Aggregation is an integral aspect of big data analytics. With rising RAM capacities, in-memory aggregation is growing in importance. There are many different factors that can affect the performance of an aggregation workload. Knowing and understanding these factors is essential for making better design and implementation decisions.

We presented a six dimensional analysis of in-memory aggregation. We used microbenchmarks to assess the viability of 20 different algorithms, and implemented aggregation operators using 14 of those algorithms. Our extensive experimental framework covered a wide range of data structures and algorithms, including serial and concurrent implementations. We also varied the query workloads, datasets, and the number of threads. We gained a lot of useful insights from these experiments. Our results show that some persisting notions about aggregation do not necessarily apply to modern hardware and algorithms, and that there are certain combinations that work surprisingly better than conventional wisdom would suggest (see Figure 12).

To our knowledge, this is the first performance evaluation that conducted such a comprehensive study of aggregation. We demonstrated with extensive experimental evaluation that the ideal approach in a given situation depends on the input and the workload. For instance, sorting-based approaches are faster in holistic aggregation queries, whereas hash-based approaches perform better in distributive aggregation queries.

REFERENCES

[1] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. 2015. A comparison of adaptive radix trees and hash tables. In *ICDE*. IEEE, 1227–1238.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *VLDBJ* 7, 1 (2013), 85–96.

[3] Çağrı Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2015. Main-memory hash joins on modern processor architectures. *TKDE* 27, 7 (2015), 1754–1766.

[4] Doug Baskins. 2002. General purpose dynamic array - Judy. http://judy.sourceforge.net/index.html.

[5] R Bayer and E Mccreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1 (1972), 173–189.

[6] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in Parallel Query Processing. In *PODS*. ACM, 212–223.

[7] Timo Bingmann. 2013. STX B+ Tree C++ Template Classes. https://github.com/bingmann/stx-btree.

[8] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD*. ACM, 521–534.

[9] Spyros Blanas, Yinan Li, and Jignesh M Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*. ACM, 37–48.

[10] Peter Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. 2017. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 103–119.

[11] John Cieslewicz and Kenneth A Ross. 2007. Adaptive aggregation on chip multiprocessors. In *VLDBJ*. 339–350.

[12] Transaction Processing Performance Council. 2017. TPC-H benchmark specification 2.17.3. http://www.tpc.org/tpch.

[13] Alain Crolotte and Ahmad Ghazal. 2012. Introducing Skew into the TPC-H Benchmark. In *TPCTC*. 137–145.

[14] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing.. In *NSDI*, Vol. 13. 371–384.

[15] Edward A Fox, Qi Fan Chen, Amjad M Daoud, and Lenwood S Heath. 1991. Order-preserving minimal perfect hash functions and information retrieval. *TOIS* 9, 3 (1991), 281–308.

[16] Goetz Graefe, Ann Linville, and Leonard D. Shapiro. 1994. Sort vs. hash revisited. *TKDE* 6, 6 (1994), 934–944.

[17] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* 1, 1 (1997), 29–53.

[18] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. 1994. Quickly generating billion-record synthetic databases. In *SIGMOD*. ACM, 243–252.

[19] C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *CACM* 4, 7 (1961), 321.

[20] Peng Jiang and Gagan Agrawal. 2017. Efficient SIMD and MIMD parallelization of hash-based aggregation by conflict mitigation. In *ICS*. ACM, 24.

[21] Nicolai M Josuttis. 2012. *The C++ standard library: a tutorial and reference*. Addison-Wesley.

[22] Thomas Kejser. 2014 (accessed June 16, 2017). TPC-H Schema and Indexes. http://kejser.org/tpc-h-schema-and-indexes/.

[23] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE, 195–206.

[24] 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. *VLDBJ* 2, 2 (2009), 1378–1389.

[25] Tobin J Lehman and Michael J Carey. 1986. A study of index structures for main memory database management systems. In *VLDB*, Vol. 1. 294–303.

[26] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. IEEE, 38–49.

[27] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*. ACM, 1–8.

[28] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. IEEE, 302–313.

[29] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys*. ACM, 27:1–27:14.

[30] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. 2017. Fast and scalable minimal perfect hashing for massive key sets. *CoRR* (2017). arXiv:1702.03154 Retrieved from http://arxiv.org/abs/1702.03154.

[31] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Eurosys*. ACM, 183–196.

[32] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-efficient aggregation: Hashing is sorting. In *SIGMOD*. 1123–1136.

[33] David R Musser. 1997. Introspective sorting and selection algorithms. *Software: practice and experience* 27, 8 (1997), 983–993.

[34] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *ESA*. Springer, 121–133.

[35] Chuck Pheatt. 2008. Intel® threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.

[36] David MW Powers. 1998. Applications and explanations of Zipf's law. In *NeMLaP3/CoNLL98*. Association for Computational Linguistics, 151–160.

[37] Steven J Ross. 2002. The Spreadsort High-performance General-case Sorting Algorithm.. In *PDPTA*. 1100–1106.

[38] 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*. ACM, 351–362.

[39] Anil Shanbhag, Holger Pirk, and Sam Madden. 2016. Locality-Adaptive Parallel Hash Joins using Hardware Transactional Memory. In *Data Management on New Hardware*. Springer, 118–133.

[40] Ambuj Shatdal and Jeffrey F Naughton. 1995. Adaptive parallel aggregation algorithms. In *SIGMOD*. ACM, 104–114.

[41] Craig Silverstein. 2005. Implementation of Google sparse_hash_map and dense_hash_map. https://github.com/sparsehash/sparsehash.

[42] Steven Ross, Francisco Tapia, and Orson Peters. 2018. Boost C++ Library 1.67. www.boost.org.

[43] GCC Team et al. 2018. Gcc, the gnu compiler collection. http://gcc.gnu.org.

[44] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 18–32.

[45] Brett Walenz, Sudeepa Roy, and Jun Yang. 2017. Optimizing Iceberg Queries with Complex Joins. In *SIGMOD*. ACM, 1243–1258.

[46] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a Bw-tree takes more than just buzz words. In *SIGMOD*. ACM, 473–488.

[47] Zhongle Xie, Qingchao Cai, Gang Chen, Rui Mao, and Meihui Zhang. 2018. A Comprehensive Performance Evaluation of Modern in-Memory Indices. In *ICDE*. 641–652.

[48] Weipeng P Yan and Per-Ake Larson. 1995. Eager aggregation and lazy aggregation. In *VLDB*, Vol. 95. 345–357.

[49] Yang Ye, Kenneth A Ross, and Norases Vesdapunt. 2011. Scalable aggregation on multicore processors. In *DMSN*. 1–9.

[50] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*. ACM, 247–260.