External Merge Sort for Top-K Queries

Eager input filtering guided by histograms

Yannis Chronis*
University of Wisconsin-Madison
chronis@cs.wisc.edu

Thanh Do
Google Inc
tddo@google.com

Goetz Graefe
Google Inc
goetzg@google.com

Keith Peters
Google Inc
petersk@google.com

ABSTRACT

Business intelligence and web log analysis workloads often use queries with top-k clauses to produce the most relevant results. Values of k range from small to rather large and sometimes the requested output exceeds the capacity of the available main memory. When the requested output fits in the available memory existing top-k algorithms are efficient, as they can eliminate almost all but the top k results before sorting them. When the requested output exceeds the main memory capacity, existing algorithms externally sort the entire input, which can be very expensive. Furthermore, the drastic difference in execution cost when the memory capacity is exceeded results in an unpleasant user experience. Every day, tens of thousands of production top-k queries executed on F1 Query resort to an external sort of the input.

To address these challenges, we introduce a new top-k algorithm that is able to eliminate parts of the input before sorting or writing them to secondary storage, regardless of whether the requested output fits in the available memory. To achieve this, at execution time our algorithm creates a concise model of the input using histograms. The proposed algorithm is implemented as part of F1 Query and is used in production, where significantly accelerates top-k queries with outputs larger than the available memory. We evaluate our algorithm against existing top-k algorithms and show that it reduces I/O traffic and can be up to 11× faster.

*Work done while at Google Inc.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06.
https://doi.org/10.1145/3318464.3389729

CCS CONCEPTS

• **Information systems → Query operators**.

KEYWORDS

Top-K; Query Operators; Out-of-core;

ACM Reference Format:
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 (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389729

1 INTRODUCTION

When analyzing today's huge data volumes, e.g. web logs, users typically want the most relevant results. Business intelligence and web analytics use top-k queries for the final or intermediate results. Users may want only a handful of result rows, but sometimes a large amount of data selected from a huge amount of data. Such queries are common in practice in big internet services companies and the requested output, k, can exceed the capacity of the main memory. For example a data scientist at Facebook might request the 50 million most commented and liked photos out of the 300 million photos posted each day [17]; an engineer at Twitter might want to perform trend analysis on the 10% most important tweets out of the 3.5 billion tweets of the past week[32]; an engineer at Google might calculate the intersection between the 40 million most active search users and the 40 million most active gmail users, the user bases of both services exceeds a billion users [34]; an operations analyst at Amazon might request half of the 100 million US prime members that are most likely to buy a certain product [18].

All methods for optimizing top-k algorithms attempt to eliminate input rows not needed in the output; ideally, before they are sorted. The standard way to evaluate top-k queries uses an in-memory priority queue [4]. The top of the priority queue is the last row to be included in the final output. As

input arrives to the top-k operator, each key is compared to the key at the top of the queue, only the row with the smaller key is retained in the priority queue. The key at the top of the queue serves as a cutoff key. Evaluating top-k queries using this algorithm is very efficient, almost only the top k rows will be sorted. However, this approach can only be used if the requested output, fits in the available memory of a single node.

When the requested output exceeds the capacity of the available memory, the entire input is externally sorted. Existing top-k algorithms are not able calculate a cutoff key and filter the input before sorting or writing to secondary storage [13, 26, 27]. Externally sorting the entire input is an expensive operation and results in unpleasant user experience as the execution of a top-k query exhibits a performance cliff; namely the sudden and drastic change in the execution cost when the output exceeds the memory capacity. An analysis of our production query logs showed that, on an average day, F1 Query [29] executes tens of thousands of top-k queries that resort to an external sort of the entire input. We observe that it is very common for top-k queries to use secondary storage, due to high contention for main memory resources or simply because of large requested outputs. With input and output sizes fixed, the size of the required secondary storage determines overall performance and is the principal metric to optimize.

To address these problems, we introduce a new adaptive algorithm for evaluating top-k queries. Our algorithm is able to eliminate input rows before sorting or writing them to secondary storage, regardless of whether the requested output fits in the available memory. When the output exceeds the memory capacity, the proposed algorithm creates a concise model of the input using histograms while sorted runs are generated. Using the input model, a cutoff key is established and continuously refined, while requiring significantly less space compared to tracking the input in its entirety. Our algorithm performs one pass over the input to generate sorted runs and then merges the runs until the top k rows are produced. Additional passes over the input would incur a high I/O cost given that we target use cases where the input is many times larger than the requested output, which in turn is many times larger than the available memory.

The drop in performance when our algorithm uses secondary storage is proportional to the size of the filtered input and not equal to running an external merge sort of the entire input, thus our algorithm avoids performance cliffs. Our approach follows the recent and promising line of work that learns the distribution of data to accelerate the execution of core database operations while, in some cases, doing so while requiring less space [9, 22, 33]. Our work is a complement to earlier work on run generation [14].

We evaluate our algorithm using inputs of different sizes and key distributions against an optimized external merge sort algorithm [14]. Our algorithm is able to effectively eliminate input rows, reduce the usage of secondary storage and can be up to 11× faster compared to our baseline. We have implemented our algorithm in F1 Query where it is used in production and has significantly sped up top-k queries with outputs larger than the available memory.

The remainder of the paper is organized as follows: Section 2 reviews related prior work. Section 3 introduces and analyzes the proposed algorithm. Section 4 discusses the applicability of the histogram technique on variants of top-k operators and related operations. Section 5 evaluates the performance of our algorithm.

2 RELATED PRIOR WORK

Top-k queries are a well studied and surveyed problem [5, 10, 16, 20, 21]. This section reviews prior work and existing execution techniques for top-k operations. If the input is already sorted as specified by the top-k clause, the problem is trivial. Therefore, the focus here is on cases in which the input is unsorted with respect to the top-k clause.

Most existing research on top-k queries focuses on efficient ways to compute the score based on which records are sorted [10, 16]. Our work is complementary as it improves the performance of a top-k operator once the score of each record is computed.

2.1 Top-k execution strategies

Selection: Optimized top-k algorithms calculate and use a cutoff key to discard tuples that will not be part of the output. The ideal value of this key is the kth and last value of the output. The calculation of the cutoff key seems similar to the selection problem described by Blum et al [2], where an algorithm is proposed to find in linear time the median value or with a simple modification the kth value. This method cannot be efficiently applied in our setting where the available memory is many times smaller than the input data. The out-of-core selection algorithm requires two passes over the input data [25] and performs random reads which are expensive. Our proposed algorithm performs a sequential pass over the input during run generation.

Min/max statistics: A possible execution strategy materializes the input before the top-k operator, collects statistics, as is common in column stores with min/max statistics, and uses the statistics to skip parts of the input that can not be part of the output. Eliminating parts of the input accelerates the execution, but the cost of materialization is prohibitive compared to using the algorithm we propose. Our algorithm eliminates parts of the input at a finer granularity (rows vs groups of rows for which we have statistics) without

incurring the cost of materialization of the entire input to secondary storage before the top-k operator. Furthermore, our algorithm can employ replacement selection [20] which makes run-generation a pipelined operation and does not stop consuming the input to sort the contents of the memory.
Resource Provisioning: Modern servers have ample main memory available, and all but the most extreme datasets can probably fit in memory. In practice even if we could restrict the size of the datasets we are able to process to the size of our memory, a query, let alone a query's operator, does not have access to the entire main memory of a server. Each server has multiple CPUs with multiple cores running hundreds or thousands of threads at any time, consequently, each thread is only allocated a small fraction of the total main memory. Avoiding using secondary storage in the context of a top-k operator would require provisioning enough main memory to store the output. Such an execution strategy in practice is impossible without wasting resources. Predicting resource requirements and execution times even in the context of well-defined database operators is far from perfect. Lacking such prediction mechanisms the need for efficient adaptive algorithms is apparent.
Late Materialization: To reduce the memory footprint of a top-k operation and turn an out-of-memory algorithm into an in-memory algorithm, one could choose to retain in-memory only the columns involved in the sort expression, augmented with a row-id. The final top-k result would be materialized by a join. With input and output sizes many times larger than the available memory, the cost of the necessary join lookup and fetch for the materialization depends on the cost of random I/Os [6, 12]. Local NVM and SSD storage could provide efficient random reads; in our environment, however, storage is disaggregated and handled by servers separate from the ones executing the query logic [29]. The cost of an I/O is a network round trip, plus the invocation of the storage service, plus an I/O in a shared and busy disk drive. In this environment, random I/Os are extremely expensive and thus our execution strategy is to retain any information once gained rather than temporarily placing it on secondary storage.
Partitioning: Using random or hash partitioning, we could find the top k elements per input partition and then produce the final result using a serial selection. This approach works best for small values of k that do not overwhelm the serial step. Range partitioning specifically for top-k is an interesting approach and we discuss it further in Section 3.3.

2.2 Query optimization

For complex queries, there are specific optimization techniques e.g., based on integrity constraints [23]. In the present paper, we focus on the cases that query optimization has produced a promising query execution plan that requires a top operation for an unsorted input of unknown size, as is typical for the result of a complex query. In general, query optimization is orthogonal to the improvements of execution algorithms, the focus of this work.

2.3 Top-K with a priority queue

For small values of k, i.e. when the requested row count fits in the available memory, the simplest algorithm for top-k operators over an unsorted input, uses an in-memory priority queue to track the k smallest key values seen so far in a scan of the input [16]. The top entry in the priority queue is the kth-smallest item. This key decides the disposition of further input rows. Input rows with keys above this key value are eliminated immediately; input rows with keys below this key value are inserted into the priority queue after the current top entry has been deleted.

This design and implementation is perfectly suitable for the easiest cases but it is neither scalable nor robust. If individual rows are unexpectedly large due to variable-size fields, or if the memory allocation is unexpectedly small due to concurrent activity, or if rows with key values equal to the kth key value are desired and the number of duplicate rows is unknown, then this algorithm may unexpectedly fail.

Extending this algorithm to work across multiple levels of a memory/storage hierarchy is possible but has severe performance implications. AlphaSort [27], for example, which is optimized for the difference between cache and DRAM performance, benefits from an internal sort algorithm within the CPU cache and a traditional external sort algorithm beyond. A priority queue "paging" in virtual memory or a database buffer pool would incur multiple page faults per heap traversal. As an example of the performance implications, consider the creation of a b-tree. Random insertions into a b-tree are no substitute for an external merge sort, which is precisely why database products implement a "create index" operation by sorting future index entries and then creating the b-tree left-to-right. A page fault per index entry is a terrible write amplification. Similarly, a page fault per key replacement in a priority queue is a terrible performance penalty.

2.4 Top-K with traditional external merge sort

Usually, the second algorithm implemented for top-k operators is external merge sort [14]. The entire input is consumed and written to sorted runs on secondary storage, the final result is produced by scanning and merging all the sorted runs until k records have been produced. We refer to this approach as "traditional external merge sort algorithm" from here on. Query execution usually starts with the in-memory algorithm but switches to this failback algorithm when it runs

out of memory. Many systems rely on their "vanilla" sort, omitting numerous simple optimizations, e.g., limiting the size of each run to the final output size. This algorithm was made obsolete by the algorithm we describe in Section 2.5.

2.5 Top-K with optimized external merge sort

External merge sort can be optimized specifically for top-k operations as described in [14]. The present paper is a continuation and complement of this prior work. The work presented in [14] focuses on cases in which the desired final output is smaller than a sorted run created by in-memory sorting, but still larger than the available memory. For such small desired outputs, the earlier work describes an incrementally sharpening filter. Run generation uses the kth key value in the first run to eliminate subsequent input rows immediately as they arrive, thus limiting the second run to smaller key values, whereupon the kth key value in the second run can eliminate more input keys for the third run, etc. The earlier work uses replacement selection [20] in run generation: new input rows with large keys go to the current run but input rows with small keys are deferred to the next run. When sorting for a top operation, deferment of an input row to the next run may shorten the current run. The shorter the current run is, the more it can sharpen the input filter.

When k, the desired output, is larger than a sorted run (and the available memory) the recommendation from [14] is to merge sorted runs to produce an intermediate run larger than the final output, derive a cutoff key from the merge output, and filter all further input with this cutoff key. It may be useful to force an intermediate merge step quite early, long before an ordinary external merge sort would invoke its first merge step, just for the purpose of establishing a cutoff key. This technique is much better than a full external sort of the entire input. However, it disrupts the continuous data flow in run generation by replacement selection, performs merge steps that are sub-optimal due to a less-than-maximal merge fan-in, and provides a cutoff key for input removal much later than is possible by using the algorithm we introduce.

The present work focuses on the latter case with large requested outputs, avoids intermediate merge steps, yet filters the input more effectively.

2.6 Massively Parallel Hardware

Shanbhag et al [30] show that the massive parallelism of GPUs can accelerate top-k operations. The best performing top-k algorithms for GPUs, Radix-Select and Bitonic Top-k, can substantially speedup execution compared to sorting the entire input or a top-k with a priority queue algorithm (Section 2.3). GPUs can accelerate various database operators but their deployment can be costly, especially for a large scale deployment that supports tens of billions of queries a day on clusters with thousands of nodes. Shanbhag et al [30] consider cases where the requested output fits in the available memory. The presented algorithms are not applicable in our target use case similarly to the top-k with a priority queue algorithm; when the requested output exceeds the memory capacity in order to locate the kth row the input must be externally sorted.

2.7 Pause-and-resume

Some query engines can create a query result page-by-page or one screenfull at a time. The first page is like a top-k query. Each subsequent page uses the same output size, k, plus an offset clause to skip over the previous result rows, contained in the already presented pages. Thus, not only top-k queries require efficient support in a query engine but also combinations of "limit" and "offset" clauses. The algorithm we introduce in Section 3 supports "offset" clauses effectively.

3 TOP-K WITH HISTOGRAMS

This section introduces our algorithm and analyzes its performance. We defer full evaluation of our production implementation to Section 5.

3.1 The Algorithm

3.1.1 Overview. Our algorithm behaves as the in-memory top-k with a priority queue algorithm while the requested output fits in memory (Section 2.3). Similar to existing approaches, when the output exceeds the memory capacity, secondary storage is leveraged to externally sort the input. Our algorithm differs from existing approaches, as it eliminates parts of the input before being sorted and/or written to runs. During execution, while sorted runs are created, our algorithm creates and refines a concise model of the input. A cutoff key is derived from the input model, continuously sharpened and used to filter input rows. Our algorithm is adaptive, it uses two different ways of calculating a cutoff key, depending on whether the requested output fits in the available memory.

Figure 1 shows how our algorithm works at a high level, when the requested output size exceeds the available memory. The input is consumed by the run-generation logic, which creates and writes sorted runs to secondary storage. The run-generation logic uses the cutoff key to eliminate input rows before reaching the in-memory sort but also before writing them to runs. The cutoff filter logic creates a histogram from each sorted run, combines histograms from multiple runs into the input model, calculates a cutoff key as soon as possible, and continuously refines it while writing runs (explained in detail in Section 3.1.2). When all the input

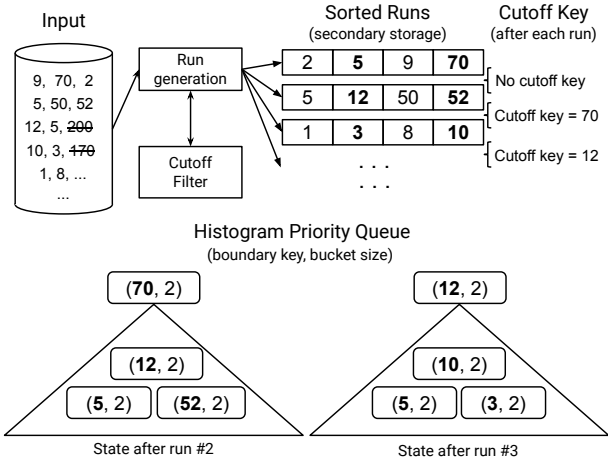

Figure 1: Our algorithm at a high level: The run-generation logic consumes and writes the input to sorted runs in secondary storage. Input rows are eliminated using the cutoff key. The cutoff filter logic maintains a priority queue to track the histogram buckets, calculates and refines the cutoff key as sorted runs are created. Each input row is represented by its sort key.

is consumed, the final result is produced by merging the sorted runs until k records have been produced.

The histograms, and consequently the input model, depend on the input data, the top-k clause and the memory capacity and are specific to each top-k query. While the requested output fits in the available memory, it is stored in a priority queue and the run-generation logic is not activated.

3.1.2 Cutoff Key Calculation. The cutoff filter logic maintains a priority queue that stores the histograms created from each run, which constitute the model of the input. A histogram, here, is a collection of buckets. As runs are written to secondary storage histogram buckets are pushed to a priority queue. Each bucket is inserted to the priority queue and thus combined with the buckets from different runs. Each histogram bucket is defined by its maximum (boundary) key and by the number of rows it represents (bucket size), the size of each bucket is variable. As new runs are created a sizing policy determines the new buckets. If all histogram buckets have the same size, say 100, then the histogram priority queue tracks 100× fewer rows than the requested output.

The priority queue that stores the histogram buckets sorts in the inverse direction compared to the requested output. Therefore, the bucket at the top of the priority queue contains the largest of the bucket boundary keys, which also is the value of the cutoff key. A cutoff key is established, namely it can be used to eliminate input rows, when the sum of the sizes of the histogram buckets in the priority queue is equal or larger to the size of the desired output, $\Sigma(bucket\ size) \geq k$. This condition guarantees that together all the histogram buckets represent at least k rows, in other words the last row to be included in the output has a key less or equal to the cutoff key. If this condition does not hold we can not use the cutoff key to eliminate input rows as we might discard a row that could be part of the output.

The cutoff key is refined when a bucket is popped from the priority queue and the top boundary key is replaced by the next smaller key. We check if we can pop from the priority queue after every insertion. A pop can occur when the sum of the buckets sizes in the priority exceeds k by more than the size of the the bucket at the top of the queue. Essentially, a cutoff key can be established if the histogram bucket at the top of the queue is removed. We insert buckets to the priority queue as runs are written to secondary storage, therefore the cutoff key may be sharpened and used to eliminate parts of the same, currently being written, run.

In Figure 1, the sizing policy creates a histogram bucket using every second key as a boundary key (marked bold), the size of each bucket is 2, which is also saved in the priority queue. Figure 1, presents the first three runs written to disk, the value of the cutoff filter immediately after each run is written to disk and the state of the histogram priority queue after runs 2 and 3. Here, k=8, so after the first two runs are written to secondary storage, 4 buckets will be inserted to the priority queue. The sum of the sizes of the 4 buckets is $8 \geqslant k$, therefore, a cutoff key is established. The value of the cutoff key is the boundary key of the histogram bucket at the top of the priority queue. After the 2nd run the cutoff key is 70, all future runs will not contain rows with keys greater than 70. Consequently, input rows with keys 200 and 170 are eliminated. After run 3 the cutoff key is sharpened and can eliminate rows with keys greater that 12.

The cutoff filter logic used in our algorithm can be combined with any run-generation algorithm. This allows us to take advantage of run-generation optimizations introduced in [14] and described in Section 2.5.

3.1.3 Algorithm Pseudocode. Algorithm 1, presents a pseudocode implementation of our top-k algorithm. We present the logic used when the requested output is larger than the available memory.

Procedure *Top-k* implements the main logic for the top operator. Line 2, initializes the cutoff filter logic that provides the cutoff key, initialization entails creating the priority queue that stores the histogram buckets as shown in Figure 1. Lines 3-5 consume and sort the input. For each input row, the top-k operator uses the cutoff filter logic to decide if it can be eliminated (line: 4), if not the row is passed on to the run-generation logic (line: 5). To decide if a row can be eliminated function *eliminate* compares its value to

Algorithm 1

Input: Input, k, SortInfo: sorting columns and direction
Output: Top k rows of *Input*

```
1:  procedure TOP-K(Input, k, SortInfo)
2:      cutoffFilter ← initFilter(k, SortInfo)
3:      for row in Input do
4:          if !cutoffFilter.eliminate(row) then
5:              SortRow(row, cutoffFilter)
6:      return Merge sorted runs until k rows are produced

7:  procedure SORTROW(row, cutoffFilter)
8:      if Available memory is full then
9:          rowsToSpill ← row(s) to write to sorted runs
10:         for r in rowsToSpill do
11:             if !cutoffFilter.eliminate(r) then
12:                 spillToSecondaryStorage(r)
13:                 cutoffFilter.rowSpilled(r)
14:     Add row to the operator's memory
```

the cutoff key according to the sorting order. If a cutoff key is not established yet, then no input rows are eliminated.

The input rows that were not eliminated at line 4, are passed on to the *SortRow* function (line: 7), which implements the run-generation logic. The input rows are sorted and written to runs according to the run-generation logic. In Algorithm 1, we don't assume any specific run-generation algorithm. The production implementation of our algorithm, used in the evaluation (Section 5), uses replacement selection and other relevant optimizations described in Section 2.5. Replacement selection does not require stopping the consumption of the input to sort the contents of the memory and create sorted runs, like using quicksort does.

Before a row is added to the top operator's memory, the run-generation logic checks if there is enough available memory (line: 8). If there is no memory available, one or more rows currently stored in memory are sorted and written to secondary storage. Before each row is written to secondary storage, it is again compared to the cutoff key (line: 11). The cutoff key may have been sharpened after the row to be spilled was admitted to the run-generation logic. If the row(s) are not eliminated they are written to secondary storage (line: 12), and also passed to the cutoff filter logic (line: 13). Function *rowSpilled* manages the histogram priority queue; it creates new histogram buckets based on the sizing policy used and sharpens the cutoff key (described in detail in Section 3.1.2).

3.2 Algorithm Analysis

This section examines the ability of our algorithm to eliminate input rows when the requested output exceeds the memory capacity. Section 3.2.1 presents a detailed execution example. Section 3.2.2 analyzes the performance of the proposed algorithm when the input size, output size, memory capacity and the histogram sizing policy is varied. The analysis here focuses on efficiency and scalability to large inputs. The input for the experiments in this section contains keys with values uniformly distributed in the range [0,1].

3.2.1 A specific example. As a concrete example, assume a top 5,000 query over an unsorted input of 1,000,000 row, the memory capacity is 1,000 rows. Clearly (because 5,000 > 1,000), the request cannot be handled in memory.

Since the key values are floating point values uniformly and randomly distributed between 0 and 1 inclusively, the final result contains key values between 0 and 0.005; all higher values should be eliminated as soon as possible. Table 1 tracks the progress of the run generation and the cutoff filter logic. In Table 1, each row presents the remaining input rows to be consumed and the value off the cutoff key before each run as well as the values of the keys for each run at each 10% quantile (decile) from 10% up to 90%. For simplicity, in this section, to create a run we fill our available memory with input rows, sort and write them to disk. We create a new histogram bucket when the row at each decile from 10% up to 90% is written to secondary storage, the key at each decile is used as the boundary keys of the bucket. The size of each bucket is 100 (10%∗1000) because it represents the number of sorted keys of the run written to secondary storage since the previous histogram bucket was created. After a cutoff key is established, to generate a run with 1000 rows we may need to consume more than 1000 input rows, as rows can be filtered before added to a run. When a run is written to secondary storage it might contains less than 1,000 rows as the cutoff filter may eliminate parts of the run. When rows after a decile are eliminated and not written to disk we leave the corresponding decile cells empty in Table 1.

Runs 1-6 contain key values from 0 to 1 with decile values about 0.1, 0.2, 0.3... 0.9. The information in the histograms gathered from these runs guarantees that there are more than 5,000 keys below key 0.9, because 6 ∗ 900 rows = 5,400 rows > 5,000 rows. We can eliminate rows with keys above 0.9 in run 6, because those rows are not going to be part of the requested top 5,000 final output rows.

After run 6, all key values larger than the cutoff key (0.9) can be eliminated immediately from the remaining input. Thus, finding 1,000 key values for run 7 consumes about 1,111 input rows. Run 7 contains key values from 0.0 to 0.9 and its 9 decile keys are 0.09, 0.18, 0.27…0.81. While run 7 is written the cutoff key is refined and run 7 ends with key value 0.72.

After run 7, the new cutoff key for run 8 is 0.72 because this cutoff guarantees at least 5,000 key values from the first

Run	Remaining Input Rows	Cutoff Key (before each run)	Values of Keys at Quantiles For Each Run					
			10%	20%	...	70%	80%	90%
1	1,000,000	-	0.1	0.2	...	0.7	0.8	0.9
...	...	-	...	...	...	...	...	...
6	995,000	-	0.1	0.2	...	0.7	0.8	0.9
7	994,000	0.9	0.09	0.18	...	0.63	0.72	
8	992,889	0.72	0.072	0.144	...	0.504	0.576	
9	991,501	0.6	0.06	0.12	...	0.42	0.48	
10	989,835	0.504	0.0504	0.1008	...	0.3528	0.4032	
11	987,851	0.45	0.045	0.09	...	0.315	0.36	
12	985,629	0.4	0.04	0.08	...	0.28		
13	983,130	0.315	0.0315	0.063	...	0.2205	0.252	0.2845
14	979,956	0.288	0.0288	0.0576	...	0.2016	0.2304	
15	976,484	0.24	0.024	0.048	...	0.168	0.192	
16	972,318	0.2	0.02	0.04	...	0.14	0.16	0.18
17	967,319	0.18	0.018	0.036	...	0.126	0.144	
18	961,764	0.1512	0.01512	0.03024	...	0.10584	0.12096	
19	955,151	0.126	0.0126	0.0252	...	0.0882	0.1008	
20	947,215	0.10584	0.010584	0.021168	...	0.074088	0.084672	0.095256
21	937,767	0.1	0.01	0.02	...	0.07	0.08	
22	927,768	0.0882	0.00882	0.01764	...	0.06174	0.07056	
23	916,431	0.074088	0.007409	0.014818	...	0.051862	0.05927	
24	902,934	0.063	0.0063	0.0126	...	0.0441	0.0504	
25	887,061	0.054	0.0054	0.0108	...	0.0378	0.0432	
26	868,543	0.048	0.0048	0.0096	...	0.0336	0.0384	
27	847,710	0.04	0.004	0.008	...	0.028	0.032	0.036
28	822,711	0.036	0.0036	0.0072	...	0.0252	0.0288	
29	794,934	0.03024	0.003024	0.006048	...	0.021168	0.024192	
30	761,866	0.02646	0.002646	0.005292	...	0.018522	0.021168	
31	724,074	0.02226	0.002223	0.004445	...	0.015558	0.017781	
32	679,083	0.02	0.002	0.004	...	0.014	0.016	
33	629,084	0.01764	0.001764	0.003528	...	0.012348	0.014112	
34	572,395	0.014818	0.001482	0.002964	...	0.010372	0.011854	
35	504,908	0.0126	0.00126	0.00252	...	0.00882	0.01008	
36	425,543	0.0108	0.00108	0.00216	...	0.00756	0.00864	
37	332,951	0.0096	0.00096	0.00192	...	0.00672	0.00768	
38	228,785	0.008	0.0008	0.0016	...	0.0056	0.0064	0.0072
39	103,786	0.0072	0.000964	0.001927	...			

Table 1: Approximate quantiles and cutoff keys. Empty quantile cells indicate eliminated rows.

7 runs: at least 700 rows each from runs 1-6 plus 800 rows from run 7 have keys less than or equal to 0.72. With a cutoff key of 0.72, 1,000 rows for run 8 require consuming 1,388 input rows. Run 8 contains key values from 0.0 to 0.72 and its 9 decile keys are 0.072, 0.144, 0.216... 0.576, 0.648. Writing run 8 ends immediately after writing the key value equal to or higher than the new cutoff key, i.e., 0.6. After run 8, the new cutoff key is 0.6, because runs 1-8 contain at least 5,000 key values less than 0.6, namely 600 rows in each of runs 1-6, $600 - 700$ rows in run 7, and 800 rows in run 8, with $6 * 600 + 600 + 800 = 5{,}000$ rows. After run 9, the cutoff key is 0.504.

After run 10, the cutoff key is 0.45. Therefore, after consuming about 1% of the input, 55% of the remaining input can be eliminated immediately. Thus, even without further sharpening of the cutoff key, this algorithm spills 2× less than the traditional external merge sort algorithm, i.e., fully sorting the input. Using the early merge step technique (employed

by the optimized external merge sort algorithm), merging 10 initial runs establishes a cutoff key able to eliminate $\frac{1}{2}$ of the remaining input immediately. Note that this merge step may stop after producing 5,000 rows; as the top 5,000 query has no need for more rows. On the other hand, a small histogram for each run containing as few as 9 buckets enables elimination of 10% of the remaining input after only 6 runs, without the merge effort. After 7 runs, the eliminated input increases to more than 20%; and to about 50% of the remaining input after 10 runs, again without the merge effort. Due to continuing incremental improvements in the cutoff key, only 39 runs are required containing less than 35,000 rows. In total, our algorithm will write to secondary storage 12× less input rows compared to the optimized external merge sort (Section 2.5) and 28× fewer rows than the traditional external merge sort algorithm for this example. These calculations assume perfectly uniform random distributions but illustrate the crucial effects clearly.

The advantage of decreasing cutoff keys begins after k input rows have been processed. With additional runs the cutoff key is refined, eventually approaching the largest key value in the final output. Larger histograms permit sharper input filters. For example, with 19 buckets per run, the cutoff key after 6 runs can be 0.85 rather than 0.9 as in the example above, 37 runs are required rather than 39 and the final cutoff key is 0.006024. The total size of the 37 runs is less than 32,000 rows. One extreme case tracks each key value, equivalent to a histogram with 1,000 buckets. This case requires only 35 runs containing less than 30,000 rows. Interestingly, a mere 50 histogram buckets already reduce the number of runs to 35. The opposite extreme case tracks only the median key value of each run, which requires 66 runs containing less than 63,000 rows. Note that this is still 15× less than in the traditional external merge sort algorithm (sorting all 1,000,000 input rows) and 7× less than in the optimized merge sort algorithm proposed in [14] for our problem setting.

3.2.2 Analysis. This section evaluate the ability of our algorithm to compute a cutoff key and reduce the number input rows written to secondary storage. Tables 2-5 present each a series of experiments where one parameter is varied. Each row of the following tables indicates one experiment, the first column shows the varied parameter. The Runs column shows how often the memory contents are sorted and written to secondary storage. The Rows column lists the number of input rows that were sorted and written to runs. The Cutoff column contains the cutoff key after the last run is written, which is also the key that ends the last run. The Ratio column is the quotient between the Cutoff column and the ideal cutoff key (smaller is better). The ideal cutoff key is the last key value in the final output; of course it is not known until the output is produced.

#Buckets	Runs	Rows	Cutoff	Ratio
0	1,000	1,000,000	-	200
1	66	62,781	0.015625	3.13
5	44	39,150	0.007373	1.47
10	39	34,077	0.0063	1.26
20	37	31,568	0.00567	1.13
50	35	30,156	0.00532	1.06
100	35	29,780	0.005162	1.03
1,000	35	29,258	0.005014	1

Table 2: Varying histogram size.

Varying Histogram Size. Table 2 illustrates the effects of sizing policies that create a diffferent number of histogram buckets per run, i.e., the amount of information gathered for each run is varied. This series of experiments repeats that of Table 1 (top 5,000 of 1,000,000 unsorted rows with memory for 1,000 rows). Here, the ideal cutoff key is 5,000 / 1,000,000 = 0.005.

The three extreme sizing policies are: no buckets collected at all, one bucket with the median of the run as a boundary key per run, and 1,000 buckets per run are collected, i.e., each key is retained as a histogram bucket of size 1. A comparison of the first two experiments in Table 2 demonstrates the value of retaining even the least bit of information about each run, i.e.,one bucket per run. More detailed information is useful but decreasingly so. A small histogram per run suffices to obtain most of the possible performance benefits. More specifically, with the absolutely minimal histogram our algorithm spills 16 times less than the traditional external merge sort algorithm (62,781 vs 1,000,000 rows spilled). Creating 100 buckets per run enables 30 times less spilling (29,780 vs 1,000,000 rows spilled).

Table 2 also suggests that adaptable bucket sizes may provide only limited advantage. For example, one might consider using large buckets early within a run and smaller buckets later, with the expectation that the small buckets will enable earlier, even if smaller, decreases of the cutoff key. However, even 10× more buckets give only a limited improvement in run count and total run sizes. For example, using 100 instead of 10 buckets improves the total run count from 34,077 to 29,780 or by less than 15%. Going from 100 to 1,000 buckets provides an even smaller, practically negligible, advantage.

Varying Output Size. Table 3 illustrates the effect of large requested output sizes. This series of experiments repeats that of Table 1 but varies the number of output rows. The number of runs written grows as the output size grows, which is expected as more input needs to be consumed for a sharp input filter to be produced. Nevertheless, our algorithm is able to eliminate substantial parts of the input. The last experiment is run thrice, with 10, 100, and 1,000 histogram

Output	Runs	Rows	Cutoff	Ratio
2,000	20	14,858	0.00245	1.23
5,000	39	34,077	0.0063	1.26
10,000	67	62,072	0.0126	1.26
20,000	113	109,016	0.025	1.25
50,000	222	218,539	0.06048	1.21
	204	200,161	0.050803	1.01
	202	198,436	0.050076	1

Table 3: Varying output size. The last experiment is run thrice, with 10, 100, and 1,000 histogram buckets per run respectively.

buckets per run respectively. Larger histograms help but only moderately so. Going from 10 to 100 histogram buckets reduces by only 10% the total usage of secondary storage.

Varying Input Size. Table 4 illustrates the effect of large inputs. This series of experiments repeats that of Table 1 but varies the number of input rows. Note that the ideal cutoff key decreases because a fixed output size of 5,000 rows is a smaller fraction for larger inputs. The Cutoff column shows the incremental sharpening of the input filter for large inputs. For example, when processing an input of 100,000,000 rows, the state of the algorithm after 1% of the input is indicated by the entries for the experiment with 1,000,000 inputs rows. Interestingly, comparing the experiments with input size 20,000 and 10,000, the 10,000 additional input rows require only 4 additional runs containing about 3,500 additional rows (13 vs 9 runs, 11,840 vs 8,332 rows); comparing the experiments with input size 200,000 and 100,000, the second 100,000 input rows require only 4 additional runs containing 4,000 additional rows; etc. Even the second 50,000,000 input rows, comparing the experiments with input size 100,000,000 and 50,000,000, require only 5 additional runs containing just over 4,000 additional rows (71 vs 66 runs, 61,235 vs 57,182 rows). There could hardly be a better illustration of the effectiveness of histograms in producing an input model that incrementally sharpens the cutoff key.

Varying Input Size - Minimal Histogram. Table 5 repeats the experiments shown in Table 4 but with a minimal histogram for each run, i.e., only one bucket is created from each run which has the median key as a boundary key. This minimal histogram sizing policy leads to twice as many run on average as the input size grows, compared to the results shown in Table 4. While sub-optimal, this configuration is still usable and vastly superior to a standard external merge sort of the entire input. For example, for the largest input size in Table 5, a traditional external sort spills the entire input whereas our algorithm reduce this to $\frac{1}{8}$% of the input rows, i.e., it filters out $99\frac{7}{8}$% of the input.

Input size	Runs	Rows	Cutoff	Ideal	Ratio
6,000	6	5,900	0.9	0.833333	1.08
7,000	7	6,699	0.8	0.714286	1.12
10,000	9	8,332	0.532978	0.5	1.06
20,000	13	11,840	0.288	0.25	1.15
50,000	19	16,690	0.116482	0.1	1.16
100,000	24	20,627	0.06174	0.05	1.23
200,000	28	24,638	0.0315	0.025	1.26
500,000	35	30,008	0.0126	0.01	1.26
1,000,000	39	34,077	0.0063	0.005	1.26
2,000,000	44	38,188	0.003175	0.0025	1.27
5,000,000	50	43,565	0.00126	0.001	1.26
10,000,000	55	47,683	0.000635	0.0005	1.27
20,000,000	60	51,735	0.000318	0.00025	1.27
50,000,000	66	57,182	0.000127	0.0001	1.27
100,000,000	71	61,235	0.000064	0.00005	1.28

Table 4: Varying input size.

Input size	Runs	Rows	Cutoff	Ideal	Ratio
6,000	6	6,000	1	0.833333	1.2
7,000	7	7,000	1	0.714286	1.41
10,000	10	9,500	0.5	0.5	1
20,000	15	14,500	0.5	0.25	2
50,000	25	24,000	0.25	0.1	2.5
100,000	34	32,250	0.125	0.05	2.5
200,000	44	41,125	0.0625	0.025	2.5
500,000	56	53,437	0.03125	0.01	3.13
1,000,000	66	62,781	0.015625	0.005	3.13
2,000,000	76	72,203	0.007812	0.0025	3.13
5,000,000	90	85,499	0.003425	0.001	3.43
10,000,000	100	94,999	0.001773	0.0005	3.55
20,000,000	110	104,500	0.000903	0.00025	3.61
50,000,000	123	116,209	0.000244	0.0001	2.44
100,000,000	133	125,708	0.000122	0.00005	2.44

Table 5: Varying input size, minimal histograms.

3.3 Summary of the analysis

In summary, even histograms of moderate size allow our algorithm to efficiently eliminate input rows. Compared to early merge steps (Section 2.5) our algorithm establishes a cutoff key faster and sharpens it more effectively without the effort of premature sub-optimal merge steps. Our algorithm is not very effective for input sizes only slightly larger than the desired output size, but its effectiveness increases rapidly with larger inputs, as seen in Table 4.

Our algorithm is vastly superior to the traditional external merge sort algorithm commonly used today. For example,

in Table 4, the Rows column shows the I/O effort required by our algorithm, whereas the Input Size column shows the I/O effort required by the traditional external merge sort algorithm, which will sort and write the entire input to secondary storage. The difference exceeds two orders of magnitude for inputs of moderate size (e.g., 5,000,000 versus 43,565 rows) and it exceeds three orders of magnitude for very large inputs (100,000,000 versus 61,235 rows). In other words, the proposed algorithm is both efficient and scalable to very large inputs and by not externally sorting the entire input when the output becomes larger that the available memory, avoids performance cliffs and provides a pleasant and consistent user experience.

It may be useful to compare our algorithm with the top-k with a priority queue algorithm (Section 2.3). They seem more similar than different. Both use a priority queue in a sort order opposite to the sort order of the top-k clause. Both track a number of key values proportional to the desired output size k. The first difference is that the priority queue in our algorithm tracks key values only, not entire rows; the second difference is that its entries (histogram buckets) represent groups of rows rather than individual rows.

Range partitioning specifically for top-k is an interesting approach and has many similarities to our algorithm. Range partitions and histogram buckets are very similar concepts. As soon as the partitions holding input rows with low values are sufficiently populated, partitions holding input rows with high values can be filtered. Changing the number of partitions has the same effect on filtering as the histogram sizing policy for our algorithm. Effective range partitioning requires foreknowledge of the key value distribution, specifically of approximate quantiles.

4 APPLICATIONS OF THE HISTOGRAM TECHNIQUE

Sections 4.1-4.5 discuss how the histogram technique used in our algorithm can improve variations of the top-k operator and relevant operations. The experimental evaluation of which is out of the scope of this paper, but we believe describing them is worthwhile as they showcase the wide applicability and benefits of the histogram technique.

4.1 Merge optimizations

The histogram logic presented in Section 3 pertains to run generation but can be extended to the merging of sorted runs, in particular when multiple merge steps are required. A merge step ends when the row count desired for the final output is reached or when the value of the latest merged row exceeds the cutoff key. Each merge step can also reduce the cutoff key, which is particularly useful if any original input remains unsorted and requires run generation.

The traditional policy for merging runs chooses the smallest remaining runs, so it reduces the remaining number of runs with the least effort. In a top operation, however, each merge step should choose the runs with the lowest keys, i.e., the runs produced most recently.

Histograms can also speed up run generation and merging in the presence of an offset clause, which is commonly used to support paging. The combined histogram from all runs can determine the highest key value (in an ascending sort) with a rank lower than the offset; this is the key value where the merge logic should start. Searching for this key value in merge input requires searching within runs. If runs are stored in search structures, e.g., a partitioned b-tree or a linear partitioned b-tree, this search is quite efficient.

4.2 Partially sorted inputs

Before comparing the sort orders of the query's "order by" clause and of the input table, the lists of columns ought to be reduced based on functional dependencies [31]. If the definition of the input order and the top-k ordering clause share a prefix then we can perform a top-k operation once for each distinct value of the prefix, also known as segmented execution. The same is true if the first segment satisfies the top-k clause, subsequent segments can be ignored. More generally, the sort proceeds segment by segment and ignores subsequent segments once it has produced k rows. The optimizations we introduce pertain to the last relevant segment but not to the earlier segments which are required in their entirety.

4.3 "Top K" for groups and partitions

Sometimes there is a need for the top-most data items not just globally but within disjoint groups or partitions. An example is finding the 10 million most active customers from each country. The principal difficulty here is additional bookkeeping. Instead of tracking only a single cutoff key, a grouped top operation must track cutoff keys separately for each group. In the example above, if there are customers in 180 countries, each country has its own histogram priority queue, cutoff key, etc. Smaller histograms can reduce the size of the created input models. As some groups might have only a few input rows in each run, the decision about the size of each bucket should be made independently for each group.

4.4 Parallel and distributed algorithms

A top operation can run in parallel by running the original top specification in a separate thread. While retaining many more input rows than required, and more rows than a single thread would retain, it eliminates many of the input rows. If the participating threads share an address space, they may

share a histogram priority queue. Such a group of threads retains basically the same number of input rows as a single thread. Across address spaces, threads may chatter or inform each other of current cutoff keys or even of the contents of their priority queues, thus reducing the number of retained input rows to nearly that of a single thread.

An alternative approach puts the sort and top logic on the consumer side of the data exchange and the filtering on the producer side. The producers ship to the consumers full data packets and the consumers send to the producers flow control packets containing the current cutoff key. This alternative implementation approach promises less development effort but probably also suffers from lower effectiveness than sharing histogram priority queues.

4.5 Approximate solutions

There are at least two forms of approximate top-k queries. First, the row count may be approximate. For example, a "top 100" request may produce 90, 100, or 110 rows, or anything in between. Second, the selection of rows may be approximate. For example, a "top 100" request may produce 100 rows, all of which belong to the true "top 120" rows. For grouped top requests, an approximate query response may include only some or many but not all of the groups. There may also be combinations of these types of approximation.

More research may suggest clever ways to exploit the freedom to approximate the true query result for reduced computation, reduced I/O, or reduced communication in parallel and distributed algorithms. One opportunity is immediately obvious, use approximate bucket sizes. However, even an conservatively estimated final cutoff key may lead to fewer final result rows than requested in the query.

5 EVALUATION

In this section, we present an empirical evaluation of our algorithm and compare the results to the analysis of Section 3.2. We evaluate the performance of our algorithm while we vary the input size, the output size, the distribution of keys and the size of the histograms.

The speedup achieved by our algorithm and the reduction of rows spilled to secondary storage are perfectly correlated. This correlation is natural as the performance of the baseline algorithm is I/O bound and our algorithm reduces the I/O effort.

5.1 Experimental Methodology and Setup

5.1.1 Experimental Methodology. Each of our experiments executes a query that scans an input table, sorts on a single column, applies a limit, k, and projects on all of the columns of the table. For the input table we use the schema of the *Lineitem* table from the TPC-H benchmark [8], we sort on the *L_ORDERKEY* column, the remaining columns serve as a payload. The input tables are unsorted. We sort keys that follow various distributions by using values from the distributions described in Section 5.1.4 for the sort column. The specific query we use is:

```
SELECT L_ORDERKEY,..., L_COMMENT -- full projection
FROM LINEITEM
ORDER BY L_ORDERKEY
LIMIT K;
```

5.1.2 Setup. Our algorithm is implemented as part of F1 Query which we use as the platform for our evaluation. The implementation of our algorithm uses replacement selection to perform run-generation [20] and limits run sizes to k. Run sizes when replacement selection is used are variable, their size depends on the order of the input as well as the input row sizes (which can change when variable sized attributes are used). A best effort is made to decide the target number of histogram buckets collected from each run (default: 50). If the histogram priority queue holds too many buckets (each with a small size), it might exceed its memory allocation (default: 1 MB). In that case, a consolidation step replaces all existing histogram buckets with a single bucket. The boundary key of the new bucket is the boundary key of the bucket previously at the top of the priority queue. The size of the new bucket is equal to the sum of the sizes of the buckets present in the queue at the time of the consolidation. The cost of consolidation is negligible and equal to one insertion to the histogram priority queue.

F1 Query is an internal production system and due to confidentiality restrictions we don't report absolute execution time. Instead, we report the improvement of the execution time and the reduction of the number of rows written to secondary storage compare to our baseline (Section 5.1.3).

The experiments presented in this paper were run on a workstation with one Intel(R) Xeon(R) CPU E5-1650 v3 running at 3.50GHz, 64GB of main memory and a 7200 rpm hard drive. The CPU Governor setting [3] is set to the "Performance" option. In our evaluation, the default memory allocation for a top-k operator is 1 GB, which given our input datasets is sufficient for 7 million rows. The sizes of the input tables we use range from 5 million to 2 billion rows.

5.1.3 Baseline. We compare against the top-k operator previously used by F1 Query. A priority queue is used to evaluate queries when the output fits in memory (Section 2.3). Otherwise, an optimized external merge sort is used, which employs the optimizations introduced in [14] and described in Section 2.5 (i.e. cycle initials runs, early merge steps, limit run size).

5.1.4 Distributions. We evaluate our algorithm by sorting keys that follow various distributions. To experiment with

key values following a uniform distribution we use the values from the *L_ORDERKEY* column of the *Lineitem* table from the TPC-H benchmark [8]. We call this dataset *uniform*. To experiment with keys following non-uniform distributions we use the values from two synthetic datasets *fal* and *lognormal* as in [33]. The *fal* dataset models values that follow a Zipf distribution, which is observed in numerous real world phenomena, including population of cities, and website visits [7, 24, 28]. For the *fal* distribution, we specify how quickly the values grow by using a shape parameter, z. The shape parameter allows us to model distributions ranging from uniform to hyperbolic. *fal* is created using the original generator from [11], in which $fal : N/r^z$, where N = size of dataset, r = rank in the dataset, and z = shape. For our experiments we use multiple shapes [1], which drastically change the rate of growth of the keys. The values in the *lognormal* dataset follow the log-normal distribution. This distribution has been found to model various natural phenomena, such as the time spent by users reading articles online [35] and the size of living tissue [15]. To generate the *lognormal* values, we draw samples from a log-normal distribution, we parameterize the distribution with $\mu = 0,\ \sigma = 2$.

5.2 Varying Output Size

In the experiments shown in Figure 2 we vary the output size requested from the top-k operator. We use datasets with keys that follow the *uniform* distribution and the *fal* distribution with a shape parameter of *1.25*. The size of the input is 2 billion rows and the memory capacity is 1 GB (7 million rows). Our algorithm performs similarly or slightly better compared to the baseline algorithm when the output size is the less than memory capacity or slightly larger ($k < 10$ million), as both algorithms are able to establish cutoff filters.

As the output size increases our algorithm is up to 11× faster than the baseline, as it is able to eliminate input rows and reduce the I/O effort needed while the baseline algorithm externally sorts the entire input. As k becomes a substantial percentage of the input the achieved speedup decreases. This is expected as the input size stays the same, our algorithm consumes more of the input to establish a sharp cutoff key and thus a smaller part of the input will be eliminated. The speedup of our algorithm is a direct translation of the savings in secondary storage usage (and I/O operations), as is shown in Figure 2 bottom plot. The distribution of the sort keys does not affect the performance of our algorithm.

Our algorithm provides a significant improvement for our target setup where the memory capacity is many times smaller than the size of the output which is many times smaller than the input size. Importantly, even when the output fits in the available memory and the existing top-k operator is effective, our algorithm is equally effective and thus an

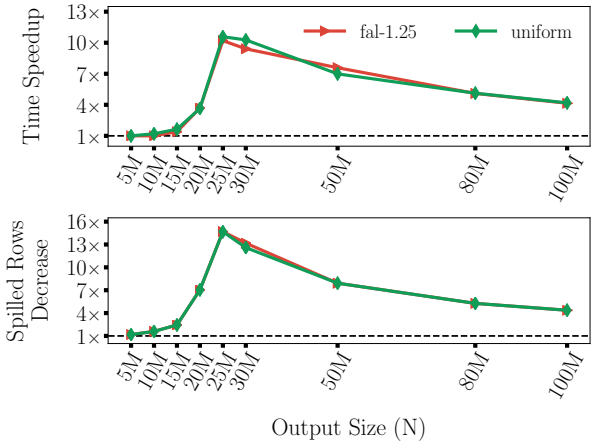

Figure 2: Improvement achieved by our algorithm when the output size is varied. Top: Speedup of execution time. Bottom: Spilled rows reduction.

a-priori choice of algorithm is not required. Our algorithm automatically switches between establishing a cutoff filter the same way as the baseline algorithm while the output fits in memory and using a concise input model created by collecting histograms otherwise.

Top in PostgreSQL. The advantage of our algorithm comes from its ability to filter input rows, when the output size is larger than the available memory. In this case, F1 Query used to perform an optimized external merge sort of the input. PostgreSQL [1] uses a traditional external merge sort algorithm. Existing database systems use one of these two approaches. The top-k operator of PostgreSQL creates sorted runs using quicksort and writes them to disk, when the entire input is sorted the runs are merged. We ran a set of experiments in PostgreSQL version 10, where we gradually increased the requested output size. We observed an order of magnitude increase in execution time when the use of secondary storage is required. This behavior is similar to the baseline top-k operator and therefore by using our proposed algorithm we expect a significant performance improvement in PostgreSQL and systems that perform an external sort of the input.

5.3 Varying Input Size

In the analysis presented in Table 4 we observed the ability of our algorithm to sharpen the cutoff key resulting in producing only a few additional runs when doubling the input size (i.e. increasing the input size from 1,000,000 rows to 2,000,000 rows only 5 extra runs where produced for the same k). Therefore, doubling the input size should increase

[1] https://www.postgresql.org/

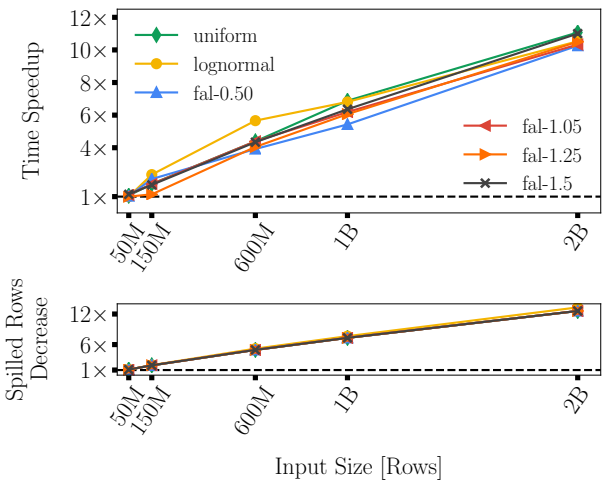

Figure 3: Improvement achieved by our algorithm when the input size is varied for sort keys following multiple distributions. Top: Speedup of execution time. Bottom: Spilled rows reduction.

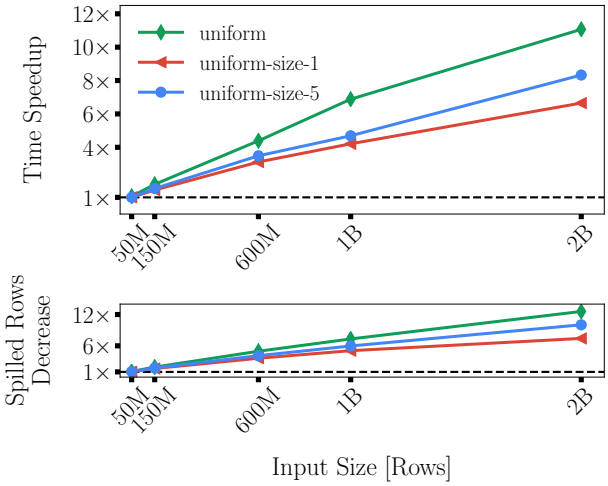

Figure 4: Improvement achieved by our algorithm when the input size is varied. The size-1 and size-5 lines present the improvement when from each run we collect histograms with 1 bucket and 5 buckets respectively. Top: Speedup of execution time. Bottom: Spilled rows reduction.

the speedup achieved by the our algorithm as the baseline top-k algorithm will sort and spill twice as many rows.

Figure 3 presents the improvement achieved by our algorithm when the input size is varied from 50 million to 2 billion rows and k is 30 million. The keys being sorted follow various distributions (*uniform*, *lognormal* and *fal* with shape parameters of *0.5, 1.05, 1.25, 1.5*). The memory capacity is 1 GB (7 million rows). Our algorithm achieves a speedup up to 11× and decreases the number of rows spilled to secondary storage up to 13×. The results in Figure 3 validate our expectations, the benefit for input sizes slightly larger than the output size is small, 1.1× faster when the input contains 50 million rows and 2× for an input of 150 million rows, but it rapidly increases as the input size increases. The behavior of our algorithm is not affected by the distribution of the sort keys, both the execution time speedup and spill reduction are almost identical for uniform and non-uniform distributions.

5.4 Varying Histogram Size

In Figure 4, we replicated the same experiment 3 times, each time we collect a different number of histogram buckets from each sorted run (the input size is varied, k=30 million rows, memory capacity is 1GB). The line named *uniform* creates 50 buckets per run (default configuration), lines *uniform-size-1* and *uniform-size-5* create 1 and 5 bucket(s) per run respectively. Even a minimal histogram of size 1, results in a significant speedup of up to 6.6×.

In Figure 5 we vary the size of the histogram used by our algorithm, input size is 2 billion rows, k is 30 million, and we use the *uniform* distribution. A histogram of size 0 does not eliminate any input rows. The benefit of increasing the histogram size diminishes after a certain point, increasing the histogram size from 50 to 100 increases the speedup by less than 0.1×. These results verify the analysis presented in Table 2.

5.5 Overhead of the cutoff filter

The benefits of filtering are obvious when a substantial part of the input is discarded before going through the run-generation logic and/or spilled to secondary storage. The effectiveness of the filtering depends on the ability of the input model to establish and sharpen the cutoff key and it is a combination of the input data, k and memory budget. It is crucial to ensure that when the filter is not effective the cost of maintaining and updating the priority queue that stores the histogram buckets and the cost of comparisons between the input records and the cutoff key do not hurt the query performance.

To measure this overhead we created an artificial adversarial input, which continuously sharpens the filter but no input rows are eliminated. We observed a 3% overhead compared to the same top-k operator without the cutoff key logic. Thus, discarding even a small fraction of the input is enough to offset the overhead of maintaining and refining the cutoff key.

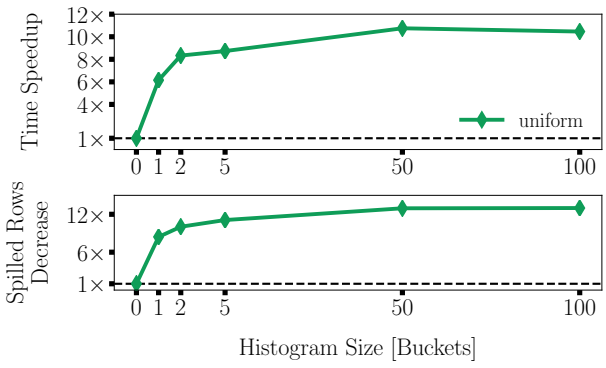

Figure 5: Improvement achieved by our algorithm when the histogram size is varied. Top: Speedup of execution time. Bottom: Spilled rows reduction.

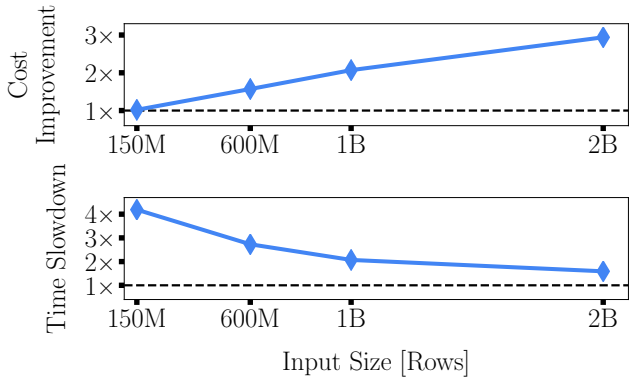

Figure 6: Cost improvement and execution time comparison between our algorithm and an in-memory top-k algorithm for various input sizes, k=30 million

5.6 Cost of resource utilization

In this section, we are interested in the cost of executing a top-k operation when the requested output is large but can still fit in the memory of a single node. We define cost similarly to a pay-as-you-go (i.e. cloud) environment, namely the cost is calculated as *size of resource * time used*. The resource we are interested in is main memory. We compare the cost of executing our algorithm against the cost of an in-memory top-k with a priority queue operator (Section 2.3).

Figure 6 presents the cost improvement and the difference in execution time between our algorithm and the in-memory top-k algorithm. We vary the input size and keep k fixed to 30 million. Our algorithm has a memory budget of 1GB (7 million rows), the in-memory algorithm is allocated sufficient main memory for the entire output. Our algorithm is cheaper to run for all but the smallest input sizes. The cost benefit increases as input sizes increase, our algorithm can be up to 3× cheaper to run for the experiments presented in Figure 6. The in-memory implementation can be up to 4× faster, which is expected as it does not use secondary storage. The performance difference diminishes as the input size increases. For an input of 2 billion rows the in-memory top-k operator is only 1.59× faster but 3× more expensive. Larger requested outputs result in bigger improvements.

The trade-off between memory usage and latency should be tuned based on specific workload and operational needs. Effectively, this result shows that it is cheaper to run more top operations concurrently, where each one is allocated a small part of the available memory rather than running, fewer concurrent top-k operations, where each one is allocated sufficient memory to avoid using secondary storage. Memory is a space-shared resource [19], making large allocations to one operator costly.

6 CONCLUSION

Business intelligence and web log analysis workloads use top-k queries to produce the most relevant results. Top-k queries may produce a small amount of data but sometimes a large amount of data selected from a huge amount of data. Existing top-k algorithms are efficient when the requested output fits in memory, as they can eliminate input rows before sorting them, but resort to an external sort of the entire input otherwise. Externally sorting the input results to far from ideal performance and user experience. In our workloads, every day tens of thousands of top-k queries resort to an external sort of the input, due to large requested outputs or high contention for main memory resources.

In this paper, we introduced a new adaptive algorithm that is able to filter input rows, regardless of whether the requested output fits in the available memory, reduce the usage of secondary storage and accelerate the execution of top-k queries. Our algorithm is scalable to large inputs and outputs and is used in production as a part of F1 Query where it has significantly sped up top-k queries with large outputs.

ACKNOWLEDGMENTS

We would like to thank Stratis Viglas, Herald Kllapi and Jeff Naughton for their valuable comments on drafts of this paper.

REFERENCES

[1] Spyros Blanas, Yinan Li, and Jignesh M Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 37–48.

[2] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. 1973. Time bounds for selection. *J. Comput. Syst. Sci.* 7, 4 (1973), 448–461.

[3] Dominik Brodowski and N Golde. 2016. Linux CPUFreq governors. *http://www.mjmwired.net/kernel/Documentation/cpufreq/governors.txt*

(2016).
[4] Michael J. Carey and Donald Kossmann. 1997. Processing top n and bottom n queries. *IEEE Data Eng. Bull.* 20, 3 (1997), 12–19.
[5] Michael J Carey and Donald Kossmann. 1998. Reducing the braking distance of an SQL query engine. In *VLDB*, Vol. 98. 24–27.
[6] Josephine Cheng, Don Haderle, Richard Hedges, Balakrishna R Iyer, Ted Messinger, C Mohan, and Yun Wang. 1991. An efficient hybrid join algorithm: A DB2 prototype. In *[1991] Proceedings. Seventh International Conference on Data Engineering*. IEEE, 171–180.
[7] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. 2009. Power-law distributions in empirical data. *SIAM review* 51, 4 (2009), 661–703.
[8] Transaction Processing Performance Council. 2008. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html* 21 (2008), 592–603.
[9] Jialin Ding, Umar Farooq Minhas, Hantian Zhang, Yinan Li, Chi Wang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and David Lomet. 2019. ALEX: an updatable adaptive learned index. *arXiv preprint arXiv:1905.08898* (2019).
[10] Ronald Fagin. 2016. Optimal Score Aggregation Algorithms. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 55–55.
[11] Christos Faloutsos and HV Jagadish. 1992. On B-tree indices for skewed distributions. (1992).
[12] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25, 2 (1993), 73–169.
[13] Goetz Graefe. 2006. Implementing sorting in database systems. *ACM Computing Surveys (CSUR)* 38, 3 (2006), 10.
[14] Goetz Graefe. 2008. A general and efficient algorithm for "top" queries. In *Data Engineering Workshop, 2008. ICDEW 2008. IEEE 24th International Conference on*. IEEE, 548–555.
[15] Julian Huxley, Richard E Strauss, and Frederick B Churchill. 1932. Problems of relative growth. (1932).
[16] 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), 11.
[17] Business Insider. 2019. Facebook Photos Statistics. Retrieved 02/16/2019 from https://www.businessinsider.com/facebook-350-million-photos-each-day-2013-9
[18] Business Insider. 2020. Amazon Prime Users Statistics. Retrieved 04/09/2020 from https://www.businessinsider.com/amazon-more-than-100-million-prime-members-us-survey-2019-1
[19] Herald Kllapi, Eva Sitaridi, Manolis M Tsangaris, and Yannis Ioannidis. 2011. Schedule optimization for data processing flows on the cloud. In *Proceedings of the 2011 International Conference on Management of Data*. ACM, 289–300.
[20] Donald Ervin Knuth. 1973. *The art of computer programming: sorting and searching*. Vol. 3. Pearson Education.
[21] D Kossmann and M Carey. 1997. On saying" enough already!". In *SQL, inProc. of the 1997 ACM-SIGMOD Conference on Management of Data', Tucson, Arizona.*
[22] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.
[23] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F Ilyas, and Sumin Song. 2005. RankSQL: query algebra and optimization for relational top-k queries. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 131–142.
[24] Wentian Li. 2002. Zipf's Law everywhere. *Glottometrics* 5 (2002), 14–21.
[25] Tian Mi and Sanguthevar Rajasekaran. 2013. A two-pass exact algorithm for selection on Parallel Disk Systems. In *2013 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 000612–000617.
[26] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. 1994. AlphaSort: A RISC machine sort. In *ACM SIGMOD Record*, Vol. 23. ACM, 233–242.
[27] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. 1995. Alphasort: A cache-sensitive parallel external sort. *The VLDB Journal* 4, 4 (1995), 603–627.
[28] David MW Powers. 1998. Applications and explanations of Zipf's law. In *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*. Association for Computational Linguistics, 151–160.
[29] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shiv Venkataraman. 2018. F1 Query: Declarative Querying at Scale. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1835–1848. https://doi.org/10.14778/3229863.3229871
[30] 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 (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1557–1570. https://doi.org/10.1145/3183713.3183735
[31] David Simmen, Eugene Shekita, and Timothy Malkemus. 1996. Fundamental Techniques for Order Optimization. *SIGMOD Rec.* 25, 2 (June 1996), 57–67. https://doi.org/10.1145/235968.233320
[32] Internet Live Stats. 2020. Twitter statistics. Retrieved 04/09/2020 from https://www.internetlivestats.com/twitter-statistics/
[33] Peter Van Sandt, Yannis Chronis, and Jignesh M Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 36–53.
[34] The Verge. 2019. Google's billion user services. Retrieved 04/09/2020 from https://www.theverge.com/2019/7/24/20708328/google-photos-users-gallery-go-1-billion
[35] Peifeng Yin, Ping Luo, Wang-Chien Lee, and Min Wang. 2013. Silence is also evidence: interpreting dwell time for recommendation from psychological perspective. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 989–997.