A Quick and Exact Method for Distributed Quantile Computation

Ivan Cao
Computer and Information Science
University of Mississippi
Oxford, MS, USA
icao@go.olemiss.edu

Jaromir J. Saloni
Computer and Information Science
University of Mississippi
Oxford, MS, USA
jsaloni@go.olemiss.edu

David A. G. Harrison*
Computer and Information Science
University of Mississippi
Oxford, MS, USA
daharri6@olemiss.edu

***Abstract*—Quantile computation is a core primitive in large-scale data analytics. In Spark, practitioners typically rely on the Greenwald–Khanna (GK) Sketch, an approximate method. When exact quantiles are required, the default option is an expensive global sort. We present *GK Select*, an exact Spark algorithm that avoids full-data shuffles and completes in a constant number of actions. GK Select leverages GK Sketch to identify a near-target pivot, extracts all values within the error bound around this pivot in each partition in linear time, and then tree-reduces the resulting candidate sets. We show analytically that GK Select matches the executor-side time complexity of GK Sketch while returning the exact quantile. Empirically, GK Select achieves sketch-level latency and outperforms Spark's full sort by $\approx 10.5\times$ on 10^9 values across 120 partitions on a 30-core AWS EMR cluster.**

This is the extended version of the paper published in the 2025 IEEE International Conference on Big Data (BigData),

I. Introduction

Computing exact quantiles in large-scale distributed datasets represents a challenge in modern data processing systems. Medians, percentiles, and other order statistics appear frequently in applications ranging from financial risk analysis to real-time monitoring. In large-scale distributed systems such as Spark, quantiles are often computed approximately to reduce overhead. While approximate answers are sufficient in many contexts, regulatory reporting, fairness audits, and scientific analyses often require reproducibility or correctness guarantees that only exact quantiles can provide. This paper proposes an exact quantile method that improves upon Spark's default full-sort approach.

The quantile computation problem has been extensively studied since Hoare's seminal work on the "Find" algorithm in 1961 [1], which introduced the concept of selection without full sorting. Subsequent theoretical advances, including Blum et al.'s linear-time median-of-medians algorithm [2] and Floyd-Rivest's sampling-based improvements [3], established strong foundations for sequential quantile computation. However, these classical approaches face significant scalability challenges when applied to distributed computing environments where data is partitioned across multiple nodes and communication costs dominate performance considerations.

*Corresponding author.

Current distributed computing frameworks, particularly Apache Spark, offer two primary approaches for quantile computation, each with limitations. Exact quantile methods typically require a complete sort operation, resulting in computational complexity and expensive shuffle operations that transfer data across the entire cluster. While this approach guarantees correctness, it becomes expensive for large datasets. Alternatively, approximate methods such as the Greenwald-Khanna sketch [4] provide computational efficiency but sacrifice exactness.

The gap between these approaches creates a need for algorithms that can deliver exact results while maintaining computational efficiency comparable to approximate methods. In distributed systems like Spark, where data locality and communication minimization are critical, traditional sequential algorithms often fail to leverage the parallel processing capabilities effectively, resulting in suboptimal performance that scales poorly with dataset size and cluster resources.

This paper addresses the problem of exact quantile computation in distributed environments by proposing GK Select, a novel algorithm that combines the precision of exact methods with the efficiency characteristics typically associated with approximate approaches. Our method leverages approximate quantiles as intelligent pivot selection for a distributed quickselect variant, reducing the computational overhead compared to full sorting while maintaining exact results.

The remainder of this paper is organized as follows: Section II surveys related work in both exact and approximate quantile computation methods, Section III presents our GK Select algorithm design and theoretical analysis, Section IV evaluates performance through comprehensive experiments, and Section V concludes with implications for distributed data processing.

II. Related Work

A. Exact Selection Methods

1) Sequential Algorithms: The selection problem has its roots in Hoare's partition-based method, where the same in-place partition primitive that powers QuickSort also yields a linear-time, selection-by-partition scheme (FIND/QuickSelect) in expectation [1]. Worst-case linear-time selection was later established by the median-of-medians algorithm of Blum, Floyd, Pratt, Rivest, and Tarjan (BFPRT), which guarantees

$O(n)$ time by deterministically constructing a good pivot at each step [2]. While BFPRT provides strong worst-case bounds, its constant factors often make randomized or sampling-based variants preferable in practice. Floyd and Rivest's SELECT refines the pivot choice via controlled sampling to achieve expected linear time with notably smaller constants than vanilla QuickSelect, both in analysis [5] and in empirical comparisons [3]. These classical results clarify the trade-off between pivot quality and iteration count that exists in nearly all exact quantile/selection procedures.

2) Parallel and Distributed Algorithms: On the theoretical side, parallel selection on shared-memory PRAM models achieves a faster time complexity. Cole introduced an optimally efficient algorithm with linear work and $O(\log \log n)$ time on EREW PRAM [6]. Han later matched the EREW lower bound with an $O(\log n)$-time algorithm using $\frac{n}{\log n}$ processors [7]. Although these results are optimal within their models, they assume unit-cost shared memory and abundant processors, which diverges from modern data-analytics settings (e.g., Spark) where communication, partitioning, and limited parallelism dominate performance.

In a model closer to modern cluster computing—the Coarse-Grained Multicomputer (CGM)—Al-Furaih et al. proposed a "serial-pivot, parallel-count" selection algorithm [8]. Their approach uses random sampling to shrink the search interval to a small candidate set with high probability, and then solves the reduced problem in parallel across processors. All communication is assumed to take place via logarithmic fan-in and fan-out, akin to Spark's `treeReduce` and `TorrentBroadcast`. By contrast, Jeffers's distributed Quickselect employs direct driver–executor communication, but still maps naturally to Spark's map/shuffle/aggregate execution model and serves as a concrete reference design [9].

Currently, the exact quantile computation method implemented in Spark is a full sort based on Parallel Sort by Random Sampling (PSRS). In PSRS, each processor first sorts its local block and selects regular samples, which are gathered and sorted to choose global pivots. The pivots partition the data into buckets that are redistributed among processors, ensuring balanced workloads. A final local sort within each bucket yields the globally sorted sequence. [10]. Where the Spark implementation differs from the approximation is that it performs a full shuffle of all data during the redistribution stage, incurring communication overhead. Because every record must be moved and sorted globally, this method is far more expensive than approximate or selection-based approaches, with the shuffle and global sort dominating runtime costs at scale.

Practical distributed realizations of selection typically follow a "count-and-discard" loop: choose a pivot, have each worker partition locally, aggregate global counts to determine whether to search left or right, discard the opposite side, and iterate. Open-source implementations in message-passing environments (and in cluster settings more broadly) embody this pattern; Jeffers's implementation exemplifies it, exposing an explicit round structure with pivot broadcast, local partition/count, and global decision to "search left" or "search right" [9]. The efficiency of such schemes is highly sensitive to pivot quality: poor pivots increase the number of global rounds (and thus synchronizations), whereas good pivots collapse the search quickly. This sensitivity motivates strategies that systematically improve pivot selection in distributed contexts.

B. Approximate Methods

1) Sketch-based Approaches: Approximate quantile summaries trade exactness for strong space and mergeability properties, which is attractive for streaming and distributed data. The Greenwald–Khanna (GK) summary provides deterministic ϵ-approximate rank guarantees with worst-case space $\frac{1}{\epsilon} log(\epsilon N)$ and can be maintained online [4]. Such summaries can be computed per partition and combined to answer global quantile queries with bounded rank error, a design that underlies widely used "approximate quantile" operators in distributed systems. From a broader viewpoint, data sketching offers a unifying lens for these techniques—compact, composable summaries that support approximate query answering at scale [11]. Comprehensive surveys document the algorithmic landscape and engineering trade-offs among deterministic sketches, randomized summaries, and hybrid methods across streaming and distributed models [12].

2) Sampling Methods: Beyond deterministic sketches, sampling improves pivot or rank estimation with minimal state, reducing communication or iteration counts in distributed settings. In the sequential regime, SELECT's sampling analysis explains why better pivots lead to faster convergence [3], [5]; analogous intuitions carry over to distributed execution where a high-quality pivot can shrink the candidate set precipitously after one round of local partitioning and global counting.

Positioning of This Work: Prior literature largely splits into (i) exact selection with strong asymptotic guarantees in idealized parallel models [6], [7] or practical count-and-discard variants sensitive to pivot choice [9], and (ii) approximate quantile computation via compact sketches with explicit rank-error bounds [4], [11], [12]. Our approach bridges these lines by using an *approximate* summary (GK) to inform an *exact* distributed selection procedure: the sketch supplies a pivot that is already close to the target rank, which in turn reduces the number of global rounds and the volume of shuffled data relative to full sorting or to pivot-agnostic distributed Quickselect. Conceptually, this mirrors the classical role of sampling in SELECT [3], [5], but replaces sampling with a deterministic summary tailored to Apache Spark [4].

III. Implementing Algorithms in Spark

Spark expresses computations as transformations on immutable, partitioned datasets called *Resilient Distributed Datasets (RDDs)*. Each transformation produces a new RDD, which guarantees determinism and fault tolerance but prevents in-place modification of data. As a consequence, algorithms that rely on reordering keys or performing in-place partitioning (e.g., Hoare's Quickpartition) must be adapted: they create a new dataset rather than modifying an existing one. Creating a new dataset at least introduces a copy of the keys upon

which we are computing the kth order statistic. To ensure consistency and avoid recomputation, the results of such reordering operations must be explicitly persisted.

Spark also distinguishes between *transformations* and *actions*. Transformations are lazily evaluated, whereas an action (e.g., `collect`, `count`, `reduce`) triggers execution of the computation lineage. When an action is invoked, Spark materializes all preceding transformations and forces evaluation of the entire dependency chain. Stage boundaries occur at actions and at wide transformations such as shuffles, where data must be redistributed across executors. These boundaries impose synchronization barriers and often dominate execution cost, since no further progress can be made until all data movement has completed. Taken together, immutability, stage boundaries, and communication costs shape the design and performance of selection algorithms in Spark.

Symbol	Description
n	Total number of elements across all partitions.
n_i	Total elements in the ith partition.
π	Pivot used when performing QuickPartition.
P	Number of partitions.
q	Specific quantile level being queried (e.g., 0.5 for median).
Δk	Rank distance between approximate and exact quantile

TABLE I
DEFINITIONS USED IN ALGORITHM DESCRIPTIONS AND ANALYSIS.

IV. RELATED ALGORITHMS

In this section we describe in detail each of the algorithms compared against GK Select.

1) Spark Full Sort: Spark computes exact quantiles by performing a global sort of the entire dataset, implemented as a variant of Parallel Sorting by Regular Sampling (PSRS) [10]. Because Spark's execution model revolves around immutable datasets and stage boundaries, the cost of full sort is dominated not by comparison work but by communication and synchronization.

1) **Sampling.** Each executor independently selects a small, fixed-size sample from its local partition. Because RDDs are immutable, this creates a new sampled dataset.
2) **Collect (first stage boundary)** The driver executes `collect` to gather all samples. This action forces the sampling transformation to materialize and introduces the first stage barrier: all executors must complete their sampling and send results before the driver can proceed.
3) **Splitter selection.** The driver sorts the samples, chooses $P-1$ splitters at target quantiles, and broadcasts them via `TorrentBroadcast` Broadcast incurs communication from driver to executors but does not impose a stage barrier.
4) **Range partitioning (shuffle, second stage boundary).** Executors assign each record to one of the P splitter ranges and emit the record to the corresponding bucket. This triggers a global shuffle in which every executor must send and receive data. Because no executor can make further progress until its shuffle data has arrived, this step forms the second stage boundary.
5) **Local sort.** Each executor sorts its assigned bucket using `UnsafeExternalSorter`. Small partitions are sorted in memory using dual-pivot QuickSort; large partitions spill to disk and are merged via multiway merge.

The two actions—`collect` and the shuffle—dominate elapsed time. Both require all executors to synchronize and transfer potentially large volumes of data. Although the sort within each partition is efficient, the global data movement inherent in PSRS makes Spark's full sort a communication-bound algorithm whose cost grows with both n and the number of partitions P.

2) Al-Furaih Select for Spark: We implemented a Spark-adapted variant of Al-Furaih et al.'s "serial pivot, parallel count" selection algorithm [8]. We call this Al-Furaih Select (AFS). The method proceeds iteratively, shrinking the search interval around the desired order statistic.

1) **Pivot broadcast.** Each round, the driver selects a pivot and broadcasts it via Spark's `TorrentBroadcast`, a logarithmic-depth tree requiring $O(\log P)$ communication but no stage boundary.
2) **Local QuickPartition and Count.** `mapPartitions` invokes `QuickPartition` on each executor on each of its partitions, counting elements $< \pi$, $= \pi$, and $> \pi$ in their local partitions. Because Spark datasets are immutable, this produces a new RDD. Work per partition i is $O(n_i)$. Results are persisted for reuse in subsequent rounds.
3) **Tree reduction of counts and pivot candidates.** Executors send their local counts and two candidate pivots (one below and one above p) into a `treeReduce`. The reduction proceeds in $O(\log P)$ steps, summing counts, and discarding one of the candidate pivots below and one of the candidate pivots above applying reservoir sampling to maintain uniform pivot selection. The `treeReduce` introduces a stage boundary: no executor can proceed until counts from all partitions have been aggregated.
4) **Driver decision and broadcast.** The driver computes Δk, the rank error of π. If π is high, it picks the left candidate; otherwise, the right. The new pivot is then broadcast via `TorrentBroadcast`, starting the next round.
5) **Iteration.** Steps 2–4 are repeated until the pivot lies exactly at rank k. The expected number of rounds is $O(\log n)$ due to geometric shrinkage of the candidate range.

Supplying candidate pivots in step 3 from both sides of the current pivot p allows the driver to choose the next pivot from the correct direction after computing Δk without incurring an additional broadcast and `treeReduce` to select a pivot. This reduces the number of actions per round, and therefore stage boundaries, from two to one.

The only stage boundary occurs at `treeReduce`; broadcasts add latency but not synchronization. As in Spark's full

sort, stage boundaries dominate runtime, while local scans contribute only $O(n/P)$ work per executor.

3) Jeffers Select for Spark: Jeffers Select is identical to AFS except that Step 3 replaces `treeReduce` with `collect`. The driver gathers counts and candidate pivots directly from all executors, sums the counts, and randomly selects the next pivot. Like `treeReduce`, `collect` is an action and thus a stage boundary. Because these messages are small and aggregation is cheap, driver-side computation is often faster than a full `treeReduce`, though for large P the all-to-one communication could dominate.

Symbol	Description
S	The ordered collection of tuples comprising the sketch.
$\|S\|$	The number of tuples in the sketch.
v_i	Sample value in strictly increasing order $v_1 < v_2 < \cdots$
g_i	Gap, lower bound on number of values between v_{i-1} and v_i
Δ_i	Slack, maximum rank of v_i can exceed its minimum rank

TABLE II
ADDITIONAL DEFINITIONS USED IN THE GK SKETCH DESCRIPTION.

4) GK Sketch for Spark: If we have all the data ahead of time and can fully sort it, we could create a summary of the data by pulling out a sample at every fifth percentile. For example, if there are 101 data points, we could select values at indices $0, 5, 10, 15, \ldots, 100$. Such a summary is approximately one fifth the size of the original data. Given a query value x, we could binary search this summary, find the closest sample, and use its index $\times 5$ approximates the rank of x. The returned rank would be guaranteed within 5% of the true rank.

Of course this strategy assumes access to the entire dataset and requires a full sort, which is expensive in Spark if our only objective is approximate quantile computation.

Greenwald and Khanna introduced the GK Sketch in [4]. We refer to their implementation hereafter as the *Classical GK Sketch*. It incrementally constructs a summary while scanning the data stream, maintaining a bounded rank error. This fits well with Spark's execution model, where each partition is processed as a stream via iterators. For example, `mapPartitions` applies a function to a partition by providing sequential access through an iterator, without requiring that the entire partition fit in memory.

Spark's GK sketch as of version 3.5.5, executes a variant of the Classical GK Sketch independently across partitions and then merges sketches back to the driver. The final sketch can then be queried to provide approximate ranks for any key x. The Classical GK Sketch is defined only for streams. Spark adds an operation to merge sketches which allows GK Sketch to be computed independently for each partition before merging in the driver.

The Classical GK Sketch is represented as an ordered collection of *summary tuples*.

$$(v_i, g_i, \Delta_i), \quad 1 \le i \le |S|$$

where the components of this tuple are defined in Table II.

The ordered collection may be maintained internally as a balanced tree or similar data structure allowing $O(\log n)$ inserts.

For every internal tuple (v_i, g_i, Δ_i) in the summary, GK Sketch maintains the following invariant:

$$g_i + \Delta_i \le \lfloor 2\varepsilon n \rfloor \tag{1}$$

Greenwald and Khanna [4] prove that maintaining the Invariant 1 guarantees that the rank of any value returned by the summary deviates from the target rank by at most εn in either direction.

With classical GK Sketch, when a new element x arrives from the stream:

1) Find its correct position in the sorted summary (via binary search on v_i).
2) Insert $(x,\ g,\ \Delta)$ with $g = 1$ and $\Delta = g_{\text{succ}} + \Delta_{\text{succ}} - 1$ (and $\Delta = 0$ at the extremes).
3) After every $\lceil 1/(2\varepsilon) \rceil$ insertions, compress the summary by merging tuples whose combined gap and slack still satisfy the invariant.

The compress guarantees that the sketch size remains bounded by

$$S \le \frac{1}{\varepsilon} \log(\varepsilon n) + 1$$

where n is the number of elements included in the sketch so far. This can be restated in terms of space complexity as

$$S = \Theta\left(\frac{1}{\varepsilon} \log(\varepsilon n)\right) \tag{2}$$

Spark GK Sketch modifies Classical GK Sketch as follows:

1) Rather than inserting each arriving sample into the sketch, Spark GK Sketch appends the sample to the head buffer.
2) When the head buffer reaches maximum size B, the buffer is *flushed* meaning it is sorted in $O(B \log B)$ time and then merged in linear time $O(B + |S|)$ into the sketch.
3) If the resulting sketch is larger than `compressThreshold`, the algorithm then compresses the sketch in $O(|S|)$ time in the same manner as the Classical GK Sketch.

The head buffer is implemented as an array. Appending to an array takes $O(1)$ time. Merging the array into the sketch takes linear time.

By performing an append onto a buffer rather than immediately inserting each sample into the sketch, the sketch can be implemented as a dynamic array of tuples rather than a data structure that supports logarithmic inserts like a balanced tree. Because the inserts only occur periodically, the merge cost becomes amortized, but we show it does have some impact on the time complexity analysis as Spark implements it as of this writing and is discussed in section IV-A1.

The default value for B is `defaultHeadSize = 50000`. The default `compressThreshold = 10000`. Unless the buffer is forcibly flushed before reaching B, flushing will also result in the sketch exceeding the default `compressThreshold` resulting in a compress. We thus

consider the time complexity of flushing to include a call to compress.

By batching samples before compressing, Spark GK Sketch can temporarily exceed the memory bound. The extra tuples are pruned during the subsequent compress after the batch insert to restore the sketch to the $\Theta\left(\frac{1}{\varepsilon}\log(\varepsilon k)\right)$ space bound.

A. Theoretical Analysis

In this section we analyze Spark GK Sketch and modified GK Sketch. In this analysis we assume that the ith partition contains n_i records and that the data has already been distributed equally among the partitions. As such, for all i, $n_i = \frac{n}{P}$ where P is the number of partitions.

1) Executor Time Complexity of Spark GK Sketch: Sorting, inserting, and compressing B records into a sketch of size S takes

$$T_{\text{flush}} = O(B \log B + S) \tag{3}$$

Across n_i records, we would have performed $F = \lceil n_i/B \rceil$ flushes including a final flush to handle records that had not yet triggered a compression.

Let T_{exec} denote the time complexity of the computations performed by the executor across an entire partition.

$$T_{\text{exec}} = \sum_{i=1}^{F} T_{\text{flush},i} = \sum_{i=1}^{F} O(B \log B + S(k_i)) \tag{4}$$

where $S(k_i) \leq \frac{1}{\varepsilon}\log(\varepsilon k) + 1$ and $k_i = B \cdot (i-1)$. Thus Equation (4) becomes

$$T_{\text{exec}} = \sum_{i=1}^{F} O\left(B \log B + \frac{1}{\varepsilon}\log(\varepsilon B \cdot (i-1))\right)$$

$$T_{\text{exec}} = O\left(n_i \log B + \tfrac{F}{\varepsilon}\log(\varepsilon B) + \frac{1}{\varepsilon}\log\left((F-1)!\right)\right) \tag{5}$$

Stirling's formula allows us to simplify Equation (5) to

$$\begin{aligned} T_{\text{exec}} &= O\left(n_i \log B + \tfrac{F}{\varepsilon}\log(\varepsilon B) + \frac{1}{\varepsilon} F \log F\right) \\ &= O\left(n_i \log B + \frac{1}{\varepsilon}\tfrac{n_i}{B}\log(\varepsilon B) + \frac{1}{\varepsilon}\tfrac{n_i}{B}\log\tfrac{n_i}{B}\right) \\ &= O\left(n_i \log B + \frac{1}{\varepsilon}\tfrac{n_i}{B}\log(\varepsilon n_i)\right) \end{aligned} \tag{6}$$

Since we assume $n_i = \frac{n}{P}$,
Equation (6) becomes

$$T_{\text{exec}} = O\left(\tfrac{1}{\varepsilon}\tfrac{n}{P}\log(\varepsilon\tfrac{n}{P})\right)$$

However, this is only valid if $\frac{1}{\varepsilon}\frac{\log(\varepsilon n_i)}{B} \gg \log B$. With Spark's defaults ($\varepsilon \approx 10^{-2}, B = 5 \times 10^4$) the inequality is false for any achievable dataset.

Assuming base 2 logarithms,

$$\tfrac{1}{\varepsilon}\tfrac{\log(\varepsilon n_i)}{B} \gg \log B$$

$$n_i \gg \tfrac{1}{.01}(5 \times 10^4)^{500}$$

which is vastly more than the number of atoms in the universe ($\approx 10^{80}$), so we should not eliminate the first term.

$$T_{\text{exec}} = O\left(\frac{n}{P}\log B + \frac{1}{\varepsilon}\tfrac{n}{P}\tfrac{1}{B}\log(\varepsilon\tfrac{n}{P})\right)$$

This is not the same time complexity as derived for the Classical GK Sketch.

2) Driver Time Complexity of Spark GK Sketch: With Spark's approxQuantile (as of current versions, including Spark 3.x and later), the GK driver then performs a `collect` action to transfer the per-partition sketches from the executors back to the driver. On the driver, it merges them sequentially via a `foldLeft`, i.e., an iterative pairwise merging. This forms the global sketch before querying the final approximate quantiles.

Due to space constraints, we omit the derivations of the driver time complexity, but we point out that `foldLeft` results in asymptotically worse performance than if the driver were to instead locally perform a recursive tree reduce.

With `foldLeft`, merging dominates yielding

$$T_{\text{driver}} = \Theta\left(\frac{P}{\varepsilon}\log(\varepsilon n)\right) \tag{7}$$

3) Modified GK Sketch: To recover the asymptotic behavior of the classical GK sketch, we consider a lightweight modification of Spark's implementation. The goal is to avoid fixed-size buffering effects and the linear `foldLeft` merge on the driver. The changes are

1) B starts small. After each flush+compress, set $B \leftarrow \lceil \alpha\,|S| \rceil$, where $\alpha > 1$.
2) Sketches tree reduced recursively in the driver.

Note that we have not modified Spark. We use the stock implementation Spark GK Sketch for performance results in Section VI. This modified version is used in analysis only.

Let T_{mSGK} denote the per-insertion time complexity for the Modified Spark GK Sketch (mSGK) implementation.

$$T_{\text{mSGK}} = T_{\text{insert}} + T_{\text{compress}}$$

$$\begin{aligned} T_{\text{compress}} &= \frac{1}{n}\sum_{j=1}^{k} O(\alpha|S(n_j)|) \\ &= \frac{1}{n} O(\alpha|S(n_1)| + \alpha|S(n_2)| + \cdots + \alpha|S(n_k)|) \end{aligned} \tag{8}$$

where k is the number of calls to compress and j is the jth call to compress. The $\frac{1}{n}$ factor appears because we are amortizing the cost of the compressions across n inserts.

Let us use a continuous approximation.

$$H(n) = \alpha|S(n)| = \frac{\alpha}{\varepsilon}\log(\varepsilon n)$$

$$k \approx \int_{n_0}^{n} \frac{dx}{H(x)} = \frac{\varepsilon}{\alpha} \int_{n_0}^{n} \frac{dx}{\log(\varepsilon x)}$$

Let $u = \varepsilon x$ and $du = \varepsilon dx$, so the above becomes

$$k \approx \frac{1}{\alpha} \int_{\varepsilon n_0}^{\varepsilon n} \frac{du}{\log u} = \frac{1}{\alpha} \left(\mathrm{li}(\varepsilon n) - \mathrm{li}(\varepsilon n_0)\right)$$

$$\mathrm{li}(x) = \frac{x}{\log x} + O\left(\frac{x}{(\log x)^2}\right) \approx \frac{x}{\log x}$$

$$k \approx \frac{1}{\alpha} \left(\mathrm{li}(\varepsilon n) - \mathrm{li}(\varepsilon n_0)\right) = \frac{1}{\alpha} \frac{\varepsilon n}{\log(\varepsilon n)} - \frac{1}{\alpha} \frac{\varepsilon n_0}{\log(\varepsilon n_0)}$$

Assuming $n \gg n_0$, we can rewrite the above as

$$k \approx \frac{1}{\alpha} \frac{\varepsilon n}{\log(\varepsilon n)} \tag{9}$$

Amortizing over n insertions,

$$T_{\text{compress}} = \frac{1}{n} \sum_{j=1}^{k} O(\alpha |S(n_j)|) < O\big(\tfrac{k}{n} \cdot \alpha \cdot \tfrac{1}{\varepsilon} \log(\varepsilon n)\big) \tag{10}$$

Substituting (9) into (10) yields the amortized compress time

$$T_{\text{compress}} = O\left(\frac{1}{n} \cdot \frac{1}{\alpha} \frac{\varepsilon n}{\log(\varepsilon n)} \cdot \alpha \cdot \frac{1}{\varepsilon} \log(\varepsilon n)\right)$$

which simplifies to

$$T_{\text{compress}} = O(1)$$

Appending to the buffer takes $O(1)$ until the buffer is full. Because copying into a new buffer is $O(B)$ where $B_j = \alpha \cdot |S_{j-1}|$, an analogous derivation as was used for compressing yields $O(n)$ cost for n appends, giving amortized $O(1)$ per append.

However, before a buffer can be merged into the sketch it must be ordered, which takes $B_j \log B_j$ time. Thus

$$\begin{aligned} T_{\text{insert}} &= O(1) + \frac{1}{n} \sum_{j=1}^{k} O(B_j \log B_j) \\ &= O(1) + \frac{1}{n} \sum_{j=1}^{k} O(|S_j| \log |S_j|) \end{aligned} \tag{11}$$

where

$$\sum_{j=1}^{k} O(|S(n_j)| \log |S(n_j)|) < O(k \cdot |S(n)| \cdot \log |S(n)|)$$

so Equation (11) becomes

$$T_{\text{insert}} = O\left(\frac{1}{n} \cdot k \cdot |S(n)| \cdot \log |S(n)|\right) \tag{12}$$

Substituting Equations (9) and (2) into (12) yields

$$T_{\text{insert}} = O\left(\frac{1}{n} \cdot \frac{1}{\alpha} \cdot \frac{\varepsilon n}{\log(\varepsilon n)} \cdot \frac{1}{\varepsilon} \log(\varepsilon n) \cdot \log\big(\tfrac{1}{\varepsilon} \log(\varepsilon n)\big)\right)$$

which simplifies to

$$T_{\text{insert}} = O\left(\frac{1}{\alpha} \log\big(\tfrac{1}{\varepsilon} \log(\varepsilon n)\big)\right) \tag{13}$$

Because $T_{\text{compress}} = O(1)$, the above becomes

$$T_{\text{mSGK}} = O\big(\log \tfrac{1}{\varepsilon} + \log\log(\varepsilon n)\big) \tag{14}$$

Often the first term is viewed as a constant, causing Equation (14) to simplify to

$$T_{\text{mSGK}} = O(\log\log(\varepsilon n))$$

This is the same time complexity as the Classical GK Sketch. However, because we may later tune ε to control the tradeoffs within GK Select, we prefer to keep the form presented in Equation (14).

When we insert all elements of a partition of size $n_i = \frac{n}{P}$, Equation (14) becomes

$$T_{\text{exec}} = O\big(\tfrac{n}{P} \log \tfrac{1}{\varepsilon} + \tfrac{n}{P} \log\log(\varepsilon \tfrac{n}{P})\big) \tag{15}$$

For large n and small ε the above becomes

$$T_{\text{exec}} = O\big(\tfrac{n}{P} \log\log(\varepsilon \tfrac{n}{P})\big) \tag{16}$$

Because we scale B as a constant factor larger than the space bound,

$$S_{\text{exec}} = O\big(\tfrac{1}{\varepsilon} \log(\varepsilon \tfrac{n}{P})\big) \tag{17}$$

Rather than 'foldLeft', if the driver implements a'treeReduce' that executes entirely in the driver, e.g., by performing a 'foldLeft' on pairs of partitions and then recursively until only one sketch remains, the time complexity becomes:

$$T_{\text{driver}} = \Theta\left(P \cdot \tfrac{1}{\varepsilon} \log(\varepsilon \tfrac{n}{P})\right) + \Theta\left(P \cdot \tfrac{1}{\varepsilon} \log(\varepsilon \tfrac{n}{P})\right) + \Theta\left(\tfrac{1}{\varepsilon} \log(\varepsilon n)\right)$$

Combining terms yields

$$T_{\text{driver}} = \Theta\left(P \cdot \tfrac{1}{\varepsilon} \log(\varepsilon \tfrac{n}{P})\right)$$

The driver and executor space complexity, network volume, and network latency remain unchanged from Sketch's GK implementation.

V. GK Select Algorithm

To compute the k-th smallest element efficiently under the Spark execution model, we propose *GK Select*, an exact quantile algorithm guided by an approximate pivot.

1) **Approximate quantile.** The driver invokes `approxQuantile`, which uses the GK Sketch to estimate the target quantile and broadcast the pivot π.
2) **Local partitioning.** Each executor runs `QuickPartition` within `mapPartitions`, counting C_i the elements below π in its local partition.
3) **Rank correction.** The driver performs `collect` to gather all C_i and computes $\Delta k = k - \sum_i C_i$, the distance between the approximate and true ranks.
4) **Candidate extraction.** If π is low, each executor uses QuickSelect to extract the Δk smallest values above π; if π is high, it extracts the Δk largest below π.
5) **Tree reduction.** The extracted candidate sets are merged via `treeReduce`. At each step, only values within Δk of π are retained; others are discarded.
6) **Final selection.** After the final reduce at the driver, Δk candidates remain, representing the gap between the approximate and exact quantile. The boundary value (or its neighbor if parity requires) is the exact quantile.

Time Complexity	Step	Description
$T_{\text{GK,e}}$	S:1	Executors construct local GK sketches
$T_{\text{GK,d}}$	S:2	Driver collects and merges GK sketches
$T_{\text{b,d}}$	S:3	Driver broadcasts approximate
$T_{\text{qp,e}}$	S:4	Executor local Hoare (Quick) partitions
$T_{k,\text{e}}$	S:5	Executor local counts below pivot
$T_{\text{collect,d}}$	S:6	Driver collects counts
$T_{\text{bk,d}}$	S:7	Driver broadcasts Δk
$T_{\text{select,e}}$	S:8	Executor selects candidates
$T_{\text{reduce,e}}$	S:9	Reduce across executors
$T_{\text{final,d}}$	S:10	Driver finds the exact quantile

TABLE III
TIME COMPLEXITY NOTATION FOR GK SKETCH. SUBSTITUTE S FOR T TO DENOTE THE ANALOGOUS SPACE COMPLEXITY.

1) GK Select Executor Time Complexity:

$$T_{\text{exec}} = T_{\text{GK,e}} + T_{\text{qp,e}} + T_{k,\text{e}} + T_{\text{select,e}} + T_{\text{reduce,e}} \tag{18}$$

For purposes of this analysis we adopt the Modified Spark GK Sketch implementation, because it preserves the time complexity of Classical GK Sketch.

$$T_{\text{GK,e}} = O\big(\tfrac{n}{P}\log\tfrac{1}{\varepsilon} + \tfrac{n}{P}\log\log(\varepsilon\tfrac{n}{P})\big) \tag{19}$$

$$T_{\text{qp,e}} = O\big(\tfrac{n}{P}\big)$$

Because the Hoare (Quick) Partition partially orders the elements as the number below the pivot value is the count, which takes $O(1)$ time to determine and report; therefore

$$T_{k,\text{e}} = O(1)$$

$$T_{\text{select,e}} = O\big(\tfrac{n}{P}\big)$$

During a tree reduce there are P leaves, but $P/2 + P/4 + \cdots + 1 = P - 1$ merges of Step S:9 candidates, and thus the average computation per executor becomes

$$T_{\text{reduce,e}} = O(\varepsilon n)$$

Summing terms yields

$$T_{\text{exec}} = O\big(\tfrac{n}{P}\log\tfrac{1}{\varepsilon} + \tfrac{n}{P}\log\log(\varepsilon\tfrac{n}{P})\big) + O(1) + O\big(\tfrac{n}{P}\big) + O(\varepsilon n)$$

which simplifies to

$$T_{\text{exec}} = O\big(\tfrac{n}{P}\log\tfrac{1}{\varepsilon} + \tfrac{n}{P}\log\log(\varepsilon\tfrac{n}{P}) + \varepsilon n\big)$$

Because selecting the Δk closest to the approximate quantile never needs process more than the number of keys in the partition, $O(\varepsilon n)$ is dominated by the other two terms and the cost simplifies to

$$T_{\text{exec}} = O\big(\tfrac{n}{P}\log\tfrac{1}{\varepsilon} + \tfrac{n}{P}\log\log(\varepsilon\tfrac{n}{P})\big)$$

2) GK Select Driver Time Complexity:

$$T_{\text{driver}} = T_{\text{GK,d}} + T_{\text{b,d}} + T_{\text{collect,d}} + T_{\text{bk,d}} + T_{\text{final,d}}$$

Cost to collect and merge GK sketches using the Modified Spark GK Sketch implementation yields:

$$T_{\text{GK,d}} = \Theta\big(\tfrac{P}{\varepsilon}\log(\varepsilon\tfrac{n}{P})\big)$$

Torrent broadcast to distribute the approximate quantile:

$$T_{\text{b,d}} = O(1)$$

Cost to collect counts, sum them, and determine the difference between the approximate and the true quantile:

$$T_{\text{collect,d}} = O(P)$$

Torrent broadcast of the difference between the approximate and the true quantile:

$$T_{\text{bk,d}} = O(1)$$

Time to find the minimum or maximum of the numbers returned from the `treeReduce` of candidates:

$$T_{\text{final,d}} = O(\varepsilon n)$$

Summing components yields

$$T_{\text{driver}} = O\big(\tfrac{P}{\varepsilon}\log(\varepsilon\tfrac{n}{P}) + \varepsilon n\big)$$

3) GK Select Executor Space Complexity: GK Select incurs the space requirements of the Classical GK Select. However, because it Hoare partitions around the approximate pivot and then Quickselect those within Δk, it must materialize the entire partition once for the Hoare partition, persist it, and materialize it again for the Quickselect. These are artifacts of the way Spark streams data sequentially in each partition when performing `mapPartitions`. These artifacts represent well motivated design decisions as they allow Spark to stream the data from disk when the partition exceeds the memory available in an executor. However, it does incur an additional $O(\frac{n}{P})$ space requirement,

$$S_{\text{exec}} = O(\tfrac{1}{\varepsilon}\log(\varepsilon\tfrac{n}{P}) + \tfrac{n}{P})$$

Because a sketch can never contain more than all of the keys in the partition, the second term dominates yielding

$$S_{\text{exec}} = O(\tfrac{n}{P})$$

4) GK Select Driver Space Complexity: To implement GK Select, the driver incurs all of the space requirements of GK Sketch plus the space required to hold the final Δk closest candidates to the approximate quantile,

$$S_{\text{driver}} = O\left(\tfrac{P}{\varepsilon}\log(\varepsilon\tfrac{n}{P}) + \varepsilon n\right)$$

5) Algorithm Tradeoffs: We assemble time complexities for all algorithms in Tables IV and V. Asymptotically AFS and Jeffers Select clear have the best executor time complexity. Assuming that the values are evenly distributed across the partitions we would expect near linear speed up. However a significant difference in cost structure appears in Table V: AFS and Jeffers Select both update a pivot such that it converges to the exact kth order statistics in $O(\log n)$ rounds. Every round acts as stage barrier preventing progress until the next pivot value is determined.

AFS appears to have better driver time complexity, but this is achieved by introducing a `treeReduce` of the counts rather than a `collect`. A `treeReduce` will always have more communication overhead to set up the tree spanning executors, which it recovers if the computation at each reduce step is substantial, but in the case of AFS the only computation being performed at each node in the reduce is to sum the counts of the number of values below the pivot. Only in a regime with extremely large clusters might AFS outperform Jeffers Select.

GK Select has the same executor time complexity as the classical GK Sketch, but this is only because Hoare's Quick-Partition and Quickselect run in linear time. The additional operations performed by GK Select will increase the constant factor hidden by time complexity analysis relative to GK Sketch.

The biggest downside of GK Select appears to be on the driver side. All candidates within Δk of the approximate quantile are reported to the driver and then driver selects the exact quantile as the farther of the candidates. This introduces additional driver side time and space complexity of $O(\Delta k)$ where $\Delta k \leq \varepsilon n$. If $\varepsilon > \frac{1}{P}$ then the number of keys processed by the driver can exceed the keys in the average-sized partition. Because driver nodes are often less well-endowed than executors this could become a problem.

If ε is set too low then the size of the sketches may limit performance given that sketches grow roughly inversely proportional to ε. One might be tempted to set $\varepsilon = \frac{1}{P}$, but then driver time and space complexity grows roughly with the square of the number of partitions. If we intend to process such large GK sketches, it might make sense to perform a `treeReduce` when merging sketches between partitions rather than performing a `collect` and merging on the driver. Doing so would push merge overhead into the cluster allowing us to accommodate much larger sketches, achieving proportionally lower error in 'approxQuantile' and thus proportionally reduce the size of the Δk closest returned to the driver in the last step of GK Select.

VI. Results

A. Experimental Setup

In order to effectively compare the performance of the proposed GK select algorithm, the execution time was measured across the following data set sizes ranging from 1 million to 1 billion random integers. The number of cores was varied across each of the data set sizes, ranging from 12 partitions (3 cores) to 120 partitions (30 cores). Each trial was run 100 times, computing the mean time at the end. The resulting times were compared across several algorithms: Full sort, AFS, Jeffers, and GK Sketch. The GK Sketch algorithm in this case is the Apache Spark implementation, `approxQuantile`. We used AWS EMR release emr-7.9.0 with instance type `m5.xlarge`, having 4 vCore, 16 GiB memory running Spark version 3.5.5 and Hadoop version 3.4.1. Storage type is EBS using 15GiB volumes.

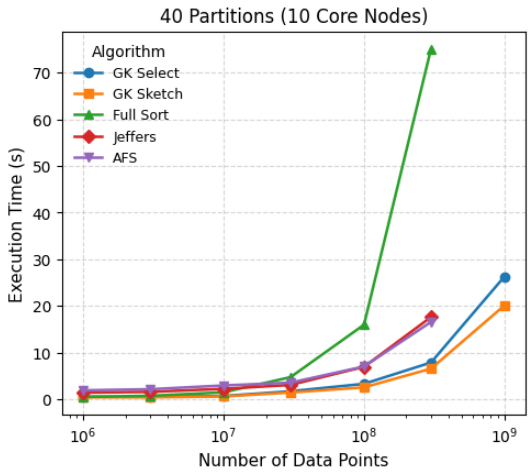

Fig. 1. Performance with 10 cores

TABLE IV
ASYMPTOTIC EXECUTOR AND DRIVER COMPLEXITY FOR QUANTILE METHODS.

Algorithm	Executor time	Driver time	Executor memory	Driver memory
Spark Full Sort	$O\left(\frac{n}{P}\log\frac{n}{P}\right)$	$O(Pk\log(Pk))$	$O\left(\frac{n}{P}\right)$	$O(Pk)$
AFS	$O\left(\frac{n}{P}\right)$	$O(\log n)$	$O\left(\frac{n}{P}\right)$	$O(1)$
Jeffers Select	$O\left(\frac{n}{P}\right)$	$O(P\log n)$	$O\left(\frac{n}{P}\right)$	$O(P)$
Classical GK Sketch	$O\left(\frac{n}{P}\log\frac{1}{\varepsilon}+\frac{n}{P}\log\log(\varepsilon\frac{n}{P})\right)$	n/a (streaming)	$O\left(\frac{1}{\varepsilon}\log(\varepsilon\frac{n}{P})\right)$	n/a (streaming)
Spark GK Sketch	$O\left(\frac{n}{P}\log B+\frac{1}{\varepsilon}\frac{n}{P}\frac{1}{B}\log(\varepsilon\frac{n}{P})\right)$	$\Theta\left(\frac{P}{\varepsilon}\log(\varepsilon n)\right)$	$O\left(B+\frac{1}{\varepsilon}\log(\varepsilon\frac{n}{P})\right)$	$O\left(\frac{P}{\varepsilon}\log(\varepsilon\frac{n}{P})\right)$
GK Select	$O\left(\frac{n}{P}\log\frac{1}{\varepsilon}+\frac{n}{P}\log\log(\varepsilon\frac{n}{P})\right)$	$O\left(\frac{P}{\varepsilon}\log(\varepsilon\frac{n}{P})+\varepsilon n\right)$	$O\left(\frac{n}{P}\right)$	$O\left(\frac{P}{\varepsilon}\log(\varepsilon\frac{n}{P})+\varepsilon n\right)$

TABLE V
COMMUNICATION AND STAGE COMPLEXITY FOR QUANTILE METHODS.

Algorithm	Network latency	Network volume	# Shuffle stages	# Actions	# Persists	Exact/Approx.
Spark Full Sort	$gk+gP\log P+g\frac{n}{P}+2L$	$O(n)$	1	2	0	Exact
AFS	$(L+gh\log P)O(\log n)$	$O(P\log n)$	0	$O(\log n)$	$O(\log n)$	Exact
Jeffers Select	$(L+gh\log P)O(\log n)$	$O(P\log n)$	0	$O(\log n)$	$O(\log n)$	Exact
Spark GK Sketch	$g\,O\left(\frac{1}{\varepsilon}\log(\varepsilon\frac{n}{P})\right)+L$	$O\left(\frac{P}{\varepsilon}\log(\varepsilon\frac{n}{P})\right)$	0	1	0	Approx.
GK Select	$g\,O\left(\frac{1}{\varepsilon}\log(\varepsilon\frac{n}{P})\right)+2g\log P+gP+g\varepsilon n\log P+3L$	$O\left(\frac{P}{\varepsilon}\log(\varepsilon\frac{n}{P})+\varepsilon nP\right)$	0	3	1	Exact

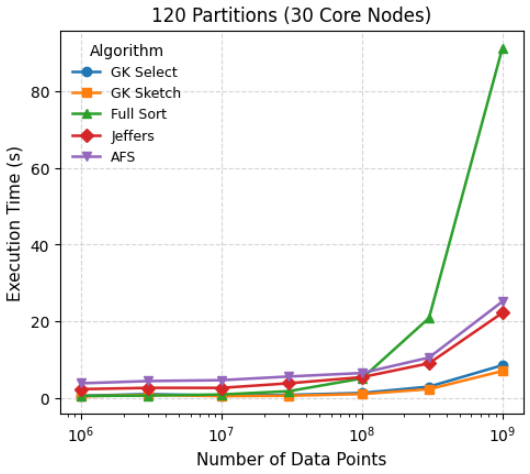

Fig. 2. Performance with 30 cores

In the 30-core configuration (Figure 2), parallelism compresses absolute runtimes across the board while preserving the relative ordering observed at smaller core counts. GK Sketch and GK Select remain dominant: they stay sub-second through 10^7 points, are only $1-1.3$ s at 10^8, and increase to roughly $7-9$ s even at 10^9 points. This shallow growth indicates near-linear scaling with small constant factors. Notably, these results show that even as n becomes very large, GK Select achieves exact computation with only a modest time penalty—there is little practical sacrifice in runtime for exactness under multi-core execution. By contrast, Full Sort benefits from parallelism at moderate sizes but exhibits the expected superlinear growth as n rises, and Jeffers/Al-Furaih do not extend to the largest inputs in this plot, suggesting less favorable scaling or resource limits relative to the GK methods.

VII. CONCLUSION

We revisited exact quantile computation in distributed systems and introduced *GK Select*, a sketch-guided selection algorithm that couples a deterministic approximate summary with an exact, partition-based selection tailored to Spark. By using a GK summary to supply a high-quality pivot and structuring execution into a constant number of broadcast/reduce stages, GK Select avoids the full global reorder required by PSRS and collapses synchronization to a few stages. Empirically, GK Select matches the speed of the approximate GK operator while returning exact answers, consistently outperforming full sort and beating count-and-discard baselines (Jeffers, AFS) whose multiple rounds amplify shuffle and latency costs.

Methodologically, GK Select uses Spark's built-ins to keep work local and communication bounded: *approxQuantile* seeds the pivot, *mapPartitions()* performs local Quickpartition, a lightweight *collect()* estimates rank error to obtain Δk, partitions extract only candidates within Δk of the target rank, and a *treeReduce()* prunes to the exact answer—restricting cross-partition movement to a narrow window rather than shuffling all n records. This communication pattern aligns with logarithmic fan-in/fan-out rather than centralized driver–executor exchanges, and it eliminates iterative global rounds. Across datasets from 10^4 to 10^9 elements and up to 30 cores (averaged over 100 trials), GK Select remained sub-second through $n = 10^7$ and 1–1.3s at $n = 10^8$. Taken together, these results show that exact quantiles need not incur a full shuffle: with high-quality pivots and constant-stage synchronization, GK Select delivers exactness at approximate-like cost under Spark's execution model.

REFERENCES

[1] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, no. 7, p. 321, Jul. 1961. [Online]. Available: https://doi-org.umiss.idm.oclc.org/10.1145/366622.366644

[2] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Linear time bounds for median computations," in *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '72. New York, NY, USA: Association for Computing Machinery, 1972, p. 119–124. [Online]. Available: https://doi-org.umiss.idm.oclc.org/10.1145/800152.804904

[3] R. W. Floyd and R. L. Rivest, "Algorithm 489: the algorithm select—for finding the ith smallest of n elements [m1]," *Commun. ACM*, vol. 18, no. 3, p. 173, Mar. 1975. [Online]. Available: https://doi-org.umiss.idm.oclc.org/10.1145/360680.360694

[4] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 58–66. [Online]. Available: https://doi-org.umiss.idm.oclc.org/10.1145/375663.375670

[5] R. W. Floyd and R. L. Rivest, "Expected time bounds for selection," Stanford University, Tech. Rep. CS-TR-73-349, 1973, available at https://i.stanford.edu/pub/cstr/reports/cs/tr/73/349/CS-TR-73-349.pdf.

[6] R. J. Cole, "An optimally efficient selection algorithm," *Information Processing Letters*, vol. 26, no. 6, pp. 295–299, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/002001908890186X

[7] Y. Han, "Optimal parallel selection," *ACM Trans. Algorithms*, vol. 3, no. 4, p. 38–es, Nov. 2007. [Online]. Available: https://doi-org.umiss.idm.oclc.org/10.1145/1290672.1290675

[8] I. Al-Furaih, S. Aluru, S. Goil, and S. Ranka, "Practical algorithms for selection on coarse-grained parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 8, pp. 813–824, 1997.

[9] I. Jeffers, "Parallel/distributed quickselect implementation," https://github.com/ianjeffers/QuickSelect, 2020, accessed April 2025.

[10] H. Shi and J. Schaeffer, "Parallel sorting by regular sampling," *Journal of parallel and distributed computing*, vol. 14, no. 4, pp. 361–372, 1992.

[11] G. Cormode, "Data sketching," *Commun. ACM*, vol. 60, no. 9, p. 48–55, Aug. 2017. [Online]. Available: https://doi-org.umiss.idm.oclc.org/10.1145/3080008

[12] Z. Chen and A. Zhang, "A survey of approximate quantile computation on large-scale data," *IEEE Access*, vol. 8, pp. 34 585–34 597, 2020.