Quantiles at scale.

Mike Shi

@mikeshi42

ABOUT ME

Hi, I'm Mike.

- Lead observability at ClickHouse, working on ClickStack.
- Co-founded HyperDX, acquired by ClickHouse.
- I care about open-source observability from polished UX and docs to fast search and ingestion.

What is a quantile?

A way to ask: where does a value sit in a sorted distribution?

`q · N`	A quantile returns the value at position `q × N` after sorting. q ∈ [0,1], N = total count.
`p50`	The **median:** half the observations sit below it, half above.
`p99`	The value **99%** of observations fall below. One request in 100 is slower than this.

TEN REQUEST LATENCIES, SORTED (MS)

8	12	15	19	23	28	35	47	81	240
r=1	r=2	r=3	r=4	r=5	p50	r=7	r=8	p90	p99

p50 = 28 ms: the middle value **p99 = 240 ms:** nearly 9× higher

mean = 55.8 ms understates the tail and overstates the typical experience

Some important definitions

The language we'll use for error bounds throughout the rest of the talk.

Rank	Number of elements ≤ v in the stream. Rank ∈ [1, N].	`rank(55 ms) = 6`
Normalised rank	rank(v) / N. Converts position count into a population fraction. ∈ (0,1].	`6 / 8 = 0.75`
Quantile Q(q)	The inverse: given fraction q, return the value v whose normalised rank is closest to q.	`Q(0.75) = 55 ms`
Percentile	A quantile expressed as a percentage. p99 = Q(0.99). Percentile is common in SRE; quantile is standard in stats.	`p75 = Q(0.75)`

Two types of query

One dominates observability. One shows up occasionally. They share a sketch.

MOST COMMON

Quantile query

```
Q(0.99) = ?
```

"What value sits at the 99th percentile?"

→ `240 ms`

Almost every SLO and dashboard runs this direction. "is our p99 below 200 ms?"

OCCASIONAL

Rank query

```
rank(200 ms) = ?
```

"What fraction of requests finished within 200 ms?"

→ `0.94 (94th percentile)`

Appears when you have a fixed threshold and need to know how often you breach it.

How do we compute this?

There are several approaches. Each one has different trade-offs.

Two approaches to computing quantiles

Exact or approximate: a fundamental trade-off between memory and accuracy.

EXACT

Store every value. Sort at query time.

Quantiles are computed by sorting the materialised dataset. The answer is always correct.

Rank error = 0

EXAMPLE ALGORITHMS

Exact (sort-based) | Exact weighted | Interpolated weighted

SKETCH · ESTIMATE

Compress the stream into a small summary.

Queries return approximate answers within a guaranteed error bound ε. Memory stays bounded regardless of how much data has been seen.

Bounded memory **Guaranteed ε**

ERROR IS GUARANTEED AND BOUNDED & ALGORITHMS DIFFER IN *HOW* ERROR IS MEASURED

Absolute rank error: at most ε·N positions from the true rank

Relative rank error: error shrinks near the tails (better for p99, p99.9)

Value error: returned value is within ε of the true value

Exact doesn't work for observability

Both exact strategies break down under conditions observability demands at scale.

quantileExact: SORT-BASED

- Full dataset materialisation
- O(N log N) per query
- Not streamable
- O(N) memory

quantileExactWeighted: HASH-MAP BASED

- Works only at low cardinality
- Breaks at high cardinality
- O(D) memory: distinct values

State is bounded only if cardinality is bounded. Observability values rarely are.

clickhouse-client

```
SELECT quantileExact(0.9)(memory_usage) FROM query_log WHERE memory_usage > 0
┌─quantileExac⋯mory_usage)─┐
│              36 139 493  │
└──────────────────────────┘

1 row. Elapsed: 15.221 sec. Processed 7.32 billion rows , 58.53 GB (480.64 million rows/s., 3.85 GB/s.)
Peak memory usage: 39.00 GiB.
```

Sketch-based techniques

Two collection points: raw-data and pre-aggregation at source. Each comes with constraints.

COMPUTE FROM RAW DATA IN THE DATASTORE

Raw data lands in the datastore. Sketches are computed two ways: **insert-time** incremental, or **query-time** by reading the stored data.

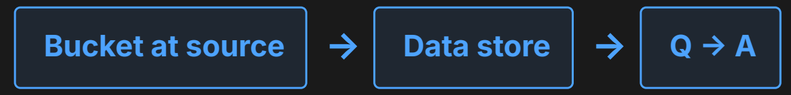
Insert-time: bounded memory, works at high cardinality, no large sorts, but you must **predefine what to measure** at insert time.

Query-time scan: flexible (any quantile over any slice), but still **reads the full stored dataset** .

Used by analytical datastores: **ClickHouse** (insert- or query-time), **Elasticsearch**.

PRE-AGGREGATE AT SOURCE

Sketches are computed at collection, before data reaches the store. Eliminates raw-data storage but requires **predefining what to measure** at instrumentation time.



Fixed schema: bucket boundaries and label dimensions must be decided up-front. Changing them means re-instrumenting the source.

Efficient at scale: the store only receives aggregates, not raw events, dramatically lower ingestion volume.

Used by Prometheus-based stores: **Prometheus**, **Thanos**, **Grafana Mimir**

Desirable properties of sketches at scale

BOUNDED MEMORY

Memory is controlled by a user-tunable parameter k. The sketch never grows beyond $O(k)$.

High-cardinality streams need per-series state that is fixed and predictable.

FAST INSERTS

Each value is incorporated without additional sorting or scanning steps.

Sketches must keep up with high-throughput streams without becoming a bottleneck.

QUERY TIME INDEPENDENT OF N

Quantile queries take $O(k)$ or $O(\log k)$ time to compute. Proportional to the size of the sketch, instead of the size of the dataset.

Query latency must be predictable regardless of dataset size or query time window.

MERGEABILITY

Two sketches over separate partitions can combine into one sketch representative of both.

Without it you can't distribute the algorithm across shards or aggregate pre-computed state.

UNDERSTOOD ERROR TYPE

Error is clearly defined and ideally configurable. **Rank-** or **value-based**; **absolute** or **relative**.

The error type determines whether the algorithm is suitable for your SLO target.

DETERMINISM (desirable)

Same input produces the same output. Not all algorithms guarantee this, some are randomised or merge-order sensitive.

Useful for reproducible results, A/B comparisons, and debugging.

ClickHouse

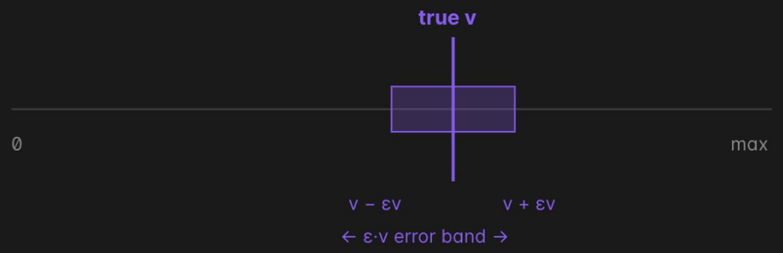
Two types of error

Two fundamentally different ways to measure how accurate a quantile estimate is.

VALUE-BASED ERROR

$$|\text{returned} - \text{true}| \,/\, \text{true} \leq \varepsilon$$

The returned *value* is within a fraction ε of the true quantile value, proportional to the magnitude of that value.

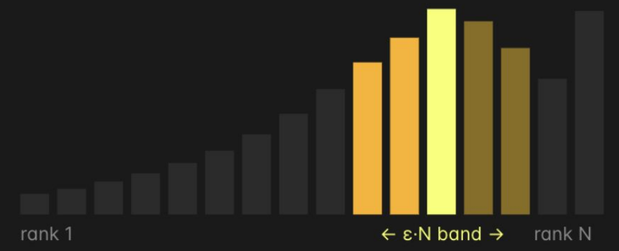


Example. True p99 = 200 ms · ε = 0.01 → within **±2 ms** (198–202 ms). True p99.9 = 4,000 ms → within **±40 ms** (3,960–4,040 ms). The bound always scales with the value.

RANK-BASED ERROR

$$|\text{rank}(\text{returned}) - q \cdot N| \leq \varepsilon \cdot N$$

The returned element's *position* in the sorted sequence is within ε·N of the target rank. The actual value is not directly bounded.



Example. N = 1,000 · q = 0.99 · ε = 0.01 → rank error ≤ **10 positions** . Any element ranked 980–1,000 may be returned. Whether those positions span 2 ms or 200 ms depends entirely on the data.

N Total number of values in the dataset

q Quantile level: a number in [0, 1]

ε Error parameter: accuracy vs memory trade-off

ClickHouse

Choosing an error type for observability

USE RANK ACCURACY WHEN…

…you care about **how many items fall above or below a threshold.**

> EXAMPLE SLO
>
> *"99% of requests must complete within 300 ms."*
>
> The algorithm must not misclassify too many requests on either side of the 300 ms cutoff. A bounded rank error limits exactly how many it can get wrong.

SLO compliance checks against a fixed threshold

Alerting: "has the error rate crossed X%?"

Classification: "is this in the slowest 1%?"

`quantileGK` `quantileDeterministic` `DataSketches Classic / KLL / REQ`

USE VALUE ACCURACY WHEN…

…you care about **how large the value is at a given percentile.**

> EXAMPLE SLO
>
> *"p99 latency must not increase by more than 2% between releases."*
>
> A 5% value error would swamp the 2% signal you're trying to detect. Only a value-bounded sketch can reliably catch small regressions.

Time-series dashboards tracking p99 / p99.9 over time

Canary and release comparisons: "is this deploy slower?"

Measurement: "what is my p99 latency in ms?"

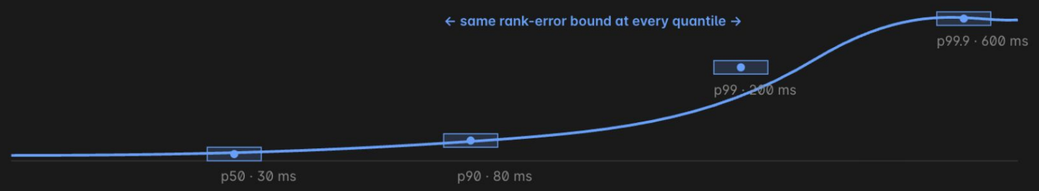
`quantileDD` `quantileBFloat16` `Exponential histograms`

ClickHouse

Rank error: allocation across quantiles

UNIFORM RANK ERROR

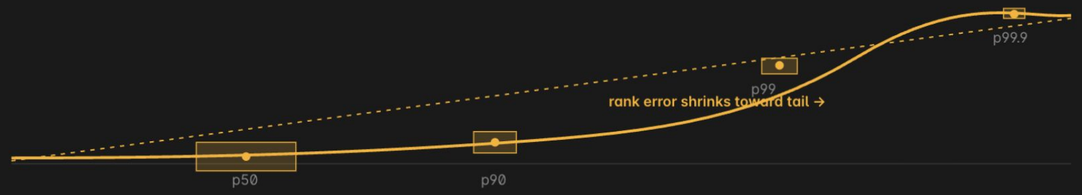
Same rank budget at every quantile.



Predictable and consistent, but the same precision must cover the median *and* the extreme tail.

TAIL-BIASED RANK ERROR

Accuracy concentrates toward higher quantiles.



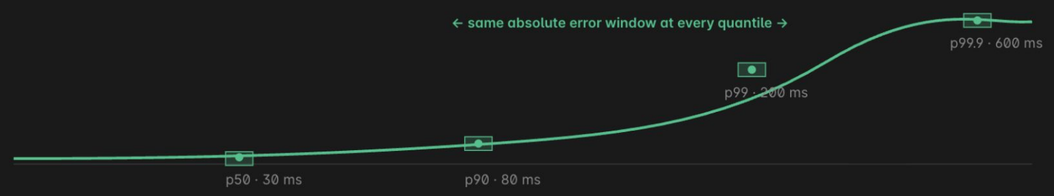
Concentrates precision where tail monitoring matters most. Accuracy at p99.9 far exceeds p50 at the cost of looser precision near the median.

Value error: absolute and relative

Value-error algorithms bound `|returned - true|` in the value domain

CONSTANT ABSOLUTE ERROR

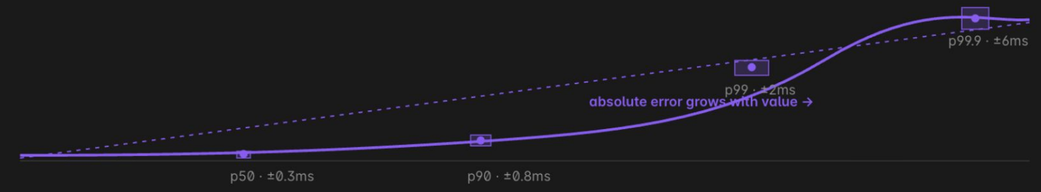
Fixed ±Δ value band at every quantile (e.g. fixed-width histogram binning).



Error is ±Δ ms regardless of the true value. Keeping Δ small across a wide range is expensive: 0–10,000 ms at ±3 ms takes 1,600+ buckets.

CONSTANT RELATIVE ERROR

ε% of the true value at every quantile.



At ε = 1%: p50=30 ms gives ±0.3 ms; p99.9=600 ms gives ±6 ms. Often the most desirable for observability, proportional differences matter more than absolute ones at the tail.

Algorithms by error type

Every quantile algorithm makes a tradeoff: what dimension is the error in, and is it constant or variable across the quantile range?

	VALUE ERROR	RANK ERROR
CONSTANT	**Constant absolute error** Fixed ±Δ value band at every quantile. Same absolute precision at p50 and p99.9. Expensive for wide value ranges; bucket count scales linearly with range. `Timing Histogram` `Prometheus Histogram`	**Constant rank error** Fixed ±ε·N rank positions at every quantile. The same count of misranked elements at p50 and p99.9. A uniform guarantee. `Reservoir Sampling` `Deterministic Reservoir` `Greenwald-Khanna` `KLL Sketch` `DataSketches Classic`
VARIABLE	**Relative value error** Error scales as ε% of the true value. Higher latencies carry a proportionally wider band. Naturally acceptable when proportional differences matter more than absolute ones. `DDSketch` `BFloat16 Histogram` `Prometheus Exponential`	**Tail-biased rank error** Rank error budget shrinks toward higher quantiles, concentrating precision at the tail. More accurate at p99.9 than at p50, suited to SLO monitoring. `T-Digest` `REQ Sketch`

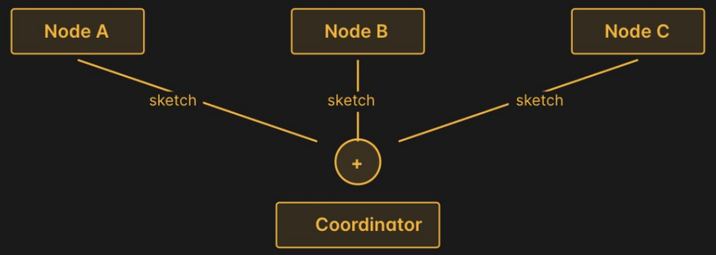
ClickHouse

Mergeability in practice

A mergeable sketch enables two capabilities that matter most at scale: distributed computation and incremental updates.

DISTRIBUTED COMPUTATION

A query fans out to independent nodes. Each builds a local sketch over its partition. Only compact sketches travel to the coordinator, which merges them into the final answer.

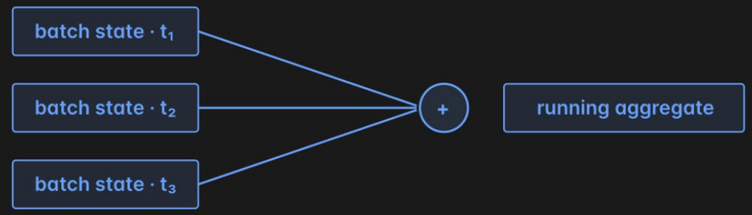


Why it matters: without mergeability, every node must ship raw rows to one machine. Mergeability keeps local data and sketch states small.

Works across nodes in a distributed cluster, replicas, or CPU threads within a single server.

INCREMENTAL UPDATES

Data arrives in batches over time. Each batch produces a sketch state. Every new state merges into the same running aggregate without reprocessing historical data.



Why it matters: a sketch built over yesterday's data merges with today's. The quantile stays current without replaying any historical events.

Used in ClickHouse `AggregatingMergeTree` and materialised views, where partial states merge on compaction.

Sketch algorithms at a glance

All algorithms are bounded-memory estimates. k = accuracy/size parameter, b = bucket count, N = elements seen.

ALGORITHM	INSERT	QUERY	MEMORY	MERGEABLE	ERROR TYPE	DETERM.	IN CLICKHOUSE
Timing Histogram	O(1)	O(1)	O(1)	Exact	Value · Constant	✓	✓
Prometheus Histogram	O(log b)	O(b)	O(b)	Exact	Value · Constant	✓	✓
Reservoir Sampling	O(1)	O(k log k)	O(k)	Approx	Rank · Constant	✗	✓
Deterministic Reservoir	O(1)	O(k log k)	O(k)	Approx	Rank · Constant	✓	✓
Greenwald-Khanna	O(1)†	O(k)	O(k log N)	Exact	Rank · Constant	✓	✓
KLL Sketch	O(1)†	O(k)	O(k)	Exact	Rank · Constant	✗	Exp.‡
DataSketches Classic	O(1)†	O(k log N)	O(k log N)	Exact	Rank · Constant	✗	✗
DDSketch	O(1)	O(k)	O(1/ε)	Exact	Value · Variable	✓	✓
BFloat16 Histogram	O(1)	O(k)	O(1)	Exact	Value · Variable	✓	✓
Prometheus Exp. Histogram	O(1)	O(b)	O(b)	Exact	Value · Variable	✓	Exp.‡
T-Digest	O(log k)†	O(log k)	O(k)	Approx	Rank · Variable	✗	✓
REQ Sketch	O(1)†	O(k)	O(k)	Exact	Rank · Variable	✗	Exp.‡

† amortised ‡ experimental branch **Exact** mergeable = merged sketch = single-pass result **Approx** = merge valid but order-dependent

DataSketches Classic is deprecated; KLL achieves the same with optimal O(k) memory.

ClickHouse

How to evaluate these algorithms?

A wide-event benchmark on **ClickHouse**. 7.32 billion rows from production query logs.

ClickHouse

Testing across data distributions

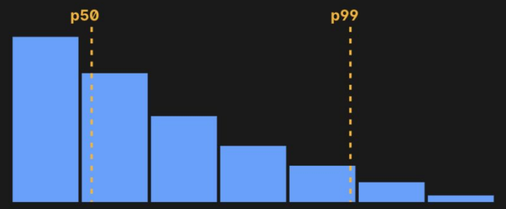
Observability signals follow different distributions, so we'll test the functions against a variety of them

TEST DATA: **7.32 billion rows** from the ClickHouse query log

query_duration_ms

HEAVY-TAILED

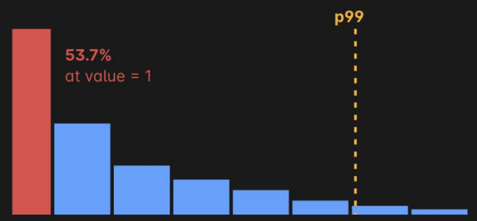
min 1 · median 9 · p99 1,315 · max 1,851,355



read_rows

POINT-MASS + HEAVY TAIL

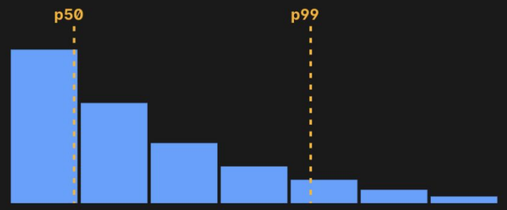
min 1 · median 1 · p99 1,015,017 · max 695B



read_bytes

HEAVY-TAILED

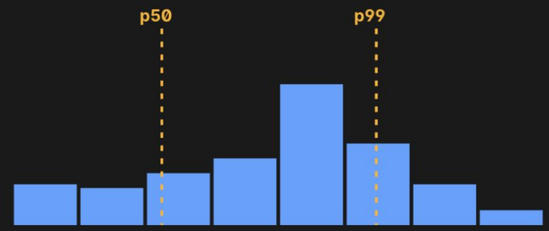
min 1 · median 16 · p99 17.6M · max 1.22T



memory_usage

MULTIMODAL

min 1 · median 64K · p99 38M · max 19.6G



What we measured

Five questions, each asked of every algorithm at the matched accuracy target.

1 **Accuracy across quantiles**

Value error against an exact ground truth at p50, p90, p99, p99.9.

2 **State generation cost**

CPU and wall-time to build the aggregation state from raw rows.

3 **State size**

Bytes per group after merge. Drives storage, network transfer, scan cost.

4 **Merge stability**

Does repeated merging compound error? Leaf vs single-merge vs hierarchical rollup.

5 **Query performance over merged states**

End-to-end latency reading a final quantile from a pre-computed state.

BASELINE: PROMETHEUS

Setting the accuracy bar

Max error across `p50`, `p90`, `p99`, `p99.9` on each metric. Lower is better.

METRIC	CLASSIC 23–36 BUCKETS	EXP S3 ~4.5% THEORETICAL	EXP S4 ★ ~2.2% THEORETICAL	EXP S5 ~1.1% THEORETICAL
query_duration_ms	13.1%	1.7%	0.8%	0.6%
read_rows	14.0%	8.6%	4.2%	2.1%
read_bytes	7.2%	7.0%	3.8%	1.9%
memory_usage	15.9%	4.5%	2.2%	1.0%

≤ 5% 5–10% > 10%

THE TARGET FOR EVERYTHING THAT FOLLOWS

≤ 4% average error across `p50`–`p99.9`, on every column.

Best of each family

Worst-case value error across all four columns at each quantile level.

ALGORITHM	P50	P90	P95	P99	P99.9	VERDICT
DDSketch(0.001) ★	0.10%	0.08%	0.08%	0.08%	0.09%	0.10% worst
quantileBFloat16 ★	0.33%	0.67%	0.43%	0.56%	0.60%	0.67% worst
GK(2000)	0.15%	0.92%	1.55%	2.36%	16.99%	overflows tail
KLL(1600)	0.26%	0.29%	3.97%	5.22%	30.95%	fails p99+
REQ(14)	1.70%	5.64%	0.88%	0.37%	0.99%	5.6% on p90
TDigest	3.64%	9.99%	2.14%	0.42%	1.22%	10% on p90
quantileTiming	0.00%	0.00%	0.00%	1.20%	0.16%	timing only
quantile (reservoir)	12.50%	27.42%	18.56%	19.47%	79.76%	unreliable

Only DD(0.001) and quantileBFloat16 clear ±4% on every column at every quantile. DD wins by an order of magnitude.

Building the state from raw rows

CPU time over 7.3B rows. Best parameter per algorithm.

FASTEST

1,050 ms

`quantile` default reservoir

FASTEST ACCURATE

1,097 ms

`quantileBFloat16` ≤0.67% worst

MOST ACCURATE

1,554 ms

`DD(0.001)` ≤0.10% worst

ALGORITHM	AVG QUERY TIME	AVG CPU TIME	VS FASTEST
`quantile`	1,050 ms	49,316 ms	1.0×
`quantileBFloat16`	1,097 ms	51,987 ms	1.0×
`quantileTiming`	1,088 ms	51,606 ms	1.0×
`DD(0.01)` — all DD within 11 ms	1,543 ms	73,486 ms	1.5×
`quantileTDigest`	2,202 ms	104,923 ms	2.1×
`KLL(1600)`	2,245 ms	107,117 ms	2.1×
`GK(2000)`	2,570 ms	119,274 ms	2.4×
`REQ(14)`	2,608 ms	124,442 ms	2.5×

What you actually pay for at scale

State size drives storage, network transfer between shards, and recurring merge cost.

ALGORITHM	ACROSS COLUMNS	AVG PER COLUMN	SCALE
DD(0.01)	3.4 – 6.3 KB	5.3 KB	Small
KLL(1600)	3.1 – 17.0 KB	7.5 KB	Small
quantileTDigest	7.5 – 7.8 KB	7.7 KB	Small, fixed
quantileTiming	12.2 KB (fixed)	12.2 KB	Small, fixed
REQ(14)	14.6 – 23.1 KB	17.6 KB	Medium
quantileBFloat16	11.6 – 25.1 KB	20.5 KB	Medium
quantile	19.7 – 34.6 KB	25.5 KB	Medium
GK(2000)	12.2 – 27.0 MB	18.2 MB	Extremely large

Mergeable in theory ≠ mergeable in practice

BIT-FOR-BIT IDENTICAL

4 algos

DD | BFloat16 | Timing | GK

WORST SINGLE-POINT SHIFT

13.54 pp

`quantile` reservoir, least stable

ALGORITHM	PRE-MERGE ERROR	POST-MERGE ERROR	AVG SHIFT (PP)	WORST POINT	STABILITY
DD(0.01)	0.54%	0.54%	0.00	0.00	Identical
quantileBFloat16 ★	0.28%	0.28%	0.00	0.00	Identical
quantileTiming	0.27%	0.27%	0.00	0.00	Identical
GK(2000)	2.70%	2.70%	0.00	0.00	Identical
quantileTDigest	1.42%	1.43%	+0.01	0.16	Stable
REQ(14)	0.59%	0.82%	+0.23	3.89	Stable
KLL(1600)	2.84%	2.73%	−0.11	11.22	Drifts at tails
quantile	6.61%	7.14%	+0.54	13.54	Some drift

The latency end-users actually feel

Not the same measurement as ② — here we query a pre-computed state and return a final quantile.

FASTEST

1.0 ms

quantileTiming

FAST & ACCURATE

1.2 ms

DDSketch

OUTLIER

76.4 ms

GK(2000) ~60× slower, deserialises 18 MB state per read

ALGORITHM	AVG QUERY TIME	AVG CPU TIME	ASSESSMENT
quantileTiming	1.0 ms	1,737 µs	Sub-2ms
DD(0.01)	1.2 ms	1,775 µs	Sub-2ms
quantileTDigest	1.2 ms	1,784 µs	Sub-2ms
KLL(1600)	1.4 ms	1,888 µs	Sub-2ms
quantile	1.6 ms	1,988 µs	Sub-2ms
quantileBFloat16 ★	1.6 ms	1,984 µs	Sub-2ms
REQ(14)	1.9 ms	2,075 µs	Sub-2ms
GK(2000)	76.4 ms	76,762 µs	~60× slower

PER-COLUMN BEST FIT

Which algorithm suits which distribution

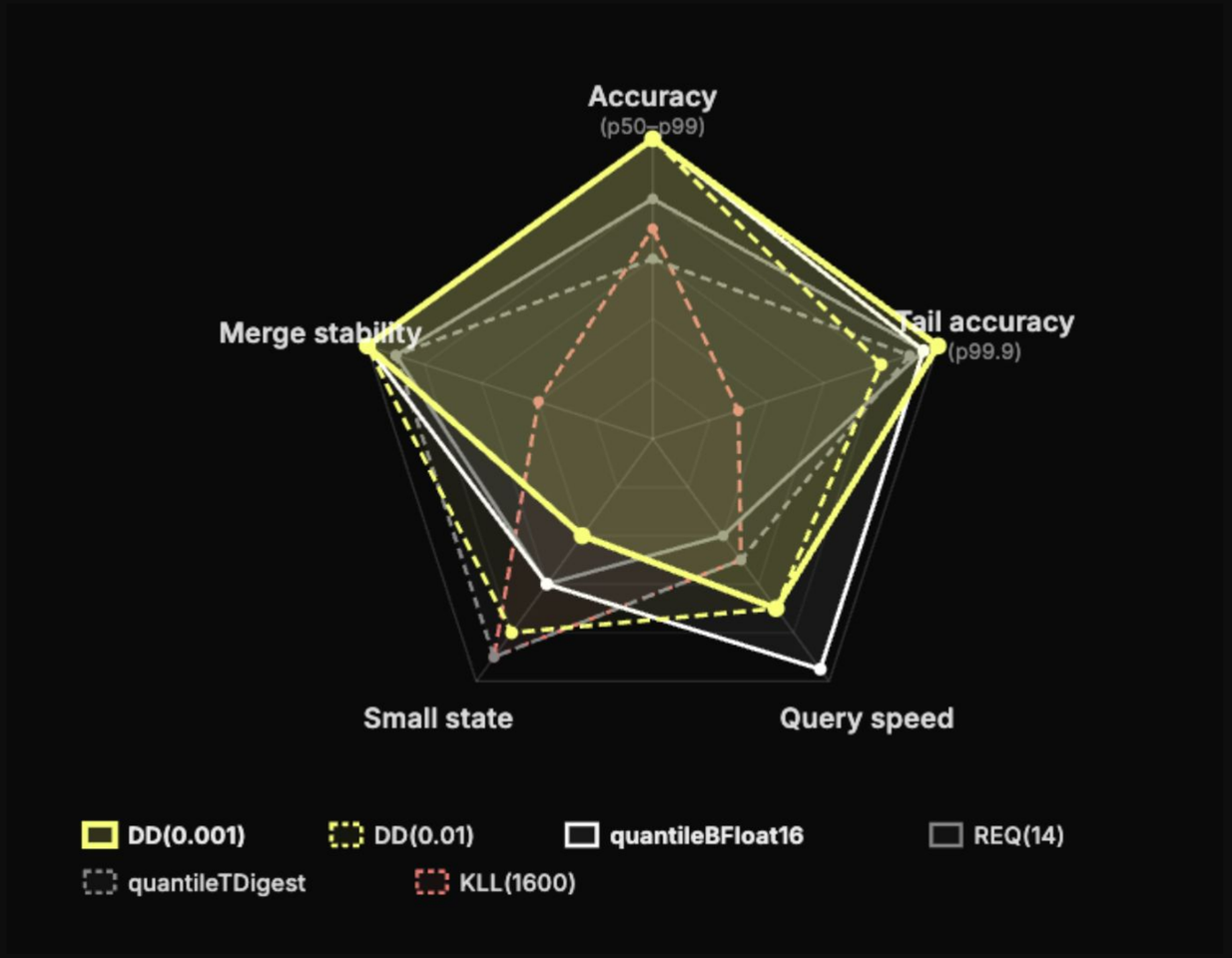
COLUMN	DISTRIBUTION	BEST FIT	WORST ERROR	NOTES
`query_duration_ms`	heavy-tailed	`DD(0.001)`	`0.05%`	`BFloat16` 0.46% · `quantileTiming` 1.20%
`read_rows`	point-mass + heavy tail	`DD(0.001)`	`0.10%`	`BFloat16` 0.67% · `TDigest` 9.99% at p90
`read_bytes`	heavy-tailed	`DD(0.001)`	`0.08%`	`REQ(14)` 0.30% · `BFloat16` 0.39%
`memory_usage`	multimodal	`DD(0.001)`	`0.10%`	`BFloat16` 0.60% · `GK(2000)` 17.0% at p99.9

> `DD(0.001)` wins every column at ≤0.10% — its relative-error guarantee is distribution-agnostic, so one parameter handles heavy tails, discrete spikes, and multimodal data without retuning.

SYNTHESIS

Trade-offs at a glance

Five tradeoff axes, six contenders. Higher is better, 1–10.



CATEGORY WINNERS

Best overall accuracy	`DD(0.001)`
Best tuning-free accuracy	`quantileBFloat16`
Best size-to-accuracy	`DD(0.01)`
Smallest accurate state	`DD(0.04)`
Best for timing columns	`quantileTiming`
Most merge-stable	`BFloat16 / DD / Timing`
Avoid for tails	`GK (all params)`
Avoid for merges	`KLL (k < 1600)`

BEHIND THE NUMBERS

A few other insights

How each algorithm's design shows up in the data.

Small p50 errors are tiny absolute diffs

`query_log` medians are small integers. One unit of rounding looks like 12.5%, but is only 1 ms in absolute value.

`TDigest` fails on discrete data

Centroids cluster on `read_rows` mass points, so the interpolation breaks. 9.99% value error at p90.

`GK` overflows at p99.9

Rank-error budget runs out; `GK` returns the dataset max instead of the quantile.

`DD` trades accuracy for size, predictably

40× tighter error costs only 18× more state. Insert and query time unchanged.

`DDSketch`'s relative value error guarantee holds in practice

Observed worst-case matches the theoretical bound to hundredths of a percent at every level.

`KLL` fails at tails and through merges

Random compaction is non-deterministic; smaller K shift 50–436 pp through merges.

`BFloat16` precision is intrinsic

7-bit mantissa = ~0.78% relative step. No parameter tuning needed.

Recommendation: `quantileDD(0.01)` on `query_log` dataset

Guaranteed ≤1% error at every level, 5.3 KB state, deterministic merges.

IN CONCLUSION

Takeaways

Six things worth carrying forward from this evaluation.

1 **Know the difference between rank and value error**

The question dictates the family. "Top 1% by rank" → GK/KLL. "p99 in ms" → DD, BFloat16, Exponential.

2 **Know your data shape**

Heavy-tailed, discrete, multimodal distributions break a different assumption. Profile before you pick.

3 **Prometheus Exponential histograms set a high bar**

Schema-as-precision is elegant and bounded, other datastores can learn from this design.

4 `DDSketch` **is an attractive quantile algorithm for observability**

Relative value error guarantee, distribution-agnostic, bit-identical through merges.

5 `quantileBFloat16` **is the surprise of the benchmark**

≤0.67% worst, no parameters, bit-identical merges, ~1.0× query speed. Use it more aggressively.

Thank you.

Questions?