Cache-Efficient Top-k Aggregation over High Cardinality Large Datasets

Tarique Siddiqui
Microsoft Research
Redmond, Washington, USA
tasidd@microsoft.com

Vivek Narasayya
Microsoft Research
Redmond, Washington, USA
viveknar@microsoft.com

Marius Dumitru
Microsoft
Redmond, Washington, USA
mariusd@microsoft.com

Surajit Chaudhuri
Microsoft Research
Redmond, Washington, USA
surajitc@microsoft.com

ABSTRACT

Top-k aggregation queries are widely used in data analytics for summarizing and identifying important groups from large amounts of data. These queries are usually processed by first computing exact aggregates for all groups and then selecting the groups with the top-k aggregate values. However, such an approach can be inefficient for high-cardinality large datasets where intermediate results may not fit within the local cache of multi-core processors leading to excessive data movement. To address this problem, we have developed Zippy, a new cache-conscious aggregation framework that leverages the skew in the data distribution to minimize data movements. This is achieved by designing cache-resident data structures and an adaptive multi-pass algorithm that quickly identifies candidate groups during processing, and performs exact aggregations for these groups. The non-candidate groups are pruned cheaply using efficient hashing and partitioning techniques without performing exact aggregations. We develop techniques to improve robustness over adversarial data distributions and have optimized the framework to reuse computations incrementally for rolling (or paginated) top-k aggregate queries. Our extensive evaluation using both real-world and synthetic datasets demonstrate that Zippy can achieve a median speed-up of more than 3× for monotonic aggregation functions across typical ranges of k values (e.g., 1 to 100) and 1.4× for non-monotonic functions when compared with state-of-the-art cache-conscious aggregation techniques.

PVLDB Reference Format:
Tarique Siddiqui, Vivek Narasayya, Marius Dumitru, and Surajit Chaudhuri. Cache-Efficient Top-k Aggregation over High Cardinality Large Datasets. PVLDB, 17(4): 644 - 656, 2023.
doi:10.14778/3636218.3636222

1 INTRODUCTION

Business Intelligence (BI) tools such as Power BI [2] and Tableau [3] make it easy for users to analyze large amounts of data. In these tools, top-k aggregation queries are used to aggregate data in large multi-core main-memory databases and present the most significant

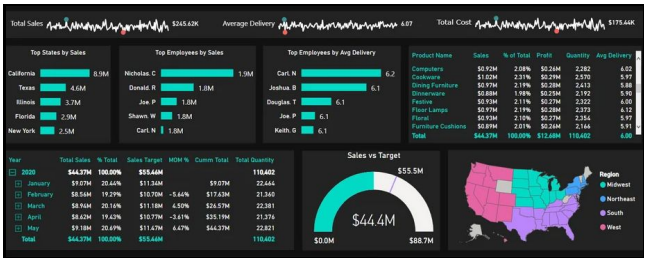


Figure 1: A BI dashboard depicting top-k aggregate queries.

results to users. These queries are a common feature in BI analytics, with most dashboards and reports often visualizing one or more top-k aggregate results as depicted in Figure 1.

The standard approach for processing top-k aggregation queries is to first compute the *exact aggregates for all groups*, followed by the selection of groups with top-k aggregate values. When the number of groups in the dataset is small, aggregation can be very fast; however, when the cardinality of groups surpasses the CPU's local cache size (L1 and L2), the performance drops noticeably due to a significant amount of data movement, i.e., the cache-line transfers between local cache and L3/main-memory. There has been work on cache-conscious aggregation algorithms [11, 25, 35] to minimize data movement in multi-core settings. However, such algorithms are not optimized for top-k and computing exact aggregates for every grouping key consumes a substantial proportion of the overall computation for *high cardinality large datasets*.

In this work, we investigate *top-k optimized aggregation* techniques that can efficiently compute the exact values for top-k aggregates *without fully aggregating all groups* in the dataset. Our motivation for such optimizations comes from the observation that real-world data distributions typically follow a skewed distribution [26] where a few groups have much higher aggregate values than the rest of the groups. For example, Figure 2 shows the CDF of standard aggregate (i.e., sum, count, max, min, avg) for groups from a real-world dataset used in Power BI [2], consisting of approximately 200 million tuples and 30 million groups. We see that top 1% of the groups have at least 4 orders of magnitude higher aggregate values than the rest of the groups. Furthermore, the value of k used in BI is generally small (e.g., 1 to 20). Fully aggregating 10s of millions of groups with skewed distribution of aggregate values only to return a few groups is not only time-consuming, but also performs potentially unnecessary computations.

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636222

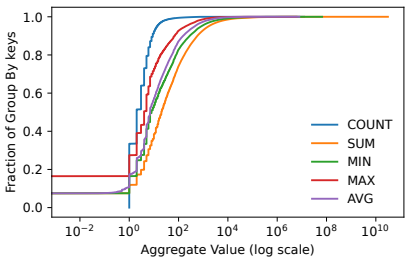


Figure 2: CDF of group-by aggregate values over a high cardinality customer dataset consisting of 200 M tuples and 30 M unique groups.

While previous research has addressed top-k optimization in relational databases and information retrieval [10, 18], there has been limited work on cache-conscious optimizations specifically targeting top-k aggregates in main-memory settings, which present unique challenges and trade-offs as discussed in subsequent sections. Some databases and distributed systems, such as Microsoft SQL Server and Apache Arrow DataFusion [1], support top-k push-down during parallel aggregation and sorting across cores. These optimizations aim to minimize the transfer of all groups from individual cores when only the top-k results are required. However, they still require fully aggregating each grouping key at every core before any keys can be pruned. Additionally, these optimizations assume that each core contains disjoint groups, which introduces significant data movement overhead when partitioning the data to ensure the same groups are located on the same node.

Our approach. In this work, we develop Zippy, a new cache-conscious aggregation technique that can efficiently identify top-k aggregates. Our primary focus is on monotonic aggregation functions such as sum, count, min, and max, which are commonly used in business intelligence applications to summarize large datasets. These functions exhibit the property that the aggregate value for a given group increases or decreases as more tuples belonging to the group are aggregated. Leveraging the monotonicity property allows us to design efficient algorithms. However, achieving a balance between minimizing data movement and early pruning of irrelevant groups while ensuring the accuracy of the final top-k results poses a significant challenge.

We observe that for high-cardinality large datasets, a cache-efficient algorithm that performs multiple passes by partitioning and re-partitioning the data [25] tends to outperform approaches that make only one or two passes [11, 35], due to reduced cache line transfers. However, partitioning dominates the overall cost in such scenarios and our goal in this work is to reduce this cost. We adopt a strategy of early partition pruning, removing partitions that are less likely to contain top-k results. However, determining which partitions to prune early is challenging.

In order to prune partitions early, we sample a subset of the data and analyze it to select an initial set of promising groups, referred to as candidate groups. We also employ these samples to validate whether the data distribution exhibits sufficient skewness suitable for top-k optimization. In cases where the distribution in not skewed, we revert to the standard approach of computing exact aggregates for all groups. Notably, even in large datasets comprising 100s of millions of tuples, the cost of sampling a smaller subset remains negligible.

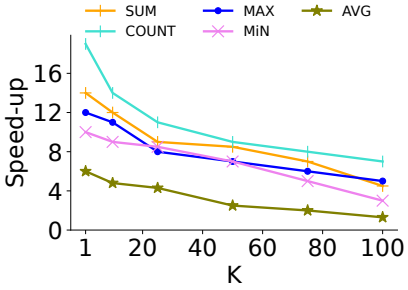


Figure 3: Speed-up of top-k optimized aggregation over state-of-the-art cache-conscious aggregation algorithm [25] for different values of k over a real data distribution depicted in Figure 2.

After identifying the initial candidate groups, we focus on optimizing multi-pass aggregation such that we simultaneously prune irrelevant partitions and compute exact aggregate values efficiently. We leverage *two cache-resident structures* to achieve this. The first structure employs optimized hash tables to store candidate groups likely to be in the top-k, enabling efficient and early computation of exact aggregates for these groups. The second structure computes coarse-grained statistics for groups less likely to be in the top-k and utilizes the software-write combining technique [6, 25, 34] to efficiently partition and evict these groups to the main memory when the cache is full. After each pass on the data, we analyze the intermediate results to compute bounds, and prune partitions that are guaranteed to not contain the top-k groups. This pruning process significantly speeds up subsequent passes.

To further reduce partitioning overhead, we minimize physical partitioning and maintain statistics for each partition hash that allow us to compute bounds on aggregates for all groups belonging to the same partition. As a result, many groups can be pruned without the need for physical partitioning. Physical partitions are only created when there is a lower likelihood of pruning.

In summary, our approach combines the above ideas to develop a multi-pass aggregation algorithm specifically tailored for high-cardinality large datasets. The performance of top-k aggregate queries is significantly improved compared to state-of-the-art cache-conscious aggregation techniques [25], as illustrated in Figure 3. Furthermore, we extend our technique to accommodate rolling or paginated top-k queries, allowing users to incrementally increase the value of k. Our contributions can be summarized as follows:

- We address the problem of improving cache efficiency in top-k aggregation for high-cardinality large datasets, which has received limited attention in multi-core main memory settings (Section 2).
- We analyze the trade-offs involved in key sub-routines such as hashing and partitioning when processing top-k aggregate queries, identifying optimization opportunities (Section 3).
- We propose a multi-pass aggregation algorithm Zippy that efficiently finds top-k aggregates while maximizing throughput (Section 4).
- We extend Zippy to support rolling top-k queries, allowing users to query top-k using moving windows (Section 5).
- We perform an extensive evaluation of Zippy on real-world and synthetic datasets. Our results show that Zippy yields a median speed-up for more than 3× for a practical range of k values (e.g.,

1 to 100) for monotonic aggregation functions and 1.4× for non-monotonic functions when compared with state-of-the-art cache-conscious aggregation techniques (Section 7).

2 PROBLEM SETUP AND ASSUMPTIONS

We consider an ad hoc OLAP data analytic setting over a multi-core main-memory system. Given a table R, consisting of dimension attribute X and measure attribute Y, we want to maximize the throughput of top-k aggregate queries with the following template:

```
SELECT X, AGG (Y) AS A
FROM R
GROUP BY X
ORDER BY A
LIMIT k
```

Our primary focus is on monotonic aggregation functions such as COUNT(*), MAX(Y), MIN(Y), and SUM(Y) with Y $\geq$ 0 where the aggregate value for a given group either increases or decreases as the tuples belonging to the group are aggregated. Such monotonic aggregation functions are widely used in BI. The challenge lies in developing *cache-resident structures and a cache-conscious algorithm that effectively utilizes the monotonicity property to reduce cache-line transfers*. Although we mainly focus on queries with a single GROUP BY attribute for ease of exposition, our techniques can be extended to support multiple GROUP BY attributes, aggregation over expressions of multiple attributes, selection predicates, as well as PK-FK joins, as described in Section 7.

Rolling Top-k. We also consider rolling top-k queries where the results are retrieved in smaller, discrete pages rather than all at once. For instance, users may see the first 10 results, then the next 10, and so on. Such queries improve performance as well as user experience when dealing with a large number of results.

2.1 Assumptions

We assume the availability of statistics such as the size of the dataset, the number of distinct values of columns, the minimum value, and the maximum value of a column. In some cases we may have access to a sample of the dataset; however, we include the sampling overhead as part of the query processing time in this work.

High Cardinality Large Datasets. Let N be the size of the table, M be the number of unique groups, and C be the number of groups (consisting of key-aggregate pairs) that can be accommodated in the local cache (L1 and L2). We target a setting where the input dataset is large and has high cardinality, i.e., N >> M >> C. The real-world datasets typically have a skew in grouping column values, as well as a skew in aggregate column values. Finally, we assume that the value of k is significantly lower (e.g., < 100) than the number of unique groups, as is typical in BI.

No indexing or data partitioning support on grouping attributes. Considering an ad hoc exploration setting, we do not assume that the input data is already partitioned (e.g., hash or range partitioned) on the grouping attribute nor do we assume the ordering of tuples in R on X. Similarly, we do not assume that the availability of an index for retrieving random samples from R corresponds to different values of X, as is typical in main-memory databases used in BI.

3 OVERVIEW OF EXISTING TECHNIQUES AND OPPORTUNITIES

We first give an overview of existing multi-core aggregation algorithms and then discuss the opportunities for improvement for top-k optimizations. Multi-core aggregations can be broadly classified into two classes: few-pass algorithms which make a fixed number of passes on the data and the multi-pass algorithms which recursively partition (with local aggregation wherever possible) the data until the partition fits into the local cache.

To illustrate our discussion, we consider a real-world distribution referred to as `RealD1` in the experimental section (Section 7). This distribution consists of $N = 200$ million tuples, with a cardinality (M) of 30 million. We assume a computational setup consisting of 24 cores, each with combined L1 and L2 caches capable of accommodating approximately 1 million groups of key-aggregate pairs and a cache line size (B_i) of 64 bytes. We further assume that input data are segmented into uniform-sized blocks that are equally distributed among the cores. For RealD1, each core processes approximately $N_i = 8$ million tuples, consisting of approximately 2 million unique groups (M_i), and the local cache can hold approximately $C_i = 50,000$ groups. In our analysis, we will explore trade-offs among various algorithms by examining cache-line transfers within the context of a single core (with similar behavior across other cores).

3.1 Few-Pass Multi-Core Aggregation

We begin by providing an overview of aggregation algorithms developed by Cieslewicz and Ross [11] and Ye et al. [35] for large multi-core main-memory settings.

ATOMIC. This approach employs a shared hash table protected by atomic instructions, with all cores working together. However, contention due to concurrent updates to aggregate values by multiple threads results in multiple compare-and-swap attempts. To address these contention issues, the following strategies have been developed aimed at reducing work sharing among cores.

INDEPENDENT. In the first pass, each thread creates a private hash table based on its part of the input. In the second pass, these hash tables consisting of partial aggregates are merged. However, this approach incurs substantial data movement overhead when the private hash table size exceeds the cache capacity. Theoretically, after the local cache is full, there is cache line transfer happening with a probability of $(1 - C_i/M_i)$ which can be significantly large for high cardinality datasets. For instance, for RealD1, after reading a few 100K of tuples, there is .98 probability of a cache line transfer on reading every additional tuple, resulting in orders of millions of cache line transfers after reading 8 million tuples.

HYBRID. Each thread aggregates its part of the input into a private hash table sized according to its local cache. Once this table reaches its capacity, older entries are evicted (similar to an LRU cache) and inserted into a global shared hash table. Similarly to the INDEPENDENT approach, this method becomes less efficient when a significant portion of the output cannot fit into the private hash tables. While this approach improves upon INDEPENDENT, for high cardinality datasets, the number of cache-line transfers is roughly in the same order.

PLAT. This approach involves performing aggregation in private hash tables whenever possible and overflowing additional data to

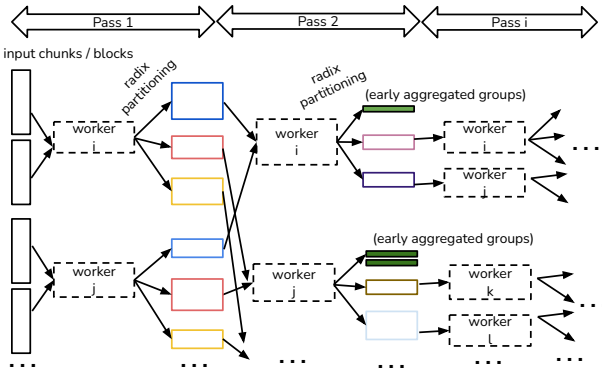


Figure 4: Illustrating Multi-Pass Aggregation [25] (Green boxes depict aggregated tuples).

partitions for later merging. However, even with data moved to partitions, the number of groups may still be large and may not fit in the local cache during the aggregation in the second pass.

3.2 Multi-Pass Multi-Core Aggregation

The aforementioned algorithms involve a fixed number of passes, which leads to high data movements over large datasets with a high cardinality of groups. To address this, Mueller et al. [25] extend PLAT to multiple passes targeting high-cardinality large dataset settings. Figure 4 illustrates this algorithm. Each core reads an input chunk and employs radix partitioning to create child partitions where each child partition has fewer distinct groups. During partitioning, each new partition is allocated to a cache line. Once a cache line reaches its capacity, the contents are efficiently moved to memory using non-temporal store instructions [6, 34]. This recursive partitioning continues until a partition has a small number of distinct groups to fit into the local cache. At this point, either hashing or a sorting-based aggregation is initiated. The higher the cardinality, the more passes are performed on the data. The framework also leverages the locality of groups in the partition to aggregate and sort the data earlier to further minimize cache-line transfers.

Analysis: Given a cache line size of B_i, each pass generates C_i/B_i new partitions. The number of passes required can be expressed as $log_{C_i/B_i} N_i/B_i$, where each pass accounts for $2 \times N_i/B_i$ cache line transfers, considering both reading and writing operations. For our running example, this approach leads to a reduction in the number of cache line transfers to approximately a few hundred thousands, significantly improving over the algorithms discussed previously. However, partitioning dominates the overall cost ($log_{(C_i/B_i)} N_i/B_i$ passes) in this algorithm. In our work, we build on this algorithm to support top k optimizations, effectively minimizing the number of partitions created during the process.

3.3 Opportunities for Top-k Optimization

We compare aggregation algorithms as well as examine the trade-offs in sub-routines involved in aggregation. This guides the design of optimizations for top-k aggregation, detailed in the following section. We first describe the implementation of hash tables and partitioning.

Hash table. We use a single-level hash table with linear probing and fix the hash table to the size of the L2 cache and consider it

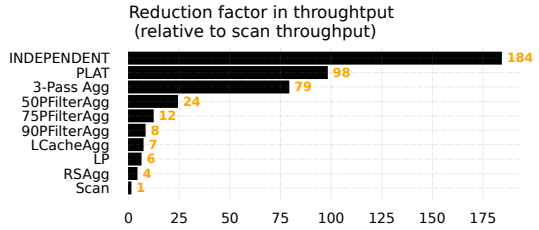


Figure 5: Understanding relative performance of operations relevant to top-k aggregate optimization.

full at a very low fill rate of 50%. As a hash function, we use 64-bit MurmurHash2, which is fast and has low CPU overhead.

Partitioning. We use radix partitioning using the software-managed buffering technique as described in [6, 34]. Essentially, we buffer one partition per cache line for each core and write it to the memory once full using non-temporal store instruction This technique has minimum overhead in terms of TLB misses during partitioning.

We perform a microbenchmark on three large, high-cardinality real-world data over a test-up described in Section 6. The dataset size ranges between 100-400 million with cardinality between 20-60 million. The on-chip cache across all cores can efficiently aggregate between 1-2 million groups after which performance degrades drastically. We compare the performance of the following three algorithms with different passes on the data: INDEPENDENT (1-Pass), PLAT (2-Pass) and 3-Pass Agg (multi-pass aggregation with three passes). We also consider the following operations which make the building blocks of our algorithm as we will see in the next section.

- `RSAgg`: Randomly sample 1% of the dataset and perform aggregation over sampled data using INDEPENDENT.
- `LP`: Compute the hash for each tuple and update the following statistics: sum, count, min, and max for each hash. We call this operation logical partitioning and set the number of partitions such that they fit within the local cache.
- `LCacheAgg`: Ignoring partitioning time in 3-Pass Agg, we measure only the time in aggregating the physical partitions once they fit within the local cache.
- `50PFilterAgg`: Applies physical partitioning and aggregation only over 50% of partitions after pass 1 in 3-Pass Agg. The selected 50% of the partitions contain the top-k results and are identified using a pre-processing step. The computation time of this pre-processing step is ignored. In other words, this operation measures the cost of partitioning and aggregating the most relevant partitions.
- `75PFilterAgg`: Similar to `50PFilterAgg` with the difference that 75% of the partitions are pruned.
- `90PFilterAgg`: Similar to `50PFilterAgg` with the difference that 90% of the partitions are pruned.

Figure 5 depicts the reduction factor in throughput for each of the above operations relative to the average throughput for the scan operation on the same datasets. We draw two conclusions. First, operations such as scan, sampling, logical partitioning, and their combined costs are significantly faster than physically partitioning the data. Second, if we can efficiently identify a small set of partitions that are likely to contain the top-k results, we can achieve substantial cost savings compared to processing all partitions. Based on these observations, we discuss an optimized top-k

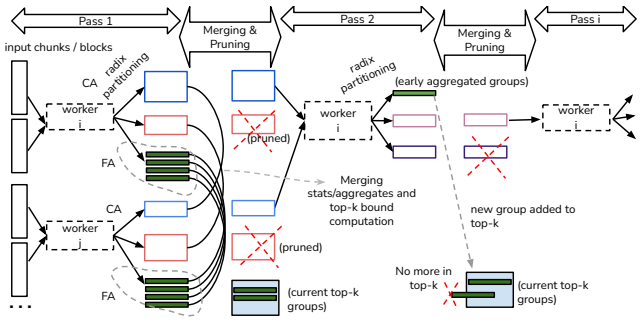


Figure 6: A simplified illustration of multi-pass aggregation with top k optimization. The sampling-based candidate selection phase is not shown.

aggregation algorithm next to improve performance on skewed data distributions.

4 TOP-K AWARE AGGREGATION

We begin by giving an high-level overview of the framework and then discuss the key techniques integral to the framework.

4.1 Overview of the Framework

We first introduce cache-optimized data structures to minimize cache-line transfers. Then, we discuss a multi-pass algorithm for efficient pruning of partitions that do not contain the top-k groups.

4.1.1 Cache-Resident Structures

We create two cache-resident structures: Fine-grained Aggregates (FA) and Coarse-grained Aggregates (CA) such that the total size of the two structures is close to the local cache size of the core.

Fine-grained Aggregates. This is used for computing the exact aggregates for candidate groups in a single pass of the data. The candidate groups are those groups that are likely (but not guaranteed) to be in top-k. We discuss shortly how we identify such groups. To compute exact aggregates, we use a single-level hash table with linear probing with a sufficiently large size to minimize collisions. The single-level hash table improves performance by eliminating branching and chaining.

Coarse-grained Aggregates. The CA refers to the partitioning of the data corresponding to non-candidate groups. With each partition, we compute the following small-space statistics: sum, count, max, min, and approximate distinct count (measured using small-space FM algorithm [15]) aggregated over all the tuples falling in the partition. As explained later, these statistics help to decide whether partitions can be safely pruned from further processing during aggregation. A partition can be logical or physical. In *logical partitioning*, we only maintain the statistics using the hash of the partition as a key without any physical movement of tuples. The logical partitions are stored similarly to FA as a single-level hash table. During physical partitioning, besides creating the hash table of statistics, we also move the input tuples to co-located memory locations using radix partitioning. We buffer one partition per cache line and write it to the memory once full using non-temporal store instruction [6, 34]. For the same cache size, we can create more logical partitions than physical partitions, which minimizes cache-line transfers. We discuss in Section 4.3 how we pick the right partitioning approach for a given input partition to achieve maximum efficiency.

We find that an equal-sized allocation of space to FA and CA results in close to maximum efficiency (see Section 6). If C is the size of the local cache of a given core, we set the sizes of FA and CA (denoted by C_f and C_c, respectively) for each core to C/2 by default.

Algorithm 1 Top-k aggregation overall framework

```
1  Input: Input data (split into blocks of fixed size tuples), K, Input size: N,
2  cardinality of grouping keys: M, FA cache size: CF, grouping column (s): X,
3  Aggregation func: A, Aggregate column: Y, Total cache size: C, Maximum
4  number of logical partitions cache can hold: Q
5  System constants: segment size s (Section 4.2), locality constant: ∝o,
6  confidence levels: α, β
7  Output: Top K groups (key-aggregate pairs for each group): top-k groups
8  Procedure TopKAggregation:
9    top-k groups = {}    // maintains top K groups
10   partitions = data   // partitions point to input data blocks in first pass
11   isOptimizable, FAgroups, topkBound =
12                   validateAndIdentifyFAgroups(partitions)
13                     // step 1 in Sec 4.1.2, Algorithm 2
14   if not isOptimizable:  // if true, run baseline
15     exact-aggregates-all-groups = Perform  Multi-pass Aggregation
16                                  // (Section 3.2)
17     top-k groups = Priority-queue-based top-k selection over
18                          exact-aggregates-all-groups
19     return top-k groups
20   exactAggregates = {}   // exact aggregates in FA groups
21   While Size(top-k groups) < K:
22     for each unprocessed partition P-i in partitions: // runs in parallel
23       partialAggregates-i, childPartitions-i
24           = AggregateAndPartition(P-i, FAgroups)
25            // step 2 (Sec 4.1.2) Algorithm 3
26     top-k groups, topkBound, exactAggregates, partitions
27           = MergeAndPrune(exactAggregates (from previous pass),
28           (partialAggregates-i, childPartitions-i) for all i)
29            //  step 3 in Sec 4.1.2, Algorithm 4
30   Return top-k groups
```

4.1.2 Overview of Algorithmic Steps

Algorithm 1 outlines the multi-pass algorithm consisting of three key steps described in Algorithms 2 to 4.

Step 1: Validating skew and identifying FA groups. The first step is to validate if the top-k optimization will improve the performance for a given input distribution and the desired number of top-k groups. For non-skewed distributions or queries with large values of k, the scope for top-k optimization is limited. On the other hand, if top-k optimization is applicable, we identify the candidate groups for FA. Algorithm 2 outlines the working of this step. Each core (core and workers are used interchangeably) scans an input chunk of the data to randomly select tuples and aggregates them on the fly to compute partial sample aggregates (lines 8-14). The partial sample aggregates across cores are merged to compute sample aggregates for the entire input (line 15). We validate the skew in sample aggregates using a confidence interval-based approach and if validated we select FA candidate groups (lines 17-33). We provide details in Section 4.2 on how we decide the sample size, perform validation as well as selection of FA groups.

Step 2: Exact Aggregation and Partitioning. After identifying the FA groups, we adaptively perform aggregation and partitioning

of the data using multi-passes over the input data (Algorithm 3). Figure 6 illustrates this step. In particular, each core reads an input partition and decides whether to perform exact aggregation or further repartition it (lines 5-24). If we can cheaply compute the exact aggregates with minimal cache-line transfers (e.g., when there are few distinct groups or there is enough locality in the occurrences of groups), we perform exact aggregation for the partition. If we are not able to do so, we only perform exact aggregation for groups belonging to FA groups (lines 26-27). We create partitions for the rest of the groups (lines 28-37). An important question to consider is whether to create logical or physical partitions (lines 15-24) that we discuss in Section 4.3. Note that for physical partitions, we create as many buffers as the number of cache lines and move the partitions to the main memory once full, following the software-managed buffer strategy discussed in Section 4.1.2 (lines 34-37). However, in many cases, logical partitioning leads to pruning without any physical partitioning.

Algorithm 2 Validating skew and selecting FA groups

```
1  Input: partitions, and other relevant inputs listed in Algorithm 1
2  Output: IsOptimizable: Is Input amenable to Top-K Optimization,
3    FAgroups: Grouping keys of FA groups
4  Procedure ValidateAndIdentifyFAgroups:
5    // computing aggregates over a sample of data
6    sampleSize = Compute sample size (Section 4.2)
7    sampling probability, p = sampleSize / N
8    for each partition-i in partitions (in parallel):
9      partialSampleAggregates-i = {}
10          // stores standard aggregates for each group in the sample
11     For each tuple in the partition-i:
12       selected = select the tuple with probability p
13       if selected:
14         update partialSampleAggregates by aggregating this tuple
15   sampleAggregates = merge (partialSampleAggregates-i for all i)
16   // checking skew in sample aggregates using CI
17   sampleAggregatesBounds = compute CI bounds for each group
18               in sampleAggregates // Section 4.2
19   Lk = Kth highest lower bound in sampleAggregatesBounds
20   tempGroups = { }
21   For each group in sampleAggregatesBounds:
22     If the lower bound of the group >= Lk, add the group to tempGroups
23   If size of tempGroups > CF:
24     isOptimizable = False // number of candidate groups much larger
25                 // than we can efficiently aggregate in FA
26     Return isOptimizable, {}
27   // otherwise we can optimize. If candidate groups smaller than FA,
28   // we can fill leftover space in FA with  heavy hitters
29   FAgroups = tempGroups
30   While the size of FAgroups < CF:
31     g = select the group (not already in FAgroups) with the
32       highest sample count aggregate in sampleAggregates
33     Add g to FAgroups
34   Return isOptimizable, FAgroups
```

Step 3: Merging intermediate results across cores and validating top-k results. After each pass on the data, we perform the merging of partial aggregates of FA groups and aggregates for newly created (child) partitions across all cores (Algorithm 4). After merging, we know the exact aggregates for FA groups. We use the partition aggregates to establish the upper bound on the aggregate values for non-candidate groups (lines 9-11). In particular, the partition-level sum provides an upper bound for sum and average aggregates, while the maximum value sets an upper bound for max and min aggregates. Using the exact aggregates and minimum values in the partition, we compute the threshold *topKBound* on the smallest possible value of kth aggregate for pruning partitions (line 12). We eliminate any partitions with upper bounds on aggregate values that are less than *topKBound* from further processing during subsequent passes (lines 17-18).

The partition-level statistics while taking less space can effectively prune partitions in scenarios where skewed distributions feature a long tail of groups with significantly smaller values. Moreover, segregating FA groups helps reduce the number of tuples sent to partitions in CA, leading to improved pruning. While more sophisticated sketches could provide better bounds, the associated space and computation overhead in using them can often be substantial leading to excessive cache-line transfers.

In the following sub-sections, we discuss in more detail the sampling process including validation and selection of FA groups for step 1. This is followed by a discussion on how we choose between aggregation and partitioning approaches for step 2.

4.2 Sampling-based Candidates Selection

In our setting, the direct sampling of specific grouping keys is challenging as we do not assume the availability of indexes on grouping columns. Instead, we employ a uniform random sampling strategy that collects samples to mirror the proportionality of groups that have a certain minimum proportion of tuples in the input dataset. This minimum proportion of tuples is defined using a configurable parameter called tolerance level Δ. We assume that groups with proportions below Δ are less likely to be candidate groups and can be disregarded for sampling efficiency.

Let the dataset comprise N tuples, with n_i representing the number of tuples for the i^{th} grouping key. We define the population ratio of the i^{th} group as $r_i = \frac{n_i}{N}$. In our sampled dataset of size s, the number of tuples corresponding to the i^{th} grouping key is n'_i. The population ratio in this sampled dataset is expressed as $r'_i = \frac{n'_i}{s}$. Using an analysis similar to [14], we determine that a sample size $s = \frac{Z^2_{\alpha/2}}{4\Delta^2}$ is adequate to ensure that the deviation in the population ratio $|r_i - r'_i|$ remains within a specified tolerance level Δ, with a confidence level of $1-\alpha$. Formally, $P(|r_i - r'_i| < \Delta, 1 \leq i \leq k) > 1-\alpha$ when $s \geq \frac{Z^2_{\alpha/2}}{4\Delta^2}$.

While Δ can be adjusted according to skewness in the input data, for simplicity, we set its default value to $\max(C, 0.01\% \times$ estimated average group size), where C represents the maximum number of groups in the local cache. Consequently, the required sample size is indirectly influenced by both the cardinality of the groups and the size of the input data via Δ.

4.2.1 Validating top-k skewness and selecting FA groups. We validate top-k skewness and select FA groups by calculating confidence intervals for each group in our sample. For sum and count, we apply Hoeffding's inequality, using the formula $\epsilon = (b-a)\frac{1}{2n'_i}(\ln \frac{2}{1-\beta})^{1/2}$, where a and b are the minimum and maximum values, and n'_i is the count of tuples per group. For max and min values, we estimate

Algorithm 3 Computing partial aggregates and child partitions for an input partition per worker thread

```
1  Input: a specific partition to be aggregated or repartitioned at a
2   given worker thread: partition-i, FAgroups,size of partition: Cp
3  Output: partialAggregates-i, childPartitions-i
4  Procedure AggregateAndPartition:
5    // first decide between exact aggregation, logical/physical partitioning
6    if the size of distinct groups in partition-i < CF: // few distinct groups
7      exactAggregation = true
8    else: // check for locality of groups
9      For each segment of size s in the partition-i:
10       calculate ds (number of distinct groups) and cs (cardinality)
11         for segment s
12     Locality of groups, l = Σs ds/cs/t, where t is the number of segments
13     if l < αo:
14       exactAggregation = true // there is sufficient locality
15   if not exactAggregation:
16     If logical partitioning was performed on partition-i in previous pass:
17       partitioning = physical  //logical did not prune previously
18     else:
19       Set Tc to the lowest aggregated count among FAgroups:
20       Estimated number of elements per logical partition, E = Cp/Q
21       if E < Tc:
22         partitioning = logical
23       else:
24         partitioning = physical
25   for each tuple in the partition-i:
26     if exactAggregation or tuple.group in FAgroups:
27       update the partialAggregates-i by aggregating this tuple
28     else:
29       compute the partition hash for this tuple
30       If partition hash is not present in childPartitions:
31         Create a new partition in childPartitions-i
32       Update partitionAggregates corresponding to the partition hash
33       if partitioning == physical:
34         move the tuple to the corresponding partition in childPartitions-i
35         if cache-line size of the corresponding partition is full:
36           store the partition tuples to memory using non-temporal
37           instruction //  (Section 4.1.1)
38   Return partialAggregates-i, childPartitions-i
```

the intervals using the $\beta/2$th and $(100-\beta/2)$th percentiles of the sample values.

Let L_k denote the kth highest lower bound on aggregate values among all groups in the samples, and C_s be the cache space occupied by groups with estimated lower bounds greater than L_k. If $C_s > C_f$, we consider that the skew in distribution is not significant enough to efficiently identify a small set of candidate groups using the cache-friendly approach, falling back to the standard aggregation procedure (lines 17-25 in Algorithm 2).

However, if $C_s \leq C_f$, we proceed with our optimization and additionally include at most h heavy hitters (with size C_h) in FA, such that the combined size of selected groups based on the confidence intervals and heavy hitters is close to C_f (i.e., $C_s + C_h \approx C_f$) (lines 30-33 in Algorithm 2). Using heavy-hitters to fill the remaining space in FA, we reduce the number of tuples per partition in CA. This improves the pruning process and minimizes the need for physical partitioning.

Advantages of sampling over a sketch-based approach. An alternative to sampling is a count-min sketch-based augmented with a priority queue [12]. A count-min sketch maintains a 2D table

Algorithm 4 Merging intermediate results, and pruning and ranking of partitions

```
1  Input: exactAggregates (from previous pass), and <partialAggregates,
2  childPartitions> computed for all partitions in current pass
3  Output: FAaggregates, CAPartitions, topKBound
4  Procedure MergeAndPrune:
5    exactAggregates = Merge partialAggregates across all partitions with
6      matching grouping keys and append them to old exactAggregates
7    childPartitionAggregates = Merge childPartitionAggregates across
8            all partitions with matching childPartition hash
9    For each childPartition:
10     compute upper bound (UB) for groups using childPartitionAggregates
11      (Section 4.1.2 step 3)
12   topKBound = Find kth highest value among exactAggregates
13     and upper bounds of childPartitions
14   top-k groups = Add all groups with exact aggregates value > topkBound
15   if Size(top-k groups) >= K:
16     Return top-k groups, +infinity, {}, {} // we are done
17   partitions = Remove any partition in childPartitions with
18           UB less than topKBounds. // unpruned partitions for next pass
19   If the number of partitions > worker threads:
20     partitions = Rank partitions using partition aggregates
21                          // (Section 4.4)
22   Return top-k groups, topKBound, exactAggregates, partitions
```

of counters with d rows and w columns, and *every tuple in the dataset is hashed* a few times to update this table. Compared to the sampling method, we find that sketch-based technique reduces throughput by 8-10× (Section 7). With sampling, a tuple is aggregated *only* when it's selected in the sample. Furthermore, we can *validate the skew in aggregate values* of the samples without additional passes. While sketch-based approaches offer error guarantees, these are not critical for our setting since the algorithm will later validate and precisely aggregate any groups that will fall into top-k.

4.3 Deciding between Exact-Aggregation and Partitioning

Given an input partition, we first assess whether we can perform an exact aggregation by examining the distinctness of groups and locality of grouping keys. If exact aggregation is ruled out, we choose between logical partitioning and physical partitioning.

4.3.1 Exact Aggregation. Exact aggregation is used in two scenarios (lines 6-14 in Algorithm 3): (1) when the number of distinct groups can fit within the local cache, (2) when there is a high locality among the groups within a partition. We measure locality using a method similar to the one in [25] that considers distinct elements for segments of contiguous tuples within the partition. If c_s represents the cardinality and d_s the number of distinct groups for a segment s, the locality of the groups is calculated as $l = \frac{\Sigma_s \frac{d_s}{c_s}}{t}$, where t is the number of segments in the partition. If $l < \alpha_o$, a predetermined constant, we determine that there is sufficient locality to perform exact aggregation efficiently for partition. The values of s and α_o are tuned once for a specific machine and are determined through benchmarking using synthetic datasets, as described in Appendix A.

4.3.2 Logical vs. physical partitioning. If the estimated number of distinct groups in a partition is significant and there is insufficient locality, we apply partitioning. Logical partitioning is preferred when partition-level statistics can prune child partitions. However,

if it is unlikely that the top-k groups can be identified using FA, logical partitioning may not result in significant pruning. In such cases, physical partitioning is preferred as it allows for early aggregation of groups with skewed frequencies, leading to improved bounds in subsequent passes.

We make this decision using the cardinality of partitions considering that a partition with more tuples is more likely to contain a group in the final results compared to those with fewer tuples. Moreover, a partition with more tuples is expected to have looser bounds than partitions with fewer tuples and hence less likely to be pruned. Let Q be the maximum number of logical partitions that CA can hold, and T_c be the lowest frequency of any group among the current candidate groups. If C_p is the cardinality of the input partition, we estimate the average number of elements per logical partition as C_p/Q (assuming uniform distribution). If $C_p/Q < T_c$, we choose the logical partitioning. Otherwise, we perform physical partitioning, anticipating that one of the child partitions may contain a grouping key that will be present in the final result. Note that if we select logical partitioning but not all child logical partitions are pruned after the pass, we automatically perform physical partitioning on unpruned partitions during the next pass.

4.4 Parallelization

The top-k optimized aggregation framework is designed for high parallelizability, with both FA and CA following a shared-nothing approach. In particular, the first step involves sampling where each core samples a tuple and performs local aggregation independently and in parallel (Algorithm 2 lines 8-14). This step is followed by the synchronization step where partial sample aggregates are merged and the distribution is validated, and the FA groups are identified (Algorithm 2 lines 15-21)). In subsequent passes, each core operates independently either computing exact aggregates for the FA groups or constructing logical or physical partitions in CA (Algorithm 3). After each pass, there is a synchronization where exact aggregates and statistics are merged across cores, and partitions that can be pruned are identified (Algorithm 4). This intermediate synchronization step is parallelizable itself. The pruning of partitions during synchronization reduces the need for physical partitioning and decreases processing time for each core in subsequent passes.

When the number of input or intermediate partitions exceeds the number of worker threads, we rank the unpruned partitions using estimated aggregated values for each grouping key within a partition (lines 19-21 in Algorithm 4). This ranking determines the order of processing by worker threads. If d represents the estimated number of distinct values of groups in a partition, and *psum*, *pcount*, *pmin*, and *pmax* represent the partition-level statistics, we estimate sum and average of a grouping key simply as $psum/d$ and $psum/count$, and max and min as *pmax* and *pmin* respectively.

5 ROLLING TOP-K

In a rolling top-k setting, the top-k aggregate query is issued repeatedly with an increasing window of k values. To avoid running the algorithm from scratch when more results are requested, we introduce the following changes to support top-k in a rolling mode.

During the first pass, we collect samples to validate the applicability of top-k optimization and to identify FA groups (Section 4). We compute confidence intervals for each grouping key in the sample. Both the sample size and confidence intervals are independent of the value of k and hence can be reused during subsequent iterations. Only skew validation and identification of new FA groups are performed for increasing k values. If the number of FA groups exceeds the FA cache size limit in any iteration, we resort to the standard approach of computing exact aggregates for all groups in the dataset. We augment the exact aggregation and partitioning process to reuse computations across queries. Intermediate results, including exact aggregates and partition statistics, are independent of the value of k and are stored in main memory allowing for their reuse. During the processing of subsequent queries, we perform look-ups using partition hash to determine if the intermediate results have already been computed during previous query processing. The third step of our algorithm, which involves merging intermediate results and validating the top-k step, remains unchanged.

6 EXPERIMENTS

Set up. We run the experiments on Windows Server 2022 with two sockets AMD EPYC 7352 24-core processors (48 logical processors), running at base speed of 2.30 GHz. The size of L1 cache is 3 MB, L2 cache is 24 MB, and L3 cache is 256 MB, and the total RAM capacity is 256 GB which is more than the size of datasets.

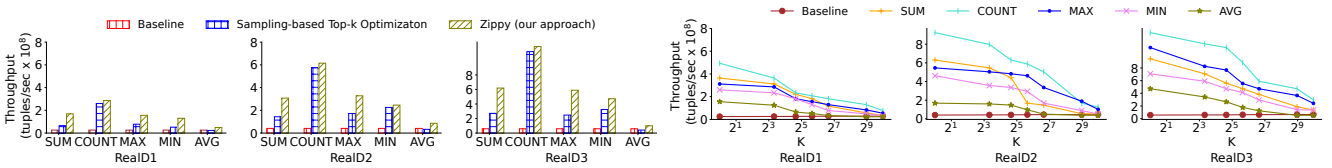
Table 1: Summary of Datasets

Name	Dataset size	# Grouping Columns	# Groups
RealD1	200 M	1	30 M
RealD2	300 M	2	37 M
RealD3	400 M	3	55 M
Synthetic Distributions [16]	200 M	1	30 M

Datasets and queries. We evaluate the standard aggregation functions: SUM, COUNT, MAX, MIN, AVG on three different <group-by, aggregate> queries over high cardinality large customer dataset of Power BI (Table 2). We also consider six synthetic distributions [16] with varying skew in Section 6.5. Prior research on cache-conscious aggregation algorithms focuses only on synthetic datasets with varying degrees of skew in the distribution of *grouping keys*. Since the top-k aggregates depend on distributions of values, therefore we additionally consider a variation of synthetic distributions with a skew in aggregation column values. Unless specified, we set the default value of k to 50, s to 100k, α and β to .95, Δ to .0001 and α_0 to .20. We individually test the sensitivity of Zippy to changes in these parameter values in Section 6.4.

Baseline and evaluation metric. We use throughput (tuples processed/sec) as the evaluation metric. In our targeted scenario, where the number of unique groups exceeds the cache size, the multi-pass partitioning-based aggregation [25] (referred as baseline in figures) outperforms other aggregation approaches evaluated in Section 3. The few-pass approaches are efficient until a certain number of groups (approximately 2 million groups) but are penalized by a high number of cache misses beyond this limit. We augment [25] with a priority queue to select top-k groups.

We also consider an additional baseline referred as *sampling-based top-k optimization*. In this approach, we pick C/2 groups (the default size of FA) that are heavy hitters in the sample (of the same size as selected by Zippy for analysis in the first step of the algorithm). We first make a pass on the data to fully aggregate the selected heavy hitters. We then perform multi-pass aggregation [25] and prune partitions using a bounding and pruning technique as used in Zippy. This baseline resembles [27] in terms of how we



(a) Speed-up over real-world distributions with k = 50. **(b) Speed-up over real-world distributions with varying k.**

Figure 7: End to end evaluation of Zippy over real-data distributions

Table 2: The breakdown of time spent as well as the percentage of groups pruned per pass during top-k aggregate processing.

Aggregate	Sampling	Pass 1			Pass 2			Pass 3+			Top-k Selection (after aggregation)	Throughput (10^8tuples/sec)
	% time	% time	% of top-k	% groups pruned	% time	% of top-k	% groups pruned	% time	% of top-k	% groups pruned	% time	
SUM	10.9%	65.6%	84%	99.7%	23.4%	16%	0.3%	0%	0%	0%	0%	3.59
COUNT	9.2%	71.1%	92%	99.9%	19.7%	8%	0.01%	0%	0%	0%	0%	6.12
MAX	8.5%	49.2%	78%	99.5%	42.1%	22%	0.5%	0%	0%	0%	0%	2.38
MIN	9.3%	53.1%	73%	99.4%	37.5%	27%	.6%	0%	0%	0%	0%	1.95
AVG	8.5%	44.5%	38%	98.6%	25%	40%	1.2%	21.8%	22%	0.1%	0%	.49
Baseline [25]	5.4%	29.2%	0%	0%	23.4%	0%	0%	34.3%	24%	99.9%	7.7%	.42

select the candidate groups, but with further optimizations such as multi-pass aggregation and early-pruning.

6.1 Evaluation over Real Data Distributions

Figure 7a depicts the speed-up of Zippy over different standard aggregation functions across three real-world distributions. Overall, Zippy presents substantial improvement over the baseline (multi-pass aggregation algorithm) for all monotonic aggregation functions and moderate improvements for AVG. Among monotonic aggregations, we see 14.6× improvement for the COUNT, 6.2× for SUM, 5.6× for MAX, and 5.1× for MIN on average for k = 50. Zippy also consistently performs better than the sampling-based top-k optimization. For COUNT, the top-k results are only dependent on the frequency of groups, therefore candidate groups can be captured more effectively during the sampling phase. The sampling-based top-k optimization baseline performs close to Zippy for Zippy since it uses the heavy-hitters as candidates and optimized for COUNT. Among other aggregates, the performance of Zippy is worst for MIN because the bounds used for pruning partitions is much more conservative compared to other aggregation functions (Section 4). Additionally, as the number of groups increases (RealD3 > RealD2 > RealD1), we observe that distributions also become more skewed, leading to larger improvements in performance.

Table 2 presents a breakdown of the time spent in different passes and the progress made in identifying top-k aggregations by Zippy averaged across the three distributions. We see that for monotonic aggregation functions, approximately 80% of top-k aggregates are identified, and around 99% of the groups are filtered in pass 1 averaged across all distributions. The exact aggregation of FA groups helps prune many partitions containing groups from the tail of the distributions. Overall, we find that Zippy takes no more than 2 passes for the queries. In contrast, the baseline approach [25] performs exact aggregation for a significantly more number of the groups, requiring three or more passes for most queries.

We next investigate the impact of varying the value of k, the cardinality of groups, and the size of the dataset:

Varying value of k. We vary the value of k from 1 to 1000 as depicted in Figure 7b. We observe that Zippy leads to orders of magnitude speed-up for all monotonic aggregation functions and significant speed-up for AVG for smaller values of k (e.g., < 20) which are common in BI dashboards and reporting. Furthermore, even for higher values of k (e.g., 100), Zippy leads to a speed-up which is multiple times that of baseline, emphasizing the significance of top-k optimizations.

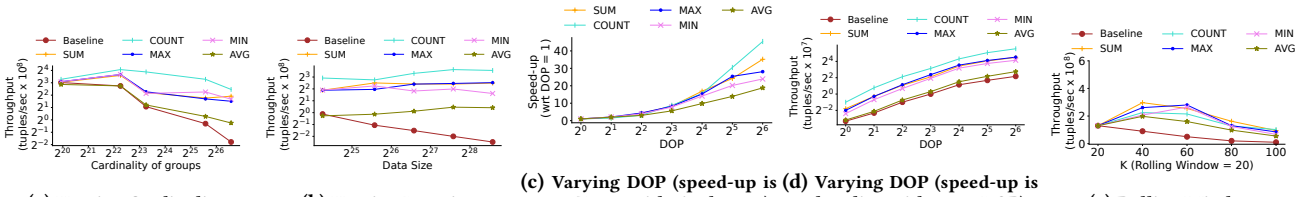
Varying cardinality. We used sampling to scale up and down the cardinality of groups in RealD3 without significantly altering the distribution of values. Specifically, we sorted the groups in decreasing order of their frequency. For down-scaling, we binned them into the desired number of cardinality of groups (depicted on the x-axis), creating a common key for all groups in the same group. For up-scaling, we binned the groups into 1000 bins, and from each bin, we sampled a grouping key and replaced half of its tuples with a new key, until we created the desired cardinality. Overall, as depicted in Figure 8a we found that as the number of groups increases, the speed-up also increases for our approach. This is due to higher number of cache-line transfers and exact aggregation computations incurred in the baseline approaches.

Varying dataset size. We vary the dataset size without changing the cardinality. To downsize, we randomly sample a tuple and remove it when the corresponding grouping-key has more than one tuple in the dataset. When increasing the size of the dataset, we duplicate a tuple and update the original value of each measure column with a new value, where new value = original value ± standard deviation of the aggregate. Figure 8b depicts the results. The increase in the size of the dataset also leads to an increase in the speed-up for our approach. This is due to the increased data movement and aggregation impact on the baseline approach, as also observed when varying cardinality above.

Overhead of memory consumption. We evaluate the overhead in committed memory usage of Zippy w.r.t. to baseline for each of the queries. Although Zippy uses additional data structures for maintaining statistics and bounds, the overhead is minimal (< 10%) compared to the memory already used by the system for operations such as sorting, partitioning, and aggregation which are common in all approaches.

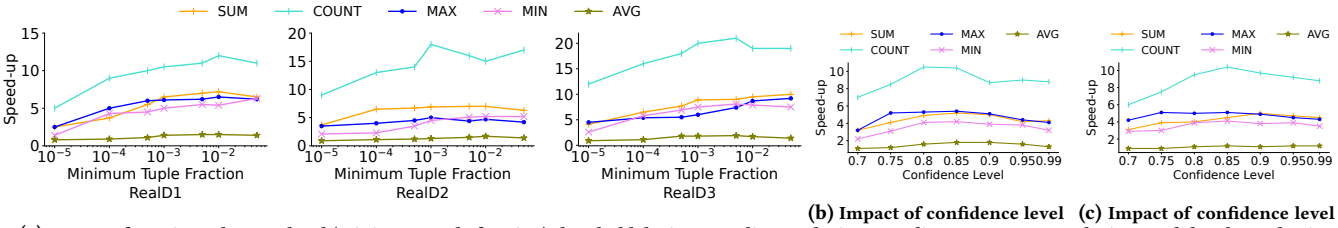
6.2 Rolling K

We evaluate the improvement of rolling-k optimizations compared to a setup (non-rolling) where we run the top-k optimized algorithm from scratch for each increased window of k without reusing



(a) Varying Cardinality (b) Varying Datasize (c) Varying DOP (speed-up is wrt to Zippy with single core) (d) Varying DOP (speed-up is wrt baseline with same DOP) (e) Rolling Window

Figure 8: Evaluating Zippy over varying cardinality, datasize, DOP, and rolling window queries



(a) Impact of varying tolerance level (minimum tuple fraction) threshold during sampling (b) Impact of confidence level during sampling (c) Impact of confidence level during candidate key selection

Figure 9: Sensitivity of Zippy to parameters used for sampling and validation

intermediate computations from prior iterations. Figure 8e shows the results where we vary k from 20 to 100 while using a rolling window size of 20. In the first iteration, we find the top-20 results, then the next 20 in the next iteration, and so on until we find the top-100 results.

As depicted in Figure 8e, as the number of rolling window increases, the speed-up compared to non-rolling optimization increases at a faster rate, peaking at an average speed-up of 7× across aggregates in the last window computation. This speed-up is due to the reuse of intermediate computations (confidence intervals of groups in sample, exact aggregates of some of the candidate groups, statistics on some of the partitions) from previous iterations. The sampling overhead is only incurred during the first iteration. Furthermore, in the optimized version, the exact aggregates from previous steps improve the pruning of partitions. In terms of memory overhead, we find that compared to non-rolling optimization, there is little (< 5%) additional memory overhead when we measure over the last rolling window (the overhead is maximum over the last window). The available main memory available is significantly more than the input data size, so the major difference is what resides inside the local cache vs. outside cache.

6.3 Parallelization

Figure 8c illustrates the improvement in speed-up on varying Degree of Parallelism (DOP) wrt to the throughput when using a single DOP as we increase the DOP from 1 to 64. As expected, we see that as the DOP increases, the speed-up improves significantly. Furthermore, the speed-up is higher for monotonic aggregation functions which benefit more due to pruning of smaller-size partitions compared to the AVG. Figure 8d evaluates the speed-up with respect to baseline [25] as we increase the DOP. We find that both the baseline and our approach scale similarly as we increase the number of cores, suggesting that the majority of our improvements come from efficient use of cache and early pruning of partitions.

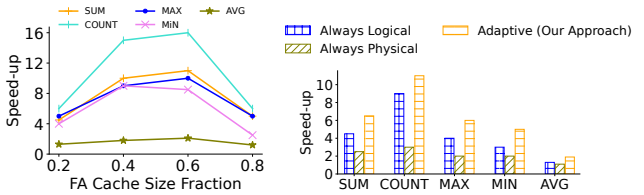
6.4 Sensitivity of Design Decisions

6.4.1 Sampling strategy. As shown in Figure 9a , we varied the minimum tuple proportion (Δ) from .00001 to .1 of the average group cardinality in the dataset. We observe that setting Δ more than .00005 (our default value is .0001) significantly boosts speed; after this range, further gains are marginal. This indicates that a small sample of the overall dataset is enough to identify the top-k grouping aggregates in skewed distributions.

6.4.2 Impact of confidence levels during sampling (α) and FA groups selection (β). Figure 9b and Figure 9c depict the sensitivity of the algorithm when varying confidence levels from .70 to .99 for both sampling as well as identifying FA groups. We observe maximum speed-up when the confidence level is between .80 and .90 after which it flattens followed by degradation in performance. Although we use a more conservative default confidence level of .95, these results indicate that for skewed real distributions even a more relaxed confidence level is sufficient. Furthermore, setting confidence to a high value (e.g., .99) adds to the overhead of additional sampling and FA groups selection with significant improvement in performance.

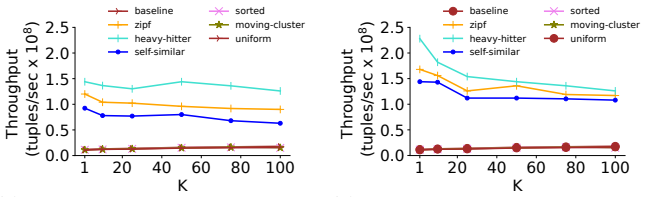
6.4.3 FA vs. CA Cache Sizing. Figure 10a depicts the impact of varying cache size of FA from 0.20 to 0.80 of the total available cache size. We noticed that as the size of the FA increases, the speed up improves; however, after .60, the performance starts degrading. This is because small-sized FA leads to more passes on the data, while large FA leads to excessive data movement between cache and main memory during partitioning. Thus, based on these observations, in our automated approach, we set the FA cache size to 0.50.

6.4.4 Partitioning. We measured the effects of our adaptive partitioning approach, where we switch between logical and physical partitioning based on statistics and the likelihood of the partitions getting pruned after the next pass (Section 4.3). Note that for logical partitioning, if not all child partitions are pruned in pass i, we first create physical partitions in pass i+1 followed by logical partitioning (as described in Section 4.3). As depicted in Figure 10b, using



(a) Impact of size allocated to FA (b) Impact of partitioning strategy

Figure 10: Impact of cache-size and partitioning strategy



(a) Varying degrees of skew in grouping keys with uniformly distributed aggregate column values

(b) Varying degrees of skew in grouping keys with skew in aggregate column values

Figure 11: Evaluating Zippy over synthetic distributions

only one of the two approaches may be sub-optimal. Between logical and physical partitioning, logical partitioning seems to perform better because of the long tail of rare groups. We observe that many partitions can be pruned in one or two passes of the datasets without performing physical partitioning. However, logical partitioning incurs the overhead of an additional scan on the partition as long as there is at least one child partition not pruned. Our adaptive approach can identify such cases early and physically partition them to minimize the number of full partition scans required.

6.4.5 Sampling vs Sketching. Table 3 reports the reduction in throughput in candidate selection as well as the overall throughput when using a sketch-based approach over our sampling approach. We consider two variants of Count Min Sketch: HCMS-1 (with d=2, w= 100) with a low storage footprint but less accuracy and HCMS-2 (with d=10, w= 2000) with a higher storage footprint but high accuracy. We report the reduction in throughput in candidate selection (first value) as well as the overall end-to-end throughput (second value) when using HCMS over the sampling approach. HCMS-1 although has high throughput in candidate selections leads to a reduction in the overall throughput due to missing more candidate groups, therefore requiring more work in later passes. On the contrary, HCMS-2 while more expensive leads to higher overall throughput due to less work in later passes. Nonetheless, both variants are at least 8× less efficient in candidate selection than the sampling-based approach.

Table 3: Sampling vs sketching. Table shows the absolute throughput results for sampling and throughput wrt to sampling results for sketch-based approaches.

Name	Sampling (throughput)		HCMS-1 (2, 100)		HCMS-1 (10, 2000)	
	Cand. Selection	End to End	Cand. Selection	End to End	Cand. Selection	End-to-End
RealD1	$.029 \times 10^8$	$.234 \times 10^8$	.21×	.82×	.13×	.89×
RealD2	$.032 \times 10^8$	$.41 \times 10^8$	.19×	.92×	.09×	.93×
RealD3	$.053 \times 10^8$	$.61 \times 10^8$	.16×	.91×	.05×	.90×

6.5 Impact of Varying Degrees of Skew

Varying grouping key distributions and uniformly distributed values. We consider multiple synthetic distributions over 200 million tuples and approximately 30 million unique groups generated using a process described in [11, 16]. The distributions we consider are: (1) uniform, (2) sorted, (3) heavy hitter, (4) zipf, (5) self-similar, and (6) moving cluster, all capturing variations in the values of group-by attributes. The values of aggregate columns are uniformly distributed between 0 and 10. For heavy hitter, 10% of the values of accounts for 50% of the tuples, while the other tuples uniformly pick the groups from the rest of the group-by keys. The self-similar distribution uses 80-20 proportion and the zipf distribution uses an exponent of 0.5. For moving clusters, the groups are chosen uniformly from a sliding window of size 1024. Finally, the sequential distribution consists of input records in segments, each consisting of a numerically repeated sequence of group-by values, where a sequence consists of group-by keys from 1 to 30 millions.

Figure 11a depicts the performance of Zippy with respect to the best-performing baseline for each of the above distributions. We observe that distributions exhibiting skew in groups such as zipf, heavy-hitter, and self-similar have significant speed-up in performance, similar to what we observe over the real datasets. For other distributions, the aggregate values of groups are similar, hence the validation fails for these distributions and we fall back to baseline approaches. As such, we incur a small overhead involved with sampling and validation, which is less than 10% of the time taken by the baseline algorithms.

Impact of skewness in values. Since the top-k aggregates depend on aggregate values, we also consider an adapted variations of the above distributions where we generate the values of tuples using a zipf distribution with an exponent of 0.5. Figure 11b depicts the result. On adding skew to the values, the performance over distributions where groups are also skewed improves even further. This is because we can prune other non-candidate partitions much earlier due to high pruning threshold. However, there is only limited improvement in the performance of distributions where there is not much skew in the distribution of groups. For large cardinality datasets, the number of groups with skewed values is fewer and they are less likely to be identified in the sampling phases. Thus, the validation fails and we fall back to the baseline approaches.

7 LIMITATIONS AND EXTENSIONS

Settings where our technique excels vs. where it falls short. As we observed in our empirical evaluation, our technique works better for high-cardinality large datasets with skew in groups and aggregate values. However, for datasets with uniform distributions of aggregate values or those with low-cardinality groups falling in the top-k range, the benefits of our technique may be limited. Nonetheless, as we saw earlier, the initial validation step can identify such cases and fall back to baseline approaches with a small overhead of sampling and analysis. For capturing extremely rare groups which may fall in top-k, we believe the top-k optimization may benefit from doing apriori processing on the dataset. For example, adding indexes on the groups can help perform stratified sampling to have more coverage of rare groups. Similarly, with indexes on measure columns, we may be able to identify tuples with extreme values and add the corresponding groups in the first aggregation pass to process them earlier. We plan to explore these extensions as future work.

Extensibility with other operators. In this work, our focus lies exclusively on top-k and aggregation operations. This approach aligns with prior research on cache-conscious aggregation algorithms [11, 25, 35]. Below we briefly discuss how our technique can interoperate with other relational operators. *Multiple group-by attributes.* In our setting, the groups for multiple group-by attributes can be combined as a single key before feeding them to our technique without any changes.
Aggregate over an expression involving multiple columns. The expression within the aggregate function can be pushed down below the aggregation operation as a scalar expression. This is typically the case during query processing in existing relational engines.
Group-by aggregates with filters. If the filter is selective such that the cardinality or the size of the dataset reduces significantly, we may not want to apply top-k optimizations. This can be decided during query optimization. When there is not a significant amount of filtering, we can augment the first pass of our proposed algorithm, where we collect samples for analysis, to also apply filtering.
*Group-by aggregates over PK-FK Joins.*The top-k optimization can be pushed down below the join on the fact table, and only the top-k groups need to be joined with the dimension table.
Group-by aggregates over non PK-FK Joins. For non PK-FK joins, our technique requires that the join between two tables is performed first and materialized. For settings where the join results are ordered on grouping key, a full-aggregation-based approach that avoids materialization may be more efficient than our approach.
Sum over expressions with negative values. The sum over expressions with negative values is non-monotonic. An workaround is ignore the negative values when computing the partition-level statistics leading to to looser bounds. However, this approach can helpful when there are very few negative values and the distributions are highly skewed as we observed for average in Section 6.

8 RELATED WORK

Differences with external memory model. Both external and cache-conscious aggregation algorithms focus on reducing data movement in large datasets through partitioning and recursive processing. Yet, cache-conscious environments require novel algorithms due to distinct challenges: smaller local cache (L1 and L2) sizes and eviction units (cache-line sizes), higher computational overheads, and enhanced parallelization possibilities via efficient L3 cache sharing and reduced core-to-core data movement. Consequently, data structures like priority queues, which have large storage needs and branching, are less preferred. Furthermore, specific partitioning methods, such as radix partitioning and software-managed buffering [6, 34] impact the design of algorithms.

Cache-conscious query processing. There has been significant work on cache-conscious query processing for main memory databases [8, 13, 22, 29], including those that focus on sorting [9, 31], aggregation [11, 25, 27, 32, 35], partitioning [28], and joins [4, 5, 7, 33]. In Section 2, we discuss the related work on aggregation algorithms for modern multi-core CPUs. Furthermore, our algorithm functions at a higher level, utilizing optimized partitioning and hashing techniques from prior work for efficient top-k aggregation.

Top-k query processing. There has been limited prior work on cache-conscious top-k aggregate optimization. The usual approach is to first compute the aggregates for all groups and then select the groups with top-k highest aggregates. This is inefficient over high cardinality and large datasets, incurring unnecessary data movement overhead. Shanbag et al. [31] propose optimizations for finding top-k values given a list of values on GPUs. Similarly, there has been many work on top-k values given a list of values (i.e., no aggregation) in disk-based systems and in IR, as well as top-k query processing for different relational operators, the detailed overview of which can be found in [10, 18]. However, the optimizations cannot be easily leveraged to reduce aggregation cost—the main bottleneck in our setting. Li et al. [21] proposed a rank-aware aggregate operator for disk-based systems using a priority queue. However, for datasets with high cardinality, maintaining intermediate aggregate results in a priority queue may result in large cache-line transfers. Additionally, in our set-up, we do not assume we can access tuples of any group via an index.

Sketches, heavy-hitters and approximate query processing. In this work, we leverage light-weight sketches for distinct count [15] and maintain coarse-grained aggregates such as sum, max, min, and count for each partition (i.e., a set of groups). There has been a large body of work on sketches including those for identifying frequent items and heavy hitters such as [19, 23, 24, 30] which could be leveraged within our framework as long as they have minimal storage overhead (i.e., should fit within the local cache of CPUs). Prior work on approximate query processing queries [17, 20] can be used for identifying candidate groups during the first phase of our algorithm. However, many techniques assume that a groups are indexed which may not be always feasible. Nonetheless, when such access indexes are available, we can leverage faster sampling approaches to improve the pruning effectiveness of our algorithm.

9 CONCLUSION

In this work, we have introduced a novel cache-conscious top-k aggregation framework designed to efficiently identify the top-k aggregates in large and high cardinality datasets. Our approach deviates from the traditional method of aggregating all data and then selecting the top-k aggregates, which incurs substantial overhead in terms of cache-line transfers. Instead, we exploit the skewness in data distribution and the small number of desired aggregates to quick top-k groups without performing complete aggregation for all groups. Through extensive evaluation, we have demonstrated that our approach yields significant performance improvements compared to conventional cache-conscious aggregation methods and is robust to adversarial distributions.

APPENDIX A: TUNING s AND α_0

The parameter s (segment size) decides how frequently we need to check for locality within a partition, and α_0 determines the cutoff point after which we switch from exact aggregation to partitioning. To determine the optimal values of s and α_0, we performed benchmarking on synthetic datasets with varying degrees of skew as described in Section 6.5. We varied s from 100 to 5 million and α_0 from 0.05 to 0.95 and measured the throughput. Our results showed that the maximum throughput of the algorithm is achieved for a wide range of s values between 10,000 and 1 million and α_0 values between 0.10 and 0.25. Based on these results, we set the default values of s and α_0 to $100k$ and 0.20 respectively.

REFERENCES

[1] 2023. Apache Datafusion). https://godatadriven.com/blog/optimizing-topk-queries-in-datafusion/ [Online; accessed 3-May-2023].

[2] 2023. PowerBI (https://powerbi.microsoft.com/en-us/). https://powerbi.microsoft.com/en-us/ [Online; accessed 3-May-2023].

[3] 2023. Tableau Public (www.tableaupublic.com/). www.tableaupublic.com/ [Online; accessed 3-May-2023].

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively parallel sort-merge joins in main memory multi-core database systems. *arXiv preprint arXiv:1207.0145* (2012).

[5] C Balkesen, G Alonso, and J Teubner. 2013. MT Ozsu. *Multicore, main-memory joins: Sort vs. hash revisited. PVLDB* 7, 1 (2013), 85–96.

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

[7] Ronald Barber, Guy Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-efficient hash joins. *Proceedings of the VLDB Endowment* 8, 4 (2014), 353–364.

[8] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[9] Badrish Chandramouli and Jonathan Goldstein. 2014. Patience is a virtue: Revisiting merge and sort on modern processors. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 731–742.

[10] Yannis Chronis, Thanh Do, Goetz Graefe, and Keith Peters. 2020. External merge sort for Top-K queries: Eager input filtering guided by histograms. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2423–2437.

[11] John Cieslewicz and Kenneth A Ross. 2007. Adaptive aggregation on chip multiprocessors. In *Proceedings of the 33rd international conference on Very large data bases*. Citeseer, 339–350.

[12] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[13] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. 1984. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data*. 1–8.

[14] Rui Ding, Qiang Wang, Yingnong Dang, Qiang Fu, Haidong Zhang, and Dongmei Zhang. 2015. Yading: Fast clustering of large-scale time series data. *Proceedings of the VLDB Endowment* 8, 5 (2015), 473–484.

[15] Philippe Flajolet and G Nigel Martin. 1985. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences* 31, 2 (1985), 182–209.

[16] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. 1994. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*. 243–252.

[17] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. 1997. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. 171–182.

[18] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)* 40, 4 (2008), 1–58.

[19] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. 2003. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)* 28, 1 (2003), 51–55.

[20] Albert Kim, Eric Blais, Aditya Parameswaran, Piotr Indyk, Sam Madden, and Ronitt Rubinfeld. 2015. Rapid sampling for visualizations with ordering guarantees. In *Proceedings of the vldb endowment international conference on very large data bases*, Vol. 8. NIH Public Access, 521.

[21] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F Ilyas. 2006. Supporting ad-hoc ranking aggregates. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 61–72.

[22] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering* 14, 4 (2002), 709–730.

[23] Gurmeet Singh Manku and Rajeev Motwani. 2002. Approximate frequency counts over data streams. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 346–357.

[24] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2006. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Transactions on Database Systems (TODS)* 31, 3 (2006), 1095–1133.

[25] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-efficient aggregation: Hashing is sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1123–1136.

[26] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.

[27] Orestis Polychroniou and Kenneth A Ross. 2013. High throughput heavy hitter aggregation for modern SIMD processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*. 1–6.

[28] Orestis Polychroniou and Kenneth A Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 755–766.

[29] Vijayshankar Raman, Gopi K Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. René Mü ller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. *Storm, and Liping Zhang* (2013).

[30] Pratanu Roy, Jens Teubner, and Gustavo Alonso. 2012. Efficient frequent item counting in multi-core hardware. In *Proceedings of the 18th acm sigkdd international conference on knowledge discovery and data mining*. 1451–1459.

[31] Anil Shanbhag, Holger Pirk, and Samuel Madden. 2018. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*. 1557–1570.

[32] Ambuj Shatdal and Jeffrey F Naughton. 1995. Adaptive parallel aggregation algorithms. *Acm Sigmod Record* 24, 2 (1995), 104–114.

[33] Kim C Kaldewey T Lee VW. 2009. Sedlar E Nguyen AD Satish N Chhugani J Di Blas A Dubey P Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. *Proc. VLDB Endow* 2, 2 (2009), 1378.

[34] Jan Wassenberg and Peter Sanders. 2011. Engineering a multi-core radix sort. In *Euro-Par 2011 Parallel Processing: 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29-September 2, 2011, Proceedings, Part II 17*. Springer, 160–169.

[35] Yang Ye, Kenneth A Ross, and Norases Vesdapunt. 2011. Scalable aggregation on multicore processors. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. 1–9.