4.1 — Advanced Topics & Technology

Data Processing at Scale

2026-03-27

Data Processing at Scale — Day 4

Course website

https://vbergeron.github.io/data-processing-at-scale/

This presentation

https://vbergeron.github.io/data-processing-at-scale/4.1-advanced-topics.pdf

Bloom Filters

The membership problem

Given a large set S, answer **“is x in S?”** — as fast and as cheaply as possible.

The naive answer

- linear search : compute is $O(|S|)$, space is $O(|S|)$
- binary search (hash set) : compute is $O(\log |S|)$, space is $O(|S|)$.

What if we trade accuracy for space?

Bloom filters

A probabilistic data structure that answers the membership question.

- $O(1)$ in time
- $O(K < N)$ (**sub-linear**) space.

Bloom filters can return **false positives** (claim an element is present when it is not), but they **never return false negatives**.

The data structure

A Bloom filter is :

- a **bit array** B of m bits, initially all zero
- k independent hash functions $h_1, \ldots, h_k$ each mapping an element to $[0, m)$.

Efficient hashes computation

Small interpolation trick, given only two hash functions h_A and h_B:

$$h_{i(x)} = \left(h_{A(x)} + i * h_{B(x)}\right) \bmod m$$

Insert x

Insert is : foreach i in $[0, k), B\left[h_{i(x)}\right] := 1$

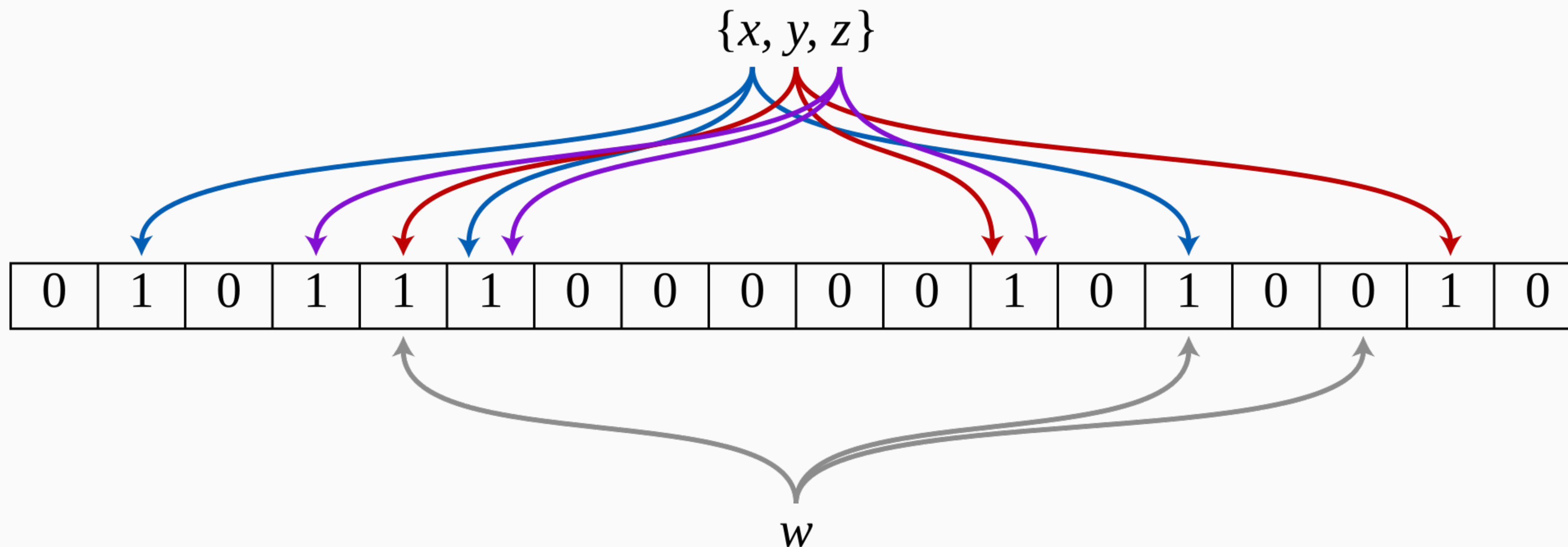
Query x

Query is : forall i in $[0, k), B\left[h_{i(x)}\right] = 1$

False positives

A false positive occurs when all k positions happen to be set by **other** elements.

There are no false negatives because insertion always sets all k bits.



Other nice features

- **Deletion is not supported** — clearing a bit might unset it for another element that shares that position.
- if m and h_i are the same for both filters
  - $B_1 \cup B_2$ is given by bitwise OR
  - $B_1 \cap B_2$ is given by bitwise AND

Tuning: the false positive rate

Given n elements inserted into m bits with k hash functions, the false positive probability is approximately:

$$p \approx \left(1 - e^{-k\frac{n}{m}}\right)^k$$

For a target false positive rate p and expected n elements, the optimal parameters are:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

$$k = \left(\frac{m}{n}\right) \ln 2$$

Example: 1 million elements at 1% false positive rate $\longrightarrow m \approx 9.6$ million bits (1.2 MB), $k = 7$ hash functions.

Where Bloom filters appear in the stack

System	Use
Apache Cassandra	Each SSTable has a Bloom filter — skip reading the file entirely if the key is definitely absent
Apache Parquet	Row group Bloom filters — added in 2.0; skip entire row groups in predicate pushdown
ClickHouse	`bloom_filter` skip index — prune granules that cannot match a `WHERE` condition

Variants

Counting Bloom filter — replace each bit with a small counter. Increment on insert, decrement on delete. Enables deletion at the cost of $4\times$ memory. Risk of counter overflow.

Scalable Bloom filter — chain multiple filters of increasing size. Start small; when the current filter would exceed the target false positive rate, add a new layer. Total memory grows logarithmically.

Cuckoo filter — stores fingerprints in a cuckoo hash table. Supports deletion natively, better cache efficiency, lower false positive rate at the same memory. Preferred over counting Bloom in modern systems.

Blocked Bloom filter — aligns bit positions to cache lines. All k bits for one element land in the same cache line $\longrightarrow$ 1 cache miss per lookup instead of k. Used in Apache Arrow and DuckDB.

More Probabilistic Data Structures

HyperLogLog — the problem

`COUNT(DISTINCT user_id)` over a billion rows requires storing all user IDs to compare them — or does it?

Exact distinct counting is a **streaming hardness result**: any exact algorithm requires $\Omega(n)$ bits in the worst case. HyperLogLog breaks this by accepting a small, bounded error.

The deal: 12 KB of memory, any cardinality, 1–2% relative error. The same 12 KB whether you have 10 000 or 10 billion distinct elements.

HyperLogLog — mechanics

Intuition: hash each element uniformly. The probability of a hash starting with k leading zeros is 2^{-k}. If the maximum number of leading zeros observed is k, you have likely seen around 2^k distinct elements.

In practice: one register is too noisy. HyperLogLog splits elements into $m = 2^b$ buckets (using the first b bits of the hash), keeps one max-leading-zeros register per bucket, and combines them with a **harmonic mean** to correct for bias.

Error: $\approx \frac{1.04}{\sqrt{m}}$ — with $m = 2^{14} = 16384$ registers, error drops below 1%.

Merging: two HyperLogLog sketches merge by taking the element-wise max of their registers — this is exact, no loss.

HyperLogLog — in the stack

System	Usage
ClickHouse	`uniq()` and `uniqHLL12()` — default for `COUNT(DISTINCT ...)`
Redis	`PFADD` / `PFCOUNT` — a HLL per key, O(1) add and count
BigQuery	`APPROX_COUNT_DISTINCT()` — HLL++ variant
Apache Spark	`approx_count_distinct()` with configurable relative SD
Druid	HLL sketch columns for rollup aggregation at ingest time

When you see `uniq(user_id)` in a ClickHouse dashboard query, the database is running HyperLogLog — not a hash set.

Count-Min Sketch — the problem

Given a high-velocity stream of events (clicks, packets, transactions), answer: **"how many times has element x appeared?"** — without storing the full stream.

Exact frequency counting requires one counter per distinct element

Count-Min Sketch — the problem

$O(|\Sigma|)$ space. For IP addresses or URLs, that is gigabytes.

Count-Min Sketch answers frequency queries in **fixed space** with a one-sided error: it may **overestimate**, but never underestimates. A count of 0 means the element was truly never seen.

Count-Min Sketch — mechanics

Structure: a $d \times w$ matrix of counters, initialized to zero. Choose d independent hash functions $h_1, \ldots, h_d$, each mapping an element to $[0, w)$.

Update element x: for each row i, increment `table[i][h_i(x)]`.

Query element x: return $\min_i$ `table[i][h_i(x)]`.

Why min? Hash collisions only inflate counters — they never decrease them. Taking the minimum across d independent rows isolates the least-collided estimate.

Error bound: with probability $1 - \delta$, the estimate is within $\varepsilon \cdot \|f\|_1$ of the true count, where $w = \lceil \frac{e}{\varepsilon} \rceil$ and $d = \lceil \ln\left(\frac{1}{\delta}\right) \rceil$.

Count-Min Sketch — in the stack

System	Usage
Apache Flink	Heavy-hitter detection in `TopNFunction` and stream sampling
Network monitoring	Per-flow packet counting in switches — hardware implementations
Databases	Query optimiser cardinality estimation for join ordering
Content delivery	Tracking hot keys / trending content without storing full logs
ClickHouse	`topK(N)(x)` uses the Space-Saving algorithm — a CMS variant

The key asymmetry: you trade **which** elements you count precisely (all of them) for **which** queries you answer precisely (only the heavy hitters, which are what you care about anyway).

Reservoir Sampling — the problem

Given a stream of unknown length N, select a **uniform random sample** of exactly k elements — without knowing N in advance, and without storing the full stream.

Naïve approach: buffer everything, sample at the end. Requires $O(N)$ memory — infeasible for an unbounded stream.

Reservoir sampling solves this in $O(k)$ memory with a provably uniform sample, processing each element exactly once.

Reservoir Sampling — mechanics

Algorithm R (Vitter, 1985):

1. Fill the reservoir with the first k elements.
2. For each subsequent element at position $i > k$: generate $j =$ random integer in $[1, i]$. If $j \leq k$, replace reservoir$[j]$ with the current element.

Why is this uniform? After i elements, each has probability $\frac{k}{i}$ of being in the reservoir — provable by induction. The final sample is exactly uniform over all N elements.

Distributed variant (reservoir merging): each partition independently samples k elements with weights. Partitions are merged by weighted sampling — enables parallel reservoir sampling over a Spark or Flink dataset.

Reservoir Sampling — in the stack

System	Usage
ClickHouse	`quantile(p)(x)` uses reservoir sampling for percentile estimation
Apache Spark	`DataFrame.sample()` and `sampleBy()` use reservoir variants
Apache Flink	Stream sampling operators; `approxQuantile` in Table API
ML pipelines	Uniform data sampling for training sets from large feature stores
A/B testing	Reservoir ensures uniform user assignment when population size is unknown

Reservoir sampling is the streaming equivalent of `ORDER BY random() LIMIT k` — except it runs in $O(k)$ memory and one pass, while `ORDER BY random()` requires materialising the full dataset.

Differential Dataflow

The recomputation problem

A `GROUP BY` query over 1 TB takes 10 seconds.
One new row arrives.
You rerun the query: 10 seconds *again*.

The question: can we process **only what changed** and produce a correct updated result?

Differential dataflow

Naïve incremental maintenance works for simple cases — a running `SUM` or `COUNT` is trivially updatable. But `JOIN`, `GROUP BY` with retraction, `DISTINCT`, and `TOP-K` are not trivially updatable.

Differential dataflow is a framework that handles incremental updates correctly.

Changes as collections of deltas

Every input and output is a **multiset of changes** ΔA: pairs (x, d) where $d \in \mathbb{Z}$ is a multiplicity.

$$\Delta A = \{(x_1, d_1), (x_2, d_2), \dots\}$$

- $(x, +1)$ — row x was inserted
- $(x, -1)$ — row x was deleted
- An update is $(x_{\text{old}}, -1)$ followed by $(x_{\text{new}}, +1)$

Changes as collections of deltas

Compaction: $(x, +1)$ and $(x, -1)$ cancel — a multiplicity of 0 means the record is absent.

The full collection A at any time is the accumulated sum of all deltas:

$$A = \sum_t \Delta A_t$$

Incremental monotonic operators

Monotonic operators propagate diffs unchanged — they never retract output.

- `map f` — $\Delta(f(A)) = \{(x, d) \in \Delta A \Rightarrow (f(x), d)\}$
- `filter p` — $\Delta\left(\sigma_p(A)\right) = \{(x, d) \in \Delta A \land p(x) \Rightarrow (x, d)\}$
- `union` — $\Delta(A \cup B) = \Delta A + \Delta B$

Incremental non-monotonic operators

Non-monotonic operators may produce retractions — a new input can invalidate a previous output.

- `join` — using the **delta rule**

$$\Delta(A \bowtie B) = (\Delta A \bowtie B) \cup (A \bowtie \Delta B)$$

Incremental non-monotonic operators — `group_by`

Model the aggregate as a function $\kappa(S, x) \to S$ where S is the **aggregation state** and x is a new row. The operator maintains one state S_k per group key k.

For **combinable** aggregates (`SUM`, `COUNT`), κ simply adds or subtracts from a scalar — efficient.
For `AVG`, S must track **(sum, count)** separately: $\kappa((s, n), (x, +1)) = (s + x, n + 1)$.
`MIN` and `MAX` require the full multiset — a single retraction of the minimum forces a scan of the remaining state.

Incremental non-monotonic operators — `group_by`

When a delta row (x, d) arrives in group k:

1. Look up the current state S_k and derive the current output $\mathrm{out}(S_k)$
2. Update: $S_k \leftarrow \kappa(S_k, (x, d))$
3. Emit $\left(k, \mathrm{out}\left(S_k^{\mathrm{old}}\right), -1\right)$ — retract old output
4. Emit $(k, \mathrm{out}(S_k^{\mathrm{new}}), +1)$ — assert new output

Incremental non-monotonic operators — `distinct`

`distinct` keeps only rows with positive total multiplicity. A new `+1` for a row already present does **not** produce output — the output was already asserted. A `-1` for a row with multiplicity 2 does **not** retract the output — the row is still present.

Only when multiplicity crosses zero does the output change:

- Multiplicity $0 \to 1$: emit $(x, +1)$
- Multiplicity $1 \to 0$: emit $(x, -1)$

This requires maintaining the **full multiplicity map** — not just presence. It cannot be implemented without state proportional to the number of distinct elements.

Monotonicity and CALM

Monotonic programs only accumulate facts — they never retract. Given more input, they produce **more** output, never less.

A monotonic dataflow can run **without coordination**: each operator processes diffs as they arrive, in any order, and the result converges to the correct answer. No locks, no barriers, no two-phase commit.

Non-monotonic operators break this: a `distinct` or `top-K` may need to retract a previously emitted output when new data arrives. Correct retraction requires knowing **when** a delta is complete — which requires coordination.

Real systems

System	Connection
Materialize	Database built on differential dataflow; SQL views stay live and correct as data arrives
Flink (Table API)	Emits `+I` / `-D` / `-U` / `+U` changelog records — the same `(record, diff)` model
ClickHouse MVs	Trigger-on-insert, no retraction — correct only for monotonic aggregates (`SUM`, `COUNT`); `AVG` requires `AggregatingMergeTree`
Apache Spark	Structured Streaming with `outputMode("update")` emits deltas; complete mode recomputes — the non-incremental fallback

Wrap-Up — the shared thesis

Bloom filters, HyperLogLog, Count-Min Sketch, and Reservoir Sampling all make the same trade: **give up exactness to gain space and speed**, with a bounded, predictable error.

Differential dataflow makes a different trade: **give up simplicity to gain incrementality**, processing only what changed while remaining exactly correct.

Both directions are responses to the same pressure: data volumes that make naive approaches infeasible. Knowing when to approximate and when to maintain exactly — and **which systems implement which model** — is the engineering judgment this session is about.

Apache Druid & Course Conclusion

druid

Apache Druid — when massive data meets massive demand

Druid is a real-time analytics database designed for the intersection of two hard problems: **ingesting high-velocity streams** and **answering sub-second queries** over petabyte-scale historical data simultaneously.

Apache Druid — when massive data meets massive demand

Course concept	Druid mechanism
Columnar storage (Day 2)	Per-segment column files, dictionary encoding
Parquet / Kafka ingestion (Days 2–3)	Streaming ingestion via Kafka; batch via S3 / HDFS
Vectorized execution (Day 3)	SIMD-friendly column scans over compressed segments
Pre-aggregation (Day 3)	Rollup at ingest time — rows collapsed into aggregated cubes
Probabilistic DS (Day 4)	HyperLogLog and quantile sketches as native column types
Incremental computation (Day 4)	Real-time segments merged continuously into historical ones

Apache Druid — architecture

Druid separates **ingestion**, **storage**, and **query** into independent services that scale independently.

- **Overlord + MiddleManagers** — coordinate and execute ingestion tasks (streaming or batch)
- **Historical nodes** — serve immutable, fully indexed segments from deep storage (S3 / GCS)
- **Brokers** — receive queries, fan out to Historicals and Realtimes, merge results
- **Real-time tasks** — ingest live Kafka streams into **in-memory segments**; publish to deep storage on completion
- **Deep storage** — S3 or HDFS; the source of truth; Historical nodes load segments from it on demand

The key property: **real-time and historical data are queryable simultaneously**. A query spanning today (in memory) and last year (on S3) returns a unified result with sub-second latency on the recent portion.

Apache Druid — rollup and sketches

Druid's most aggressive optimisation is **rollup**: at ingest time, rows with the same dimensions and timestamp granularity are collapsed into a single aggregated row.

```
{ "ts": "2024-01-01T10:00:00", "country": "FR", "page": "/home",
  "views": 1, "uniq_users": "HLL_sketch" }
```

A billion raw events become millions of pre-aggregated rows. Queries that would scan 1 TB scan 10 GB instead.

Sketches (`HyperLogLog`, `quantilesDoublesSketch` from the Apache DataSketches library) are stored **as column values** — they merge exactly at query time. The incremental `kappa` model you saw in the differential dataflow section is what Druid implements natively, at ingest speed.

Course conclusion — three principles

Every system in this course, from SQLite to Druid, is an expression of three engineering principles:

Think ahead **Algorithms matter** **Precompute aggressively**