Technical University of Crete
School of Electrical & Computer Engineering

Diploma Thesis

A Toolkit for Scalable Preprocessing and Neural Learning with Tensorflow and Dask

This Thesis Was Submitted in Partial Fulfillment of the Requirements for the Diploma in Electrical and Computer Engineering

Author:
Michael Kratimenos

Thesis Committee:

Prof. Nikolaos Giatrakos, Supervisor
Prof. Antonios Deligiannakis
Prof. Georgios Chalkiadakis

June 2025

Acknowledgements

Upon the completion of my diploma thesis, I would like to express my sincere gratitude to everyone who contributed to its development.

I am especially thankful to Professor Nikolaos Giatrakos for assigning and selecting the topic, as well as for his invaluable support, guidance, and supervision throughout the implementation and writing of this thesis.

I would also like to thank Professor Antonios Deligiannakis and Professor Georgios Chalkiadakis for dedicating their time to serve on my examination committee and for evaluating my diploma thesis.

My thanks go to postgraduate student Errikos Streviniotis as well, who contributed to the initial development of the SuBiTO Framework Python-code, which served as the foundation for this thesis.

Finally, I would like to thank my parents, my sister, and my beloved Valia, to whom I dedicate this thesis. Their unwavering support, both emotional and practical, throughout my studies and my life has been truly invaluable.

Abstract

In modern data-intensive applications, particularly those involving neural learning, the volume and velocity of incoming data pose significant challenges for real-time preprocessing and analysis. This thesis addresses the absence of Python-based solutions for stream summarization by introducing a parallel, scalable toolkit for data synopses, implemented using `Apache Dask`. Unlike existing synopses data engines, our approach is seamlessly integrated into Python ecosystems and is directly compatible with `TensorFlow`-based learning pipelines. We present an aggregation of probabilistic data structures, such as `Bloom Filters`, `HyperLogLog` and `PrioritySampler`, which maintain summaries of large data streams using sublinear memory. These structures support essential operations like `add`, `merge`, and `estimate` and are specifically implemented to allow efficient parallel computation via `Dask`. The toolkit is further embedded into SuBiTO, a Bayesian optimization framework for scalable learning, in order to optimize its performance. This optimization is shown in the experimental evaluations, where our engine significantly improves the runtime of data preprocessing tasks in distributed environments, by accelerating the synopses maintenance. Then, only concise data summaries are fed into neural learning pipelines to achieve appropriate balance between training accuracy and training speed. This work provides both a foundational toolkit and an integrated path between the neural learning pipeline and the preprocessing summarization step, offering a strong basis for future work in scalable, Python-based data summarization systems.

Contents

List of Tables

List of Figures

List of Coding Examples

Chapter 1

Introduction

In modern machine learning workflows, particularly those involving neural learning with frameworks such as `TensorFlow` [52], a critical bottleneck arises in the **preprocessing stage** - especially when working with massive data streams that exceed memory constraints. Although `TensorFlow` provides powerful mechanisms for model training, it does not have tools for scalable data summarization. This gap does not let the neural learning pipelines to integrate seamlessly with the data preprocessing summarization step.

Existing solutions for data summarization, often referred to as **Stream Synopses**, offer probabilistic methods for estimating **frequency**, **cardinality**, and **quantiles** over data streams in sublinear space. However, current state-of-the-art implementations of these methods are typically `Java`-based (e.g., `Apache DataSketches` [3], `Stream-lib` [1]), making their integration to neural learning pipelines more difficult and cumbersome. As a result, this incompatibility complicates workflows and restricts the ability to summarize big data as a preprocessing step (before the training procedure, within a Python-based pipeline) to achieve appropriate balance between training accuracy and speed of online neural learning pipelines over streaming data that continuously flow and update the neural model.

1.1 Thesis Contribution

To address this gap, this thesis introduces what is, to our knowledge, the first parallel, Python-based synopses data engine built upon `Apache Dask` [11]. Our toolkit is designed to serve two primary purposes:

- **Generic stream summarization**, which is where stream synopses are held by using scalable and memory-efficient algorithms implemented in Python.
- **Enhancement of the state-of-the-art SuBiTO framework** [50], for scalable learning over streaming data, through optimized preprocessing that leverages parallel stream computation and efficient stream summarization.

The integration of `Dask` is pivotal: it not only enables parallel execution across multi-core and distributed environments, but it also aligns perfectly with `TensorFlow`'s ability to handle `Dask`-based data structures. This ability is exactly what allows the neural learning pipeline to integrate seamlessly with the preprocessing summarization step. In contrast, common frameworks like `Apache Spark` [8] lack native compatibility with `TensorFlow`.

By incorporating stream synopsis algorithms into `Dask`-based structures, we achieve scalable and efficient data preprocessing that achieves the same statistical accuracy as `Numpy`. Furthermore, embedding this toolkit into **SuBiTO** enables improved performance during the data preprocessing step, as synopses can be maintained and queried in parallel, offering a major speedup without sacrificing accuracy.

1.2 Thesis Outline

This Thesis consists of 8 chapters, excluding this one, which are:

- **Chapter 2** introduces the fundamental stream synopsis algorithms that form the base of **our toolkit**, explaining their **theoretical** foundations and **practical** relevance.
- **Chapter 3** surveys related work, analyzing some existing systems and frameworks in the field of **stream summarization** and **scalable data processing**.
- **Chapter 4** explains `Apache Dask` [11], the core **parallel computing** framework used in our architecture.
- **Chapter 5** focuses on `TensorFlow` [52] and how it interacts with our **preprocessing** pipeline.
- **Chapter 6** details the design and implementation of our Python-based **Toolkit**, including its architecture and how it achieves parallelism.
- **Chapter 7** describes the integration with the **SuBiTO framework** [50] and how **our toolkit** enhances its performance.
- **Chapter 8** provides an **experimental evaluation** of **our toolkit** across various scenarios.
- **Chapter 9** concludes the thesis and suggests possible directions for future research and improvements.

Chapter 2

Stream Synopses

In this section we are going to present **Stream Synopses** that are of interest for our toolkit and we are going to explain the way they are build and queried. Each of these synopses corresponds to a class as we are going to see in our implementation in **Chapter 6**. For each synopsis, we are going to refer to `add, merge, estimate` methods which will later be exemplified in our implementation for distributive updating (add), merging parallel, partial synopses (merge) and querying (estimate) them. We also provide hints to methods used in our implementation in **Chapter 6**, to bind the theoretical framework and description of the various synopses, with their actual implementation and parallel computation in **Chapter 6**.

2.1 Bloom Filter

A **Bloom filter** [35] is a space-efficient probabilistic data structure representation of a stream of n elements from a universe U, used for approximate set membership queries. It allows one to test whether an element may be present in a set, with a guarantee of no false negatives and a controllable probability of false positives.

This implementation of a Bloom filter uses a boolean array of length m as the underlying storage and a set of k hash functions simulated via double hashing with `MurmurHash` [40]. All bits are initially unset (`False`). To insert an element (`add`), it is encoded into a byte format and hashed k times to determine k positions in the bit array. These positions are then set to `True`.

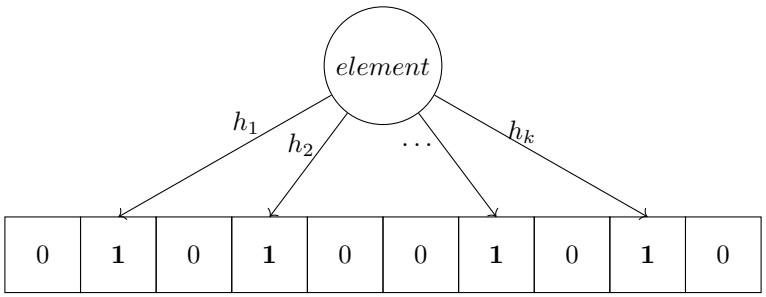

Figure 2.1: During insertion in a Bloom filter, an item is hashed using k independent hash functions, each producing an index in the bit vector. The bits at all k positions are then set to 1, marking the item as present.

Membership queries are handled by checking whether all k corresponding positions are set (`estimate`). If any of these bits is `False`, the element is definitively not present. If all are `True`, the element is assumed to be present, with a small probability of a false positive due to collisions caused by other inserted elements.

Bloom filters can be merged (`merge`) using bitwise `OR`, allowing for union of sets. The implementation includes a method to construct a saturated Bloom filter where all bits are set. This special case returns a positive result for any membership query and is primarily used for testing.

The probability of false positives is $(1 - e^{(-kn/m)})^k$, where n are the elements of the stream and m is the length of the boolean array. For given n and Bloom filter length, the false positive probability can be minimized by optimizing the ratio between true bits and Bloom filter length.

In summary, this Bloom filter implementation offers efficient insertion and membership testing with low memory overhead and provides support for filter union.

2.2 Count Min Sketch

The **CountMinSketch** [41] data structure is a probabilistic technique for summarizing data streams in sublinear space while supporting approximate query answering. This data structure is especially useful in scenarios where exact computation is unachievable due to memory constraints or high data volume, such as real-time analytics or network traffic monitoring.

At its core, the CM sketch maintains a two-dimensional array of counters (of depth d and width w), and uses d pairwise-independent hash functions to map stream items to specific counters across rows. When an item is added or updated in the stream, its count is incremented at each corresponding hashed position. To estimate the frequency of an item, the sketch returns the minimum of its associated counters, thereby reducing overestimation caused by hash collisions.

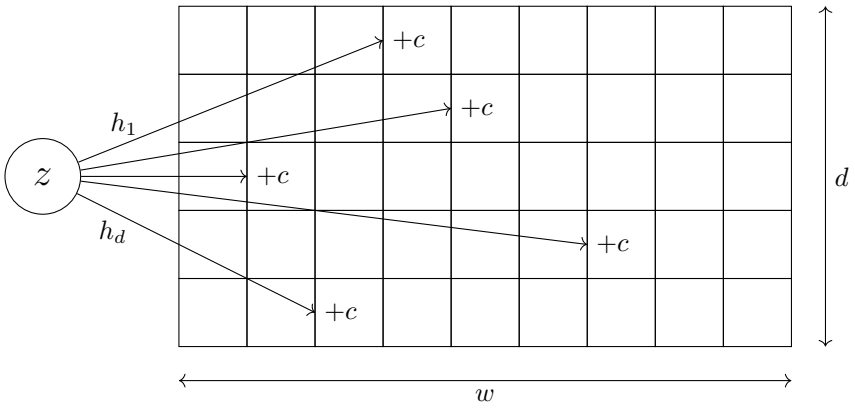

Figure 2.2: Each item z is mapped to one cell pointed by $h_i, i \in \{1, \ldots, d\}$ (hash function) in each row of the array of counts. In more detail, when an update of c to item z arrives, c is added to each of these cells [41].

The synopsis supports:

- **Initialization** based on either user-defined parameters (`depth`, `width`, and `seed`) or accuracy/-confidence targets (ε,δ), where in the case of (ε, δ), depth and width are defined by:

$$\mathrm{d} = \left\lceil \frac{-\log(1-\delta)}{\log 2} \right\rceil, \quad \mathrm{w} = \left\lceil \frac{2}{\varepsilon} \right\rceil$$

- **Updating** the sketch with new elements or values using `add()`.
- Frequency estimation is performed by **querying** the sketch for the minimum counter value across all rows using `estimate()`. Specifically, to estimate the frequency of an item z, the sketch returns:

$$f(z) = \min_{1 \le i \le d} \mathrm{CMS}[i, h_i(z)],$$

  providing guarantees that the returned estimate is never less than the true count.
- **Merging** multiple sketches with compatible parameters into a single sketch using `merge()`, allowing scalable distributed data processing.

- **Hashing functions** that ensure uniform distribution and low collision probability using modular arithmetic and `MurmurHash`.

In practice, the Count-Min sketch supports key queries like point estimation, range sums, and inner product estimation in time and space complexities that scale with $\mathcal{O}(\frac{1}{\varepsilon}\log(\frac{1}{\delta}))$. It is particularly well-suited for detecting frequent items (heavy hitters), estimating join sizes in databases, and computing approximate quantiles—all with strong theoretical guarantees and efficient implementation.

2.3 Conservative Add Sketch

The **ConservativeAddSketch** class is a variant of the standard Count-Min sketch that implements a technique known as *conservative updating* or *conservative add*. This approach aims to reduce overestimation errors inherent in the original Count-Min sketch, offering higher accuracy at a modest computational cost.

Unlike the traditional method that increments all counters corresponding to a given key by a specified amount, the conservative update strategy only increments those counters whose current value equals the minimum among all corresponding hash buckets. This ensures that inflated counts, caused by hash collisions with frequent items, are not unnecessarily propagated across all rows. As a result, this technique reduces the accumulated error in frequency estimates, especially for low-frequency items.

Let:

- d = number of hash functions (depth),
- w = width of each hash array (number of columns),
- $h_i(k)$ = the hash function for the i-th row applied to key k, where $0 < i \leq d$,
- $C \in \mathbb{N}^{d \times w}$ = the 2D sketch table (the counter matrix),
- $v \in \mathbb{N}$ = the value (increment amount),
- $k \in \mathbb{N}$ = the key index to be updated.

Step 1: Hash Bucket Indices

$$b_i = h_i(k) \quad \text{for } i = 1, \ldots, d$$

Step 2: Minimum Value Among Corresponding Buckets

$$m = \min_{0 < i \leq d} C[i, b_i]$$

Step 3: Conservative Update Rule

$$C[i, b_i] = \max(C[i, b_i], m + v) \quad \text{for each } i = 1, \ldots, d$$

Interpretation

In the implementation, the `add()` method updates only the counters whose current values match the minimum across all hash functions for a key. These selected counters are then set to $m + v$, reducing unnecessary increments caused by hash collisions and improving accuracy for low-frequency items.

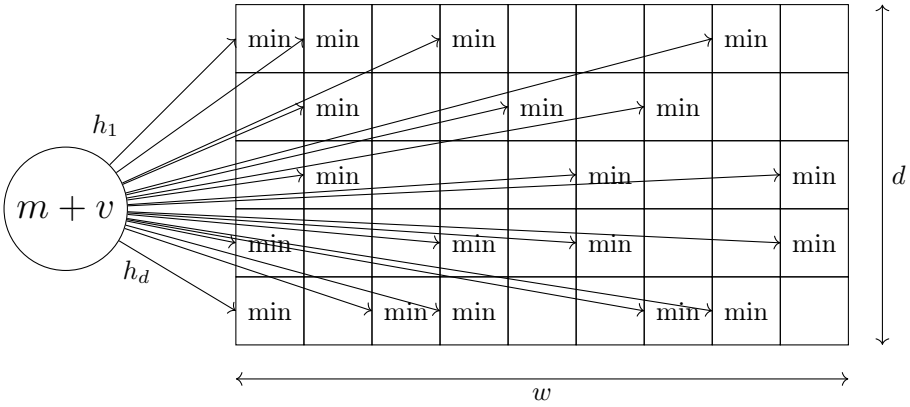

Figure 2.3: Conservative Add Sketch during update procedure. Cells with minimum values are updated to $m + v$ using hash functions $h_i, \quad i = 1, \ldots, d$.

This variant is particularly useful in applications where precision is critical, and the marginal increase in computational cost is acceptable. Although its accuracy gains are empirically significant, they are analytically less well-defined, making it a practical yet heuristic enhancement over the standard Count-Min sketch.

2.4 Partial Discrete Fourier Transform (DFT) Accumulator

The **Partial DFT Accumulator** [25] is designed to accumulate Discrete Fourier Transforms (DFTs) of multiple equal-length chunks of a signal. It leverages a technique based on a fundamental property of the Discrete Fourier Transform (DFT): the linearity property. This property states that the DFT of a sum of signals equals the sum of their individual DFTs, provided all signals are of the same length. This enables efficient computation of the DFT of the sum of those chunks, without having to compute the DFT of the entire signal at once.

2.4.1 Discrete Fourier Transform (DFT)

The **Discrete Fourier Transform (DFT)** [24] is a fundamental tool in digital signal processing that transforms a finite sequence of equally spaced samples of a signal from the time domain into the frequency domain. This transformation allows for the analysis of the signal's frequency components, which is essential in various applications such as signal analysis, filtering, and compression.

Mathematically, for a sequence $x[n]$ of length N, the DFT is defined as:

$$X[k] = DFT(x[n]) = \sum_{n=0}^{N-1} x[n] * e^{-j\frac{2\pi}{N}kn}, \quad k = 0, 1, \ldots, N-1$$

Here, $X[k]$ represents the frequency domain representation of the signal, with each value corresponding to a specific frequency component.

The inverse DFT (**IDFT**) allows for the reconstruction of the original time-domain signal from its frequency components:

$$x[n] = IDFT(X[k]) = \frac{1}{N} \sum_{k=0}^{N-1} X[k] * e^{j\frac{2\pi}{N}kn}, \quad n = 0, 1, \ldots, N-1$$

This bidirectional transformation between time and frequency domains is very important in analyzing and processing discrete signals.

2.4.2 Partial DFT

Due to the **Linear property** [25] it holds that: If each signal chunk x_i has the same length N, then:

$$DFT(\sum_i x_i) = \sum_i DFT(x_i) \quad (1)$$

With the use of identity (1) the following process can be done:

- Computation of the DFT of chunks in parallel
- Accumulation of these DFTs
- Reconstruction of the total DFT of the sum of chunks

Core Methods

- `add()`: Computes the DFT of a chunk and adds it to the internal sum (`dft_sum`). It also checks that all chunks are of the same length.
- `merge()`: Merges other PartialDFTAccumulator instances by summing their `dft_sum` values. Verifies that all accumulators have DFTs of the same length.
- `estimate()`: Returns the summed DFT.

2.5 Weighted Priority Sampler

The **Weighted Priority Sampler (WPS)** [60] is a streaming sampling algorithm designed to efficiently maintain a representative subset of items from a continuous data stream. It keeps the k items with the smallest priorities (implemented with a min-heap), where each item's priority is defined as:

$$priority = \frac{-log(U)}{weight} \quad (2)$$

Here, U is a uniformly distributed random variable in the interval $(0, 1)$, and $weight$ is a user-defined importance score for each item. This formulation ensures that higher-weight items have a higher chance of being included in the sample, while also introducing randomness to prevent bias.

Step 0: Original Values

Index	Value	Weight	U
1	3.2	1.0	0.92
2	5.1	2.5	0.35
3	0.9	1.8	0.72
4	4.0	1.2	0.50
5	2.3	0.9	0.60
6	6.7	1.0	0.95
7	1.5	2.0	0.10
8	3.8	1.5	0.55
9	4.9	0.8	0.20
10	2.6	1.1	0.65

$priority = \frac{-log(U)}{weight}$ →

Step 1: Compute Priorities

Value	Weight	Priority
5.1	2.5	0.419
1.5	2.0	1.151
4.9	0.8	2.014
4.0	1.2	0.576
0.9	1.8	0.198
2.3	0.9	0.568
3.8	1.5	0.398
2.6	1.1	0.356
3.2	1.0	0.083
6.7	1.0	0.051

Step 2: Sort by Priority

Value	Priority
6.7	0.051
3.2	0.083
0.9	0.198
2.6	0.356
⋮	⋮

→

Step 3: Keep Top-k = 4 Items

Kept Value	Priority
6.7	0.051
3.2	0.083
0.9	0.198
2.6	0.356

Figure 2.4: An example of the Weighted Priority Sampling process for 10 numbers and for $k = 4$ sampled numbers: firstly, 10 random numbers are generated, then, the values are assigned weights and random uniform values, their priorities are computed and the top-4 items are selected.

Core Features

- Items are added in chunks with `add()`, enabling use in large-scale applications,
- Internally, the sampler maintains a min-heap of size k to efficiently track the top k smallest priority items (implemented by storing priorities).
- The sampler can estimate (`estimate()`) the weighted mean of the sampled values using the stored weights and values.

2.6 Priority Sampler

The **Priority Sampler (PS)** [56] is a streaming, reservoir-style sampling algorithm that selects a fixed number of items (k) from a data stream uniformly at random. It inherits from the Weighted Priority Sampler (WPS) but operates without explicit weights, relying instead on random priorities to drive selection.

Each item is assigned a random priority drawn from a uniform distribution $[0, 1)$ and the sampler retains only the k items with the smallest priorities (simulated via a min-heap). This approach enables unbiased, uniform sampling.

Step 0: Original values

Index	Value
1	3.2
2	5.1
3	0.9
4	4.0
5	2.3
6	6.7
7	1.5
8	3.8
9	4.9
10	2.6

$\xrightarrow{\Pr = U \sim \mathcal{U}(0,1)}$

Step 1: Assign Random Priorities

Value	Priority (Pr)
5.1	0.419
1.5	0.151
4.9	0.014
4.0	0.476
0.9	0.798
2.3	0.468
3.8	0.598
2.6	0.656
3.2	0.883
6.7	0.951

Step 2: Sort by Priority

Value	Priority (Pr)
4.9	0.014
1.5	0.151
5.1	0.419
2.3	0.468
⋮	⋮

$\longrightarrow$

Step 3: Keep Top-k = 4 Items

Kept Value	Priority (Pr)
4.9	0.014
1.5	0.151
5.1	0.419
2.3	0.468

Figure 2.5: An example of the Priority Sampling process for 10 numbers and for $k = 4$ sampled numbers: firstly, 10 random numbers are generated, then, the values are assigned random uniform values, their priorities are computed and the top-4 items are selected.

Core Features

- Items can be added in batches using `add()`, with each addition evaluated based on its random priority,
- Internally, a min-heap tracks the top-k items with the smallest priority values (implemented by storing priorities),
- In our implementation, a `reservoir_priority_sampling()` method splits large datasets into chunks and applies sampling in parallel using Dask,
- The sampler supports estimation (`estimate()`) by returning the indices of sampled items.

Unlike WPS, which adjusts sampling probabilities using item weights, the Priority Sampler performs uniform random sampling, making it suitable for applications where all items are equally important or weights are not available.

2.7 Linear Counting

Linear Counting [51] is a probabilistic algorithm to estimate the number of distinct elements (cardinality) in a multiset that may include duplicates. Traditional approaches like sorting or exact hashing often incur high time or memory costs, typically scaling with $O(qlogq)$, where q is the total number of elements. Linear counting, in contrast, achieves linear time complexity $O(q)$ and maintains high estimation accuracy while using significantly less memory.

2.7.1 Algorithm explanation

- A bit array (**map**) (of size m bits) is initialized with zeros,

- Then, each addition (`add()`) in the dataset is hashed using a uniform hash function (e.g., `MurmurHash` in the implementation) and its result is mapped to a position in the array, which is decided by the hash function,
- After processing all items, the number of zeros (unmarked positions) (V_n) in the bit array is counted and the estimation (`estimate()`) of the distinct elements is produced using the following formula:

$$\hat{n} = -m * ln(V_n/m)$$

This formula derives from the probability that a given bit remains zero after n distinct values are hashed into a uniformly distributed space of size m.

The same formula to produce the final estimate is used in our implementation as well:

Coding Example 2.1: Linear Counting final estimation formula

```
1  return int(round(self.length * math.log(self.length / self.count)))
```

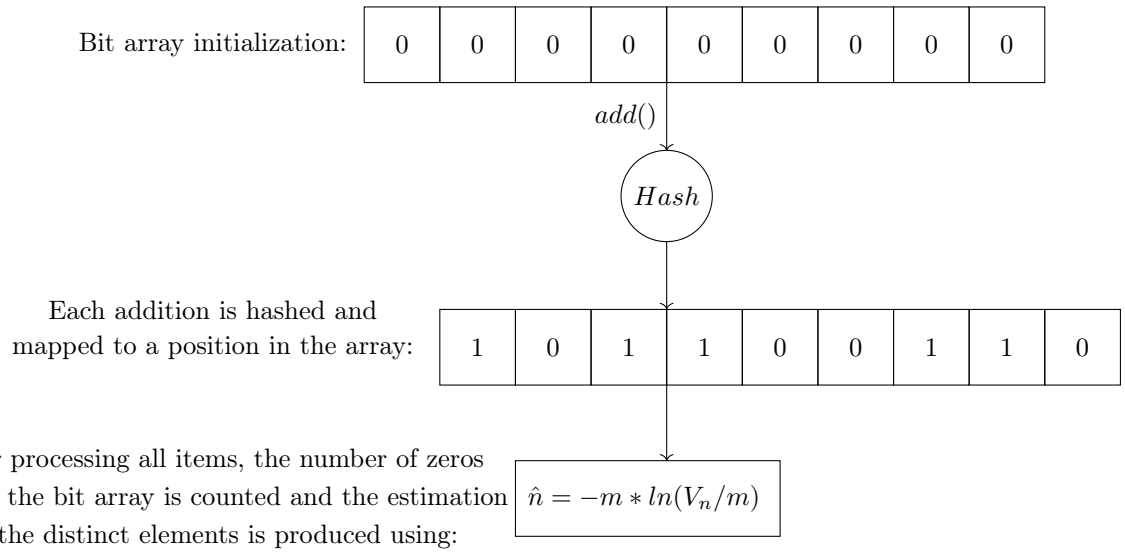

Figure 2.6: **Linear Counting** algorithm breakdown for a random element

2.7.2 Theoretical Foundations

Let $t = n/m$ be the load factor, representing the ratio of distinct items to the size of the bit array.

- Bias (relative error) is given by the following equation:

$$Bias(\hat{n}/n) = E[\hat{n}/n] - 1 = \frac{e^t - t - 1}{2n}$$

- Standard error is given by the following equation:

$$StdError(\hat{n}/n) = \frac{\sqrt{m(e^t - t - 1)}}{n}$$

  which approaches zero as $n \to \infty$, for a constant t.

These properties allow linear counting to achieve arbitrary accuracy by adjusting the size of the bit array m.

The algorithm remains accurate even with load factors greater than 1.0 (e.g. $t = 12$), allowing it to estimate distinct values in datasets with over 100 million tuples using only 10 million bits of main memory (~ 1.25 MB).

2.8 Log Log

The **LogLog** [27] algorithm is a probabilistic method to estimate cardinality (i.e., the number of distinct elements) in a multi-set using sublinear memory. It leverages the fact that the number of leading zeros in the binary representation of a hash-function's output can be used as a probabilistic indicator of element rarity.

The algorithm maintains a fixed size array M of length $m = 2^k$, where each entry stores the maximum number of leading zeros observed in hashed values mapped to that index. The index is derived from the first k bits of the hash; the remaining $32 - k$ bits determine the number of leading zeros.

For each element x, the algorithm:

- Hashes it to 32 bit integer: $h(x)$,
- Uses the first k bits to select index j,
- Computes $r = \rho(h(x))$, the position of the first 1 in the remaining $32 - k$ bits,
- Updates: $M[j] = \max(M[j], r)$.

Once all elements are processed, the estimated cardinality $\hat{n}$ is computed through:

$$\hat{n} = C_m \cdot 2^{\bar{R}}$$

where:

- $\bar{R} = \frac{1}{m} \sum_{j=0}^{m-1} M[j]$, is the average of the register values,
- C_m is a bias correction constant depending on m.

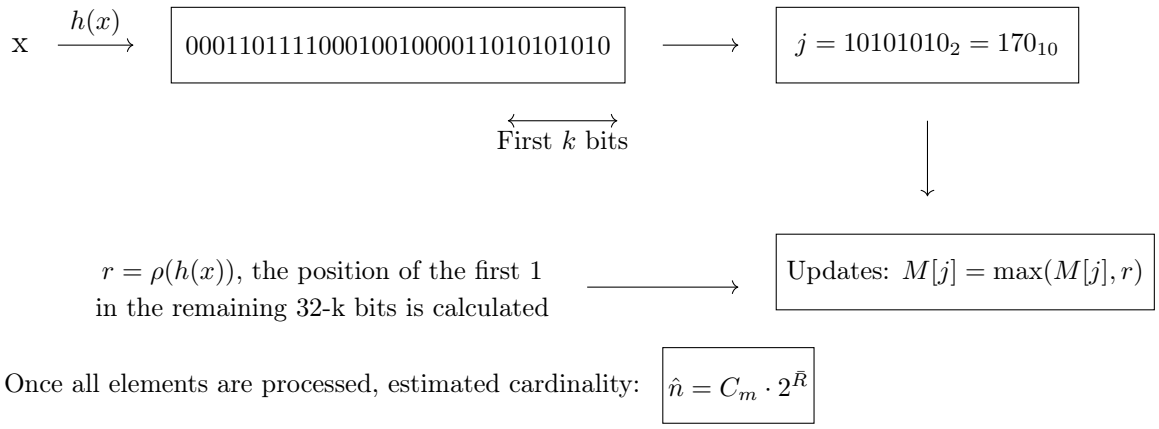

Figure 2.7: **LogLog** algorithm breakdown for a random element

In implementation, the LogLog algorithm:

- Includes a method `add()` to hash and incorporate new elements,
- Returns the final cardinality approximation using `estimate()`,
- Allows merging (`merge()`) multiple estimators via element-wise maxima:

$$M_{\text{merged}}[j] = \max(M_1[j], M_2[j], \dots)$$

Its time complexity is $\mathcal{O}(1)$ per element, with space complexity $\mathcal{O}(2^k)$, making it highly efficient for large datasets.

2.9 Adaptive Counting

Adaptive Counting [31] is an enhancement of the LogLog algorithm to estimate set cardinality. Like LogLog, it maintains a register array M of size $m = 2^k$, where each bucket stores the maximum number of leading zeros observed in hashed values.

The key innovation in Adaptive Counting is its ability to dynamically switch between two estimation strategies based on the proportion of empty buckets, improving accuracy for both small and large cardinalities without increasing memory usage.

Let:

- b_e be the number of empty buckets,
- $\beta = b_e/m$ the empty bucket ratio,
- $\beta_s = e^{-t_s} \approx 0.051$ the switching threshold, with $t_s \approx 2.89$ being the crossover point where linear counting becomes less accurate than LogLog.

The cardinality estimate $\hat{n}$ is:

$$\hat{n} = \begin{cases} \alpha_m * m^2 * 2^{\frac{1}{m} * \sum_{j=1}^{m} M(j)}, & \text{if } \beta < \beta_s \\ -m * ln(\beta), & \text{if } \beta \geq \beta_s \end{cases}$$

- When $\beta \geq \beta_s$, the algorithm behaves like **Linear Counting**, using the number of empty buckets for better accuracy in small sets.
- When $\beta < \beta_s$, it reverts to the **LogLog** estimator, optimal for larger cardinalities.

Practical Benefits

- It maintains LogLog's space complexity of $\mathcal{O}(2^k)$ and per-element processing time of $\mathcal{O}(1)$,
- It automatically adapts to varying cardinalities,
- It supports mergeability through element-wise maximum of register arrays, like **LogLog** estimator does,
- Particularly effective in real-time systems such as **traffic matrix measurement** [31], with low error rates ($< 5\%$) across a wide cardinality range.

2.10 Hyper Log Log

HyperLogLog (HLL) [37] is an advanced probabilistic algorithm used for estimating the cardinality (i.e., the number of distinct elements) of very large multisets using extremely compact memory. It extends the LogLog algorithm by reducing variance through the use of harmonic means, rather than the geometric mean used in LogLog.

Like LogLog, HLL divides the hashed input stream into $m = 2^b$ buckets using the first b bits of a 32-bit hash value $h(x)$ to determine which register to update (calculating the index j). The remaining $32 - b$ bits (Our Python implementation uses 32-bit hashing) are used to determine the position of the first 1-bit, which becomes the value $\rho(w)$ which is potentially stored in register $M[j]$ (using `add()`).

$$\rho(w) = min\{k \geq 1 \mid \text{bit } k \text{ of remaining } w = 1\}$$

Each register $M[j]$ (for bucket j) stores:

$$M[j] := max(M[j], \rho(w))$$

At the end, the algorithm computes the harmonic mean of the transformed register values (using `estimate()`) to estimate cardinality $\hat{n}$:

$$Z := (\sum_{j=1}^{m} 2^{-M[j]})^{-1}$$

$$\hat{n} = a_m * m^2 * Z$$

where a_m is a bias-correction constant dependent on m, e.g.,

$$\alpha_m = \begin{cases} 0.673, & \text{if } m = 16 \\ 0.697, & \text{if } m = 32 \\ 0.709, & \text{if } m = 64 \\ \frac{0.7213}{1+1.079/m} & \text{if } m \geq 128 \end{cases}$$

HyperLogLog achieves a relative standard error (RSE) of:

$$RSE = \frac{1.04}{\sqrt{m}}$$

This is a significant improvement over LogLog's RSE of $\frac{1.3}{\sqrt{m}}$ while requiring only 64% of the memory used by LogLog to reach the same accuracy.

HLL is mergeable through `merge()`: register sets can be combined by taking pointwise maxima:

$$M_{merged}[j] = max(M_1[j], M_2[j], \dots)$$

The algorithm includes small and large range corrections for very small and very large $\hat{n}$, respectively, e.g., in the `estimate()` method, linear counting is selected when $\hat{n} \leq 2.5m$.

2.11 Hyper Log Log Plus

HyperLogLogPlus (HLL++) [30] is an enhanced version of the HyperLogLog (HLL) algorithm, developed to improve accuracy and memory efficiency, particularly for small and large cardinalities. Like HLL, it estimates the number of distinct elements in a multiset using a register array and probabilistic hashing, but introduces performance refinements and bias corrections that yield better results in practice.

2.11.1 Core Estimation Procedure (similar with HLL)

Like HyperLogLog, HLL++ divides the hashed input stream into $m = 2^b$ buckets using the first b bits of a 64-bit hash value $h(x)$ to determine which register to update (calculating the index j). The remaining $64-b$ bits are used to determine the position of the first 1-bit, which becomes the value $\rho(w)$, potentially stored in register $M[j]$ (via `add()`).

$$\rho(w) = min\{k \geq 1 \mid \text{bit } k \text{ of remaining } w = 1\}$$

Each register $M[j]$ (for bucket j) stores:

$$M[j] := \max(M[j], \rho(w))$$

At the end, the algorithm computes a bias-corrected harmonic mean of the transformed register values to estimate cardinality $\hat{n}$. The uncorrected "raw" estimate E is calculated as:

$$E := \alpha_m * m^2 * \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

Now, for small cardinalities, HLL's raw estimate tends to be biased. HLL++ uses an empirically derived bias table and k-nearest neighbor interpolation to adjust E, yielding a corrected estimate:

$$E^* = E - bias(E)$$

HLL++ selects Linear Counting to estimate cardinality when the number of empty registers is: $V > 0$ and the empirically defined threshold for switching to linear counting is: $T > E$:

$$E_{LC} = m * ln(\frac{m}{V})$$

Otherwise, it uses the bias-corrected raw estimate E^*.
For very small cardinalities, HLL++ uses a sparse representation (a compressed format that avoids storing full register arrays) to reduce memory usage significantly.

2.11.2 Final Estimation Formula

The final estimate returned by HLL++ (using `estimate`) is:

$$\hat{n} = \begin{cases} m \cdot \ln\left(\frac{m}{V}\right), & \text{if } E \leq T_1 \text{ and } V > 0 \\ E - \text{bias}(E), & \text{if } T_1 < E < T_2 \\ -2^{64} \cdot \ln\left(1 - \frac{E}{2^{64}}\right), & \text{if } E \geq T_2 \end{cases}$$

Where:

- T_1 is the empirically derived threshold (e.g., 11,500 for $p = 14$),
- T_2 is a large value approaching 2^{64}
- V is the number of zero-valued registers.

Finally, just like HLL, two or more HLL++ instances can be merged by taking the element-wise maximum of their registers (using `merge()`).

2.12 QDigest

Q-Digest [2] is a data structure designed for efficiently answering approximate quantile queries over integer datasets, especially in resource-constrained environments such as sensor networks. It allows estimation of quantiles, range queries, frequent items, and histograms with provable error guarantees.
Q-Digest represents data as a complete binary tree over a fixed domain $[1, \sigma]$, where each node v corresponds to a bucket covering a value range $[v.min, v.max]$. Each node tracks a count of values falling within its range. Nodes are selectively retained in the digest based on two properties (constraints):

- unless it is a leaf node, no node should have a high count:

$$count(v) \leq \lfloor n/k \rfloor \quad (1)$$

- a node and its children should not have low counts:

$$count(v) + count(parent(v)) + count(sibling(v)) > \lfloor n/k \rfloor \quad (2)$$

Here, n is the total number of elements inserted, and k is a user-defined compression factor. These rules ensure large counts are preserved while smaller counts are merged upward, compressing the structure.

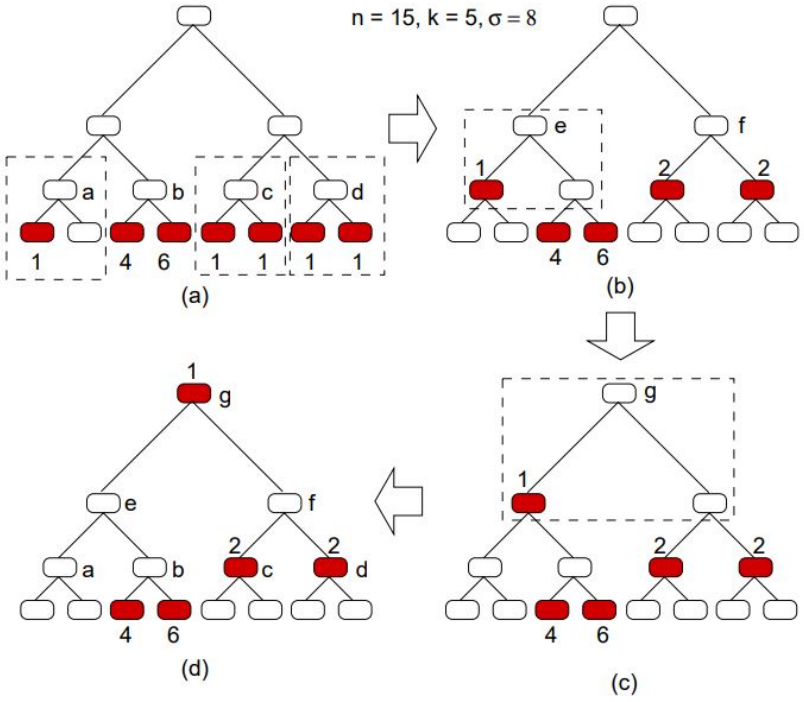

Figure 2.8: An example of building the **QDigest** from [2]. The leaf nodes represent values $[1 \dots 8]$ from left to right. Dark nodes in **(d)** are included in the final **QDigest**.

EXAMPLE 1 from [2]: Let a set of $n = 15$ values in the range $[1, 8]$ as shown in Figure 2.8**(a)**. The leaf nodes from left to right represent the values $1, 2, \dots, 8$ and the numbers next to the nodes represent the count. The number of buckets required to store this information exactly is 7 (one bucket per non-zero node). Let us assume a compression factor $k = 5, \lfloor n/k \rfloor = 3$. In Figure 2.8**(a)**, children of a, c, d violate digest property (2). So we compress each of these nodes by combining their children with them. Thus we arrive at the situation in Figure 2.8**(b)**. At this point, node e still violates the digest property. So we compress node e and arrive at Figure 2.8**(c)**. Node g still violates the digest property and so we compress g and arrive at our final q-digest shown in Figure 2.8**(d)**. Only 5 nodes are required to store it.

2.12.1 Accuracy and Error Bounds

Let m be the number of nodes retained (i.e., digest size). The relative error ϵ in any quantile query is:

$$\epsilon \leq \frac{3log\sigma}{m}$$

This bound arises from the fact that counts from merged nodes can float up the tree and affect ancestor nodes.

2.12.2 Core Operations

`add()`: In this function each inserted value x maps to a leaf node via:

$$leaf(x) = capacity + x$$

The count at the corresponding node is incremented. The structure then performs upward compression along the path from leaf to root using the `compress_upward()` function. This ensures the structure does not grow uncontrollably.

`merge()`: Two Q-Digests with the same compression factor are merged by:

- Aligning capacities through the `rebuild_to_capacity()` method described in our implementation in **Chapter 6**,

- Summing counts for each node:

$$merged\_count(v) = count_A(v) + count_B(v)$$

- Full recompression using a `compress_fully()` method described in our implementation in **Chapter 6**, to restore Q-Digest invariants.

This procedure is similar with Algorithm 2 from the paper [2]:

Coding Example 2.2: QDigest Merge Algorithm

```
1  MERGE(Q1(n1, k), Q2(n2, k)):
2    Q <- Q1 U Q2
3    COMPRESS(Q, n1 + n2, k)
```

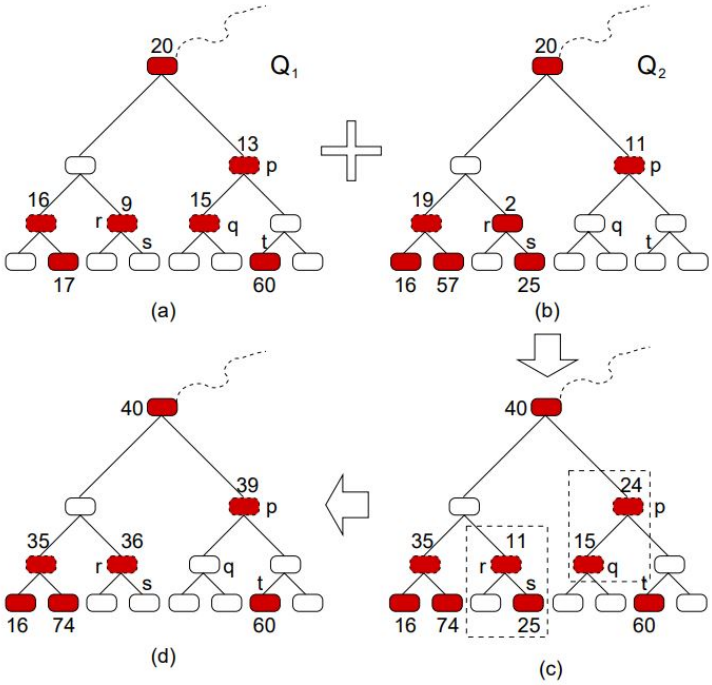

Figure 2.9: An example of merging two q-digest Q_1 and Q_2 from [2], shown in **(a)** and **(b)**. **(c)** shows the union of the two q-digests. **(d)** is the final q-digest after compression.

`estimate()`: returns the total number of elements inserted into the Q-Digest — i.e., the sum of all node counts.

$$estimate() = \sum_{v \epsilon Q} count(v) = n$$

2.12.3 Space Complexity

The maximum number of nodes in the digest is bounded by:

$$|Q| \leq 3k$$

This ensures predictable memory usage, important for embedded environments like sensor networks.

Chapter 3

Related Work

3.1 SnappyData

SnappyData [39] is a hybrid transactional/analytical processing (HTAP) system designed to support mixed workloads combining stream processing, online transaction processing (OLTP), and online analytical processing (OLAP). It tries to solve an important problem in the existing solutions, which can not handle all of these paradigms in a unified and performant manner. For example, `MemSQL` supports both OLTP and OLAP queries by storing data in dual formats (row and columns), but it needs an external streaming engine (e.g. `Storm, Kafka`) to support stream processing.

It is built on top of `Apache Spark` [8] and `Apache Geode` [6] `(GemFire)`, which helps **SnappyData** to leverage the advantages of both algorithms. More analytically, it uses `Apache Spark` as a batch-oriented computational engine, leveraging its high-throughput and `Apache GemFire` as a transactional store, taking advantage of its low-latency and in-memory transactional capabilities. This fusion enables **SnappyData** to support mutability, high availability (HA), and transactional consistency - characteristics which are not present in `Apache Spark`.

SnappyData features a hybrid storage layer capable of managing row-oriented, column-oriented, and probabilistic data structures (such as stratified samples). It supports both exact (e.g. fast reads/writes on indexed keys) and approximate query processing (AQP), which can exploit these probabilistic data structures.
Additionally, **SnappyData** can also summarize data in probabilistic data structures, such as `Stratified` Sampling, `Count-Min` Sketches, `HyperLogLog`, `Quantile` digests and other types of synopses. These structures enable **SnappyData** to support calculations of count or count distinct, summations, averages, quantile queries or cardinality estimations. SnappyData's **query engine** has built-in support for **AQP**, which can take advantage of these probabilistic structures. These features enable applications to trade accuracy for interactive speed analytics on streams or massive datasets where acceptable.
The system also provides `SQL` and `Spark` APIs, which enables the user to perceive **SnappyData** as a `SQL` database that uses `Spark`'s API for stream ingestion and stored procedures. It allows in-situ processing of streams using extended `Spark Streaming`.

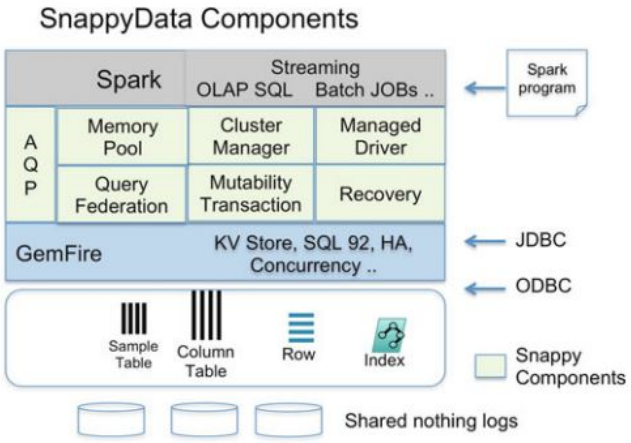

Figure 3.1: **SnappyData**'s core components (the original components from `Spark` and `GemFire` are highlighted) [39]

To ensure fault tolerance, **SnappyData** combines `Spark`'s Resilient Distributed Dataset (RDD) **lineage-based recovery** with `GemFire`'s **in-memory replication**.

Its scheduling model distinguishes between **low** (e.g., a key-based lookup) and **high**-latency operations (e.g. scans/aggregations). **SnappyData** avoids scheduling **overheads** for OLTP operations by immediately routing them to appropriate data partitions. Furthermore, its **architecture** supports data **co-partitioning**, **replication** and dynamic **rebalancing** of data, making it suitable for real-time operational analytics over large (stored or streaming) data.

3.2 Stream-lib

Stream-lib [1] is an open-source Java library for summarizing large-scale data in streams for which it is infeasible to store all events. In more detail, there are classes for estimating: **cardinality** (i.e. counting things); set **membership**; **top-k** elements and **frequency**:

- **Cardinality Estimation**: Stream-lib implements algorithms like **LinearCounting**, **LogLog** and **HyperLogLog** to estimate the number of distinct elements (cardinality) in a data stream. One particularly useful feature of these algorithms is that if there are instances of them with compatible configurations, they can be safely **merged**.
- **Frequency Estimation** (Top-K Elements): The library includes algorithms such as **CountMinSketch** and **StochasticTopper**, which count the frequency of elements in a stream of data. This is a very practical way to identify the most frequent items (top-k elements) without storing the entire dataset.
- **Set Membership**: Stream-lib provides algorithms such as **BloomFilter** for probabilistic set membership queries. These algorithms can check efficiently for whether an element is part of a dataset. They may occasionally produce false positives (indicating an element is present when it isn't), but never false negatives (missing elements that are actually present).

Many of the data structures in **Stream-lib** are designed to be **mergeable**. This feature is crucial for distributed computing environments, as it allows partial results from different nodes to be combined into a single summary.

Stream-lib does not support parallel processing.
Stream-lib offers command-line tools for quick testing and demonstration. For instance:

Coding Example 3.1: A Stream-lib command-line tool to estimate the frequency of items in the input stream

```
$ echo -e "foo\nfoo\nbar" | ./bin/topk
item count error
---- ----- -----
 foo     2     0
 bar     1     0

 Item count: 3
```

Figure 3.2: Stream-lib class diagram (no use of parallel processing)

3.3 Apache DataSketches

Apache DataSketches [3] [20] is an open-source library that got started and developed by **Yahoo** and is now maintained by the Apache Software Foundation. It is a high-performance library of stochastic streaming algorithms - known as "**sketches**". These sketches are small and stateful programs designed to process massive datasets. They can produce approximate answers to computationally difficult queries, with mathematically proven error bounds, much faster compared to traditional, exact algorithms.

3.3.1 Common Sketch Properties

- Sketches process data in a **single pass**, making them suitable for both real-time and batch.
- They maintains compact summaries, which use **sub-linear** space. In the beggining they are very small, yet these summaries grow very slowly or not at all, as the input data size increases, resulting in efficient memory utilization.
- Sketches can be **merged**, which enables distributed and parallel computation across many machines, a crucial characteristic for large systems.

- Users can **trade off** between sketch **size** and **accuracy**, in order to fit performance to particular applications' needs.

3.3.2 Key Sketch Types

- **Theta Sketch** [21]: provides approximate distinct counting with set operations like union, intersection and set difference.
- **HyperLogLog (HLL) Sketches** [4]: a highly efficient probabilistic data structure used for cardinality estimation, meaning they can count the number of unique elements in large datasets. The **Compressed Probabilistic Counting (CPC) sketch** is a similar algorithm with **HLL** and it is better than that in terms of accuracy per stored space, but it consumes more memory than **HLL**.
- **KLL Quantiles Sketch** [21]: used to estimate quantile, which enables to approximately compute percentiles and distribution summaries. This algorithm is more advanced than classic quantiles, because its sketches are more compactly organized for the same accuracy or they are more accurate for the same size.
- **Frequent Items Sketch** [29]: it tracks the approximate frequencies of items in a data stream and it calculates the most frequent items in that stream.
- **Variance Optimal Sampling (VarOpt)** [58]: maintains a representative sample from a stream of weighted items, which minimizes the variance when estimating the subset sums of items that match a given predicate. Each probability of including an item is approximately proportional to its weight relative to the total weight of all items in the sketch.

Apache DataSketches is implemented in `Java`, `C++`, and `Python`. It is compatible with computing environments that must handle Big Data, like: `Apache Hive, Apache Pig, PostgreSQL, Apache Druid`, and `Apache Spark`, among others.

This technology has been instrumental for **Yahoo** to reduce the processing times of massive data analytics from days or hours to minutes or seconds [3].

3.4 StreamApprox

StreamApprox [49] is a system for approximate computing in stream analytics, taking advantage of a novel **Online Adaptive Stratified Reservoir Sampling (OASRS)** algorithm, shown in Figure 3.3. It is developed to address the limitations of existing batch-processing approximate systems.

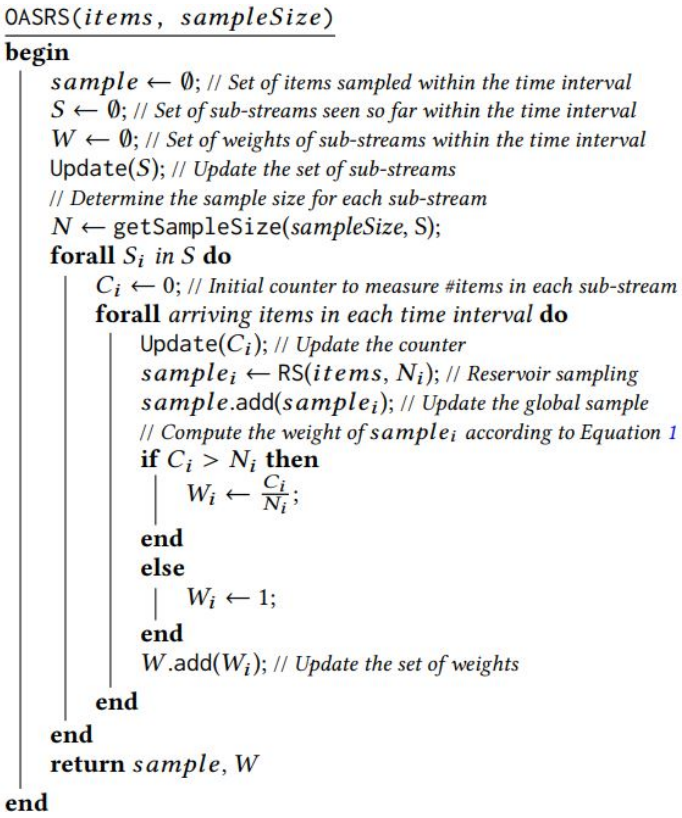

Figure 3.3: Algorithm 3: Online adaptive stratified reservoir sampling (OASRS) [49]

3.4.1 Key Contributions of SteamApprox

1. **OASRS**:
   - combines both stratified and reservoir samplings, keeping only their advantages. More specifically, OASRS does not sample a dataset depending on which sub-stream is more popular or has better statistics. It also runs efficiently in real time in a distributed manner.
   - samples "on-the-fly" without having prior knowledge of all data.
2. Unlike traditional batch-oriented approximate systems (e.g., BlinkDB), it works with both **batched** (`Apache Spark Streaming` [9]) and **pipelined** models (`Apache Flink` [5]).
3. Based on the **trade-off** between the output accuracy and computation efficiency (query execution budget), it determines the sample size accordingly. It also provides rigorous error bounds on the produced approximate results.
4. Performs the sampling operation across many workers in **distributed** systems, **without** the need for **synchronization** between these workers.
5. It achieves a 1.1x-2.4x speedup compared to a Spark-based approximate computing system, which uses `Apache Spark`'s [8] existing sampling modules.

3.4.2 Architecture

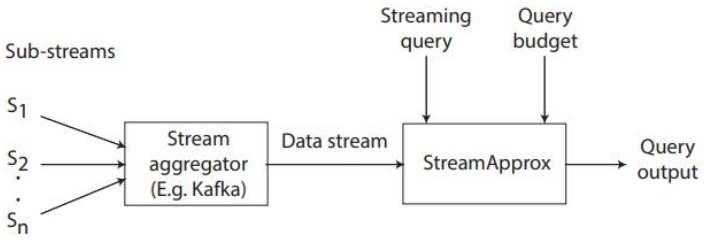

Figure 3.4: A high-level view of the **StreamApprox** architecture [49]

Figure 3.4 depicts the overall architecture of **StreamApprox**. In the beginning, incoming multiple **sub-streams** ($S_1, S_2, \ldots, S_n$) of data are processed by the system using a **Stream Aggregator** (e.g. Apache Kafka [7]), which applies the **OASRS** algorithm on these sub-streams to select a single representative **Data Stream**. The sample size is configured by a user-specified **Streaming query** (e.g. `SUM, AVG, COUNT`) that the users want to run on the incoming data, along with a user-defined **query budget**, which can be: latency-bound (e.g., complete in 1 second), resource-bound (e.g., limited CPU/memory), or accuracy-bound (e.g., 95% confidence level). Based on these queries, the appropriate sample size is determined to meet resource or accuracy constraints.
The sampled data is then processed by a distributed stream processing engine (`Apache Spark Streaming` [9] or `Apache Flink` [5]), and the results are returned with rigorously computed error bounds. This design enables **StreamApprox** to efficiently execute approximate queries in real-time stream environments while preserving statistical correctness.

3.5 A Synopses Data Engine for Interactive Extreme-Scale Analytics

Kontaxakis et al proposes the novel architecture of a **Synopses Data Engine (SDE)** [22] that enhances the scalability and flexibility of real-time and interactive data analytics over massive data streams. The **SDE**'s architecture, built on top of `Apache Flink and Kafka` [5] [7], introduces a powerful **Synopsis-as-a-Service (SDEaaS)** paradigm. It supports a broad spectrum of commonly used synopses to estimate cardinality (e.g. `HyperLogLog`), frequency (e.g. CountMin), correlation (e.g. `Discrete Fourier Transform`), set membership (e.g. `BloomFilter`) or quantiles (e.g. `GKQuantiles`).
It also provides three types of scalability:

- **Horizontal**: parallelization of the computation across multiple machines in a cluster to efficiently process large-scale data streams.
- **Vertical**: the property of scaling the computation with the volume of processed data, such as detecting highly correlated stock data streams using various statistical metrics.
- **Federated**: the ability of enabling scalable computations across geographically distributed locations where data is generated or received independently.

3.5.1 SDEaaS API Key Contributions

- A new synopsis type can be **added** or **terminated** at **runtime**, without stopping the system.
- To avoid shutting down our continuously running SDEaaS whenever we need to upgrade its capabilities (such as adding support for a new synopsis), the system is designed to support the pluggability of the code of **additional** (not included in the SDE Library) **synopses** to be **dynamically loaded** and **managed** at **runtime**.
- The SDE supports **one-shot, ad-hoc** queries on a given synopsis and returns estimates based on its current state.

- The system supports **continuous querying**, where estimations - such as counts or correlations - are generated each time the synopsis is updated, for example, when a new tuple is received.

Through extensive experiments, the authors demonstrate that **SDE outperforms** prior systems such as `Proteus` [43] in both throughput and scalability, especially when maintaining large numbers of concurrent synopses.

3.5.2 SDE Architecture

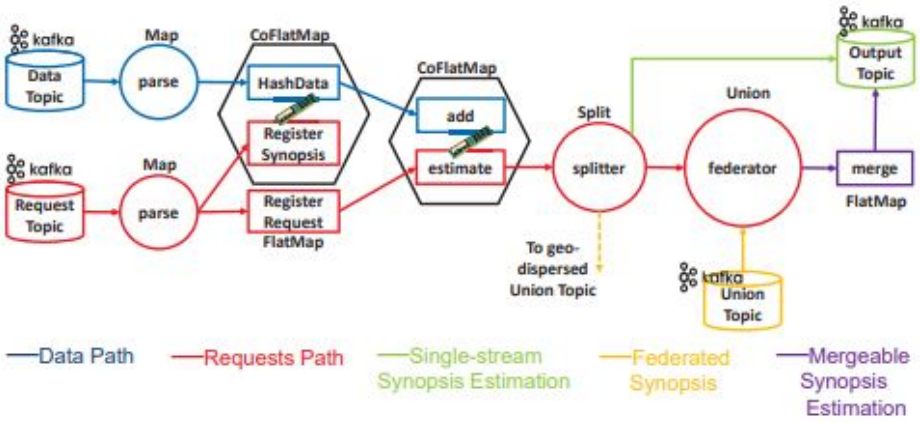

Figure 3.5: **SDE Architecture** - Condensed View [22]

SDE's architecture, illustrated in Figure 3.5, consists of three major pathways:

1. **Red Requests** Path: handles the control messages of maintaining a new synopsis and issuing a query. These requests arrive on a Kafka `RequestTopic` and are parsed into two `FlatMap` operators which produce the parameters for the messages:
   - `RegisterRequest` uses these parameters to determine in which worker(s) an ad-hoc query should be sent.
   - `RegisterSynopsis` uses the same parameters to determine which worker(s) should receive a data tuple intended to update one or more synopses, following the **blue** path shown in Figure 3.5.
2. **Updating** the Synopsis (**blue** path): ingests data tuples destined to update one or more synopses from a Kafka `DataTopic`:
   - The tuples are hashed (through `HashData`) and routed based on precomputed keys (taken from `RegisterSynopsis`) to a worker that maintains the corresponding synopsis.
   - These tuples are then directed to an `add` operator, which applies the **update** logic of the synopsis (e.g of an `FM sketch`).
3. **Query Answering** (**Red**/**Yellow** Path): Queries (ad-hoc or continuous) through the `RequestTopic` are directed to `RegisterRequest` which produces the keys following the same procedure as `RegisterSynopsis` does. Leveraging these keys the queries are routed to an `estimate` operator, where:
   - if the synopsis is locally maintained (single-stream), the answer is returned directly via the `Output Topic`.
   - If the synopsis is **federated** (distributed across clusters) or **non-federated** defined on entire **data sources**, the `federator` and `merge` operators combine partial results from remote workers (yellow and purple paths) to produce the final answer again via the `Output Topic`.

Operators like `FlatMap`, `CoFlatMap`, `Union`, and `Split` are part of `Apache Flink`'s [5] streaming model.

Another significant component is **Parallelization**, which is based on stream identifiers or synopsis-specific keys to distribute computation efficiently across the cluster.

This architecture ensures that the system:

- Is fault-tolerant and scalable by design,
- allows runtime addition of new synopses without downtime,
- minimizes redundant computations by reusing synopses across workflows.

3.6 And synopses for all: A synopses data engine for extreme scale analytics-as-a-service

In addressing the challenges of real-time analytics over massive data streams, the work of Kontaxakis et al. [23] builds upon their earlier architecture (Kontaxakis et al. [22]) on the **Synopses Data Engine (SDE)** to present a more refined and production-ready version of their stream analytics platform. While the **original SDE** introduced the core concept of **Synopsis-as-a-Service (SDEaaS)**, [23] fully materializes it by significantly adding important architectural, functional and performance improvements. Similarly with [22], [23] also supports **horizontal**, **vertical** and **federated** scalability efficiently by using `Apache Flink` [5] for **stream processing** and `Apache Kafka` [7] for **communication**.

Key Contributions

1. **SDEaaS** is implemented as a constantly running job in `Apache Flink` clusters that:
   - adds new synopses types without interrupting existing workflows,
   - can accept requests to build, stop, or query synopses at runtime. Figure 3.6 shows how users configure and send synopsis creation requests at runtime.
   - shares synopses across concurrent workflows to avoid duplication.
2. The new SDE features a **Synopsis Library**, which is an expanded version of [22]'s library, which included a more limited set. It adds a richer and more diverse set of **synopsis techniques**, grouped into:
   - Sampling (e.g. `STSampler`, `Distributed Sampling`)
   - Counting and Frequency (e.g. `Lossy Counting`, `Sticky Sampling`)
   - Correlation and Norms (e.g. `AMS Sketch`, `Radius Sketch Family`)
   - Quantiles (e.g. `GKQuantiles`, `AMQuantiles`)

   It includes support for **windowed synopses**, **dynamic configuration**, and control over **accuracy parameters** (e.g. ϵ, δ).

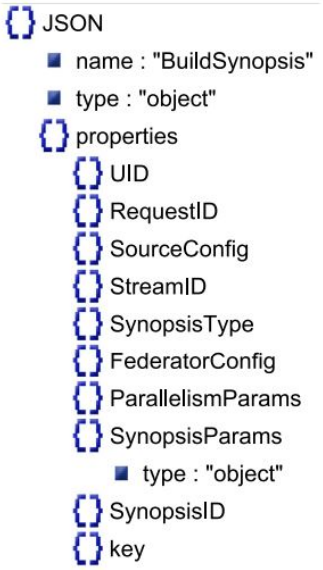

Figure 3.6: JSON snippet for BuildSynopsis request [23]

3. **SDEaaS** retains the base architecture of **SDE** from [22], but:
   - It fully integrates with `Kafka` for ingesting requests and data. It also handles communication across geo-distributed SDE instances.
   - It introduces new operators (e.g., `federator`, `coordinator`, `estimate`) that enable dynamic, parallel and federated execution.
4. [22] demonstrated potential for horizontal, vertical, and federated scalability, while [23] validates these scalability types through:
   - **Thousand-scale** concurrent synopsis maintenance versus only tens in comparable systems like `Condor` [36].
   - Significant reduction in network **communication cost** due to synopsis mergeability.
   - Synopses can share resources via **slot sharing** and as a result, streams and workloads are mapped optimally to processing units.
5. In [22] the system was a prototype. Then, in [23] it got integrated into `RapidMiner Studio` [45] with a GUI that allows users to configure **SDEaaS** through a graphical interface. Also, it is cross-platform compatible with other big data systems through JSON-based configuration.

3.7 Related Work Overview

Table 3.1 from Giatrakos et al. [10] offers a consolidated comparison of major stream summarization libraries and synopsis frameworks based on their **scalability** characteristics. Specifically, it categorizes each synopsis framework across three types of scalability: **Horizontal**, **Vertical** and **Federated**, which are already explained in section **3.5**.

Scalability → **Synopses Approach ↓**	**Horizontal**	**Vertical**	**Federated**	**Tensor Compatibility**
Apache DataSketch [3]	(Spark Integration)	X	X	X
Stream-lib [1]	X	X	X	X
StreamApprox [49]	(Stratified Sampling)	X	X	X
SnappyData [39]	(Simple Aggregates)	X	X	X
Condor [36]	√	X	X	X
SDaaS [23]	√	√	√	X

Table 3.1: Stream Summarization & Scalability [10]

From the table, it is observed that while frameworks like `Apache DataSketch` [3], `StreamApprox` [49], and `SnappyData` [39] offer **horizontal** scalability, they do not support **vertical** and **federated** scalability, thus limiting their utility in more complex and distributed settings. In contrast, **SDEaaS (Synopses Data Engine as a Service)** emerges as the most comprehensive synopsis approach, supporting all three scalability dimensions. However, none of these synopses can directly support training pipelines in `Tensorflow` [52] or `PyTorch` [44] because they are **not tensor-compatible**.
Therefore, that is the reason why our work builds upon `Apache Dask` [11], a distributed computation framework that provides tensor-compatible operations, enabling both the distributed computation of data synopses and the seamless integration of summarized outputs into deep learning frameworks.

In the following sections, the terms **structure**, **synopsis**, or **class** will be used interchangeably.

Chapter 4

Apache Dask

Dask [11] is an open-source Python library designed to facilitate parallel and distributed computing, enabling efficient handling of large (out of memory) datasets and complex computations across multiple cores or machines (workers). It integrates seamlessly with existing Python tools like NumPy, pandas, scikit-learn and Tensorflow allowing for scalable data processing without significant code modifications.

4.1 Task Graphs and Lazy Evaluation

At the heart of Dask's functionality is the construction of task graphs, which represent computations as directed acyclic graphs (DAGs). Each node in the graph corresponds to a computational task, and edges denote dependencies between these tasks. Dask employs lazy evaluation, meaning it builds the task graph without executing it immediately. Execution is deferred until explicitly triggered using `compute()`, allowing for optimization and efficient scheduling of tasks.

This is an example of python code with the task graph it produces:

Coding Example 4.1: Chaining Delayed Computations of adding elements with Dask

```
from dask import delayed

def add(x, y):
    return x + y

a = delayed(add)(1, 2)
b = delayed(add)(a, 3)
b.compute()  # triggers computation of the full graph
```

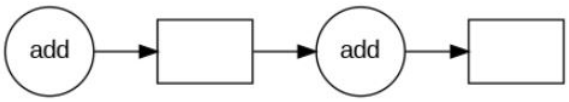

Figure 4.1: The Task graph of python example 4.1

4.2 Dask Collections

Dask offers several high-level collections that mirror familiar data structures but are designed for parallel and out-of-core computation:

4.2.1 Dask Array

`dask.array` [15] provides a parallel, chunked version of NumPy arrays. It divides large arrays into smaller blocks (chunks), where each chunk is a NumPy array, that can be processed independently and in parallel.

Coding Example 4.2: Mean value calculation using Dask Arrays

```
import dask.array as da

# Create a large Dask array with random values, divided into chunks
x = da.random.random((10000, 10000), chunks=(1000, 1000))

# Compute the mean of the array
mean_value = x.mean().compute()
print(mean_value)
```

In coding example 4.2, a large dask array of size $10^4 * 10^4$ is divided into 10 chunks of size: $1000 * 1000$, each of which can be processed in parallel to compute the array's mean value.

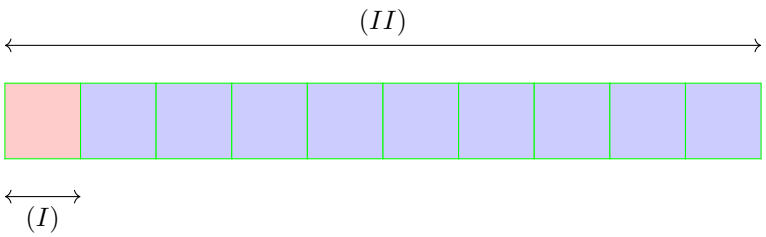

Figure 4.2: A representation of the above example's dask array of size: $10^4 * 10^4$ (II), in comparison to one of its Numpy array chunks of size: $10^3 * 10^3$ (I)

4.2.2 Dask DataFrame

`dask.dataframe` [18] offers a parallel version of pandas DataFrames, suitable for larger-than-memory datasets. It partitions the data into smaller pandas DataFrames, enabling parallel operations.

Coding Example 4.3: Mean value calculation of a column of a csv file using Dask DataFrames

```
import dask.dataframe as dd

# Read multiple CSV files into a partitioned Dask DataFrame
df = dd.read_csv('data/*.csv')

# Compute the mean of a column
mean_value = df['column_name'].mean().compute()
print(mean_value)
```

In coding example 4.3, Dask reads multiple CSV files into partitions that can be processed in parallel to compute the mean.

4.3 Parallel and Delayed Computation

4.3.1 Dask Delayed

For custom workflows that don't fit into Dask's high-level collections, `dask.delayed` [19] allows for parallelizing arbitrary Python code by deferring function execution and building a task graph.

Coding Example 4.4: Building and Executing a Parallel Task Graph with Dask Delayed

```
import time
from dask import delayed

@delayed
def inc(x):
    time.sleep(1)
    return x + 1

@delayed
def double(x):
    time.sleep(1)
    return x * 2

@delayed
def add(x, y):
    time.sleep(1)
    return x + y

# Build the computation graph
a = inc(1)
b = double(2)
c = add(a, b)

# Execute the computation
result = c.compute()
print(result)
```

In coding example 4.4, the functions are decorated with @delayed, creating the following task graph that is executed when compute() is called:

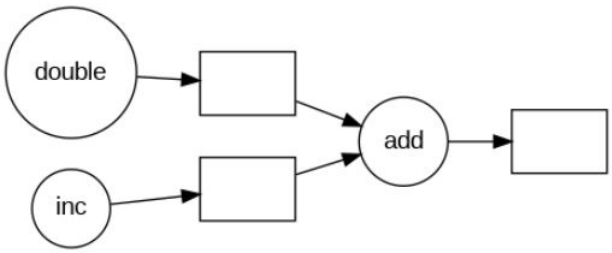

Figure 4.3: The task graph of Python coding example 4.4

It can be noticed that `inc()` and `double()` functions are executed in parallel across two cores, making this Python code faster than its sequential counterpart.

4.3.2 Dask Map_blocks

For operations on large arrays, `dask.array.map_blocks` [16] enables to apply a function to each block (chunk) of a Dask array in parallel. This is particularly useful when dealing with large datasets that do not fit into memory.

Coding Example 4.5: Block-wise Normalization of Large Arrays using `dask.array.map_blocks`

```
import dask.array as da
import numpy as np

# Create a large Dask array with random values
x = da.random.random((10000, 10000), chunks=(2500, 2500))

# Define a function to apply to each block
def normalize(block):
    return (block - np.mean(block)) / np.std(block)

# Apply the function to each block
normalized = x.map_blocks(normalize)

# Compute the result
result = normalized.compute()
```

In coding example 4.5 , `normalize()` is applied to each $2500*2500$ block of the array x. The computation is executed in parallel using all available CPU cores (such as the 4 cores provided by Google Colab, therefore that is why the array is divided by 4 ($10000/4 = 2500$) to enable efficient processing across the array's partitions) across all blocks.

4.4 Distributed Computing with LocalCluster

Dask's `distributed` module [26] enables the execution of computations across multiple cores or machines. `LocalCluster` sets up a scheduler and workers on the local machine, facilitating parallel processing.

Coding Example 4.6: Setting up a Local Cluster using four workers and one thread per worker

```
from dask.distributed import Client, LocalCluster

# Set up a local Dask cluster
cluster = LocalCluster(n_workers=4, threads_per_worker=1, dashboard_address=':8888')
client = Client(cluster)
```

This setup allows Dask to distribute tasks across the available cores on the local machine (Google Colab was used which has four available cores (`workers`)). Only one `threads_per_worker` is used to avoid causing race conditions. Execution can be monitored with a real-time dashboard at `http://localhost:8888`.

4.5 Exposing Local Dask Dashboard with Ngrok

To monitor and debug Dask computations, the Dask dashboard provides real-time insights. If one runs Dask on a local machine and needs to access the dashboard remotely, Ngrok [57] can be used to expose the local server.
Steps to set up Ngrok:

- Start the Dask cluster and note the dashboard address (e.g., `http://127.0.0.1:8888`) [26].
- Start Ngrok to expose the dashboard [57]:

Coding Example 4.7: Setting up of Ngrok to expose Local Dask Dashboard

```
from pyngrok import ngrok, conf
import getpass

print("Enter your authtoken, which can be copied "
"from https://dashboard.ngrok.com/get-started/your-authtoken")
conf.get_default().auth_token = getpass.getpass()

ui_port = 8888
public_url = ngrok.connect(ui_port).public_url
print(f" * ngrok tunnel \"{public_url}\" -> \"http://127.0.0.1:{ui_port}\"")
```

- Ngrok will provide a public URL (ngrok tunnel) (e.g., `http://abc123.ngrok-free.app`) which can be used to access the Dask dashboard remotely.

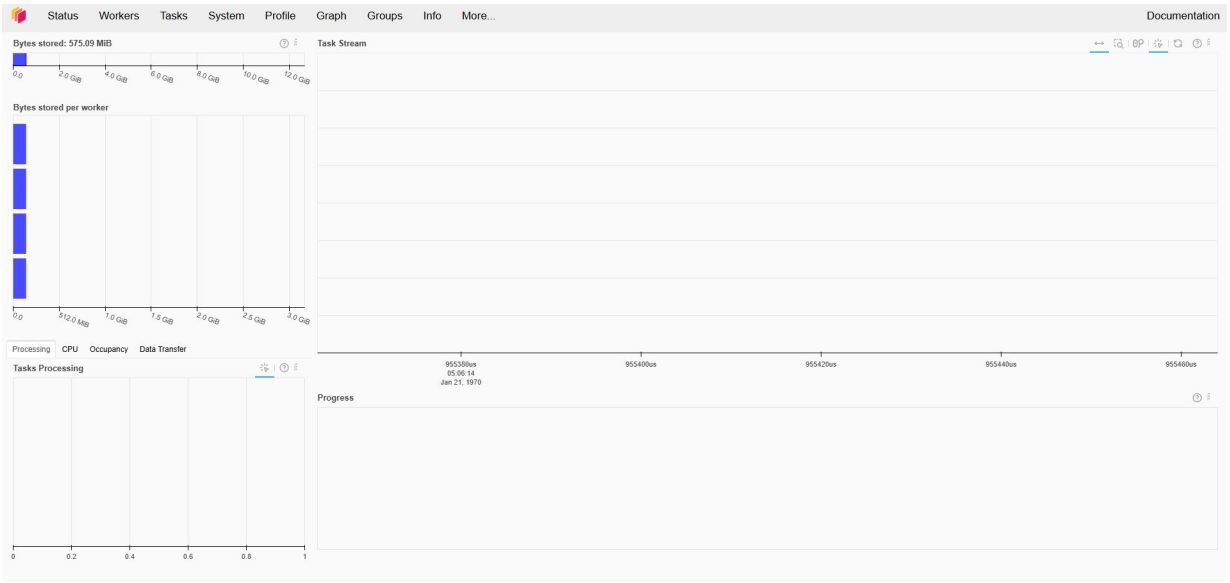

Figure 4.4: The local dask dashboard [17] set with 4 workers (cores) before any python script is executed

In the **upper left** box of Figure 4.4 all the bytes that are stored into cluster memory are shown (Bytes Stored).

In the box below that the **bytes stored per worker** are shown, where a quarter of the whole data are stored in each, as four workers are used.

The **lower left** of this image has four tabs where one of them shows the **tasks that are being processed** by each worker, where when all workers execute in parallel four tasks are shown to be executed, equal with the system's workers multiplied by the threads each worker has, which in our case is one. Moreover, the **CPU tab** shows the CPU that each worker utilizes. Also, the **Occupancy tab** shows the occupancy, in time, per worker. The total occupancy for a worker is the amount of time Dask expects it would take to run all the tasks, and transfer any of their dependencies from other workers, if the execution and transfers happened one-by-one. Finally, the Data Transfer tab shows the size of open data transfers from/to other workers, per worker.

The **lower right box** shows the progress of each individual task-prefix. The color of each bar matches the color of the individual tasks on the task stream from the same task-prefix.

Finally, the **upper right box** shows the task stream, which is a view of which tasks have been running on each thread of each worker. Each row represents a thread, and each rectangle represents an individual task. The color for each rectangle corresponds to the task-prefix of the task being performed, and matches the color of the **Progress box**. This means that all the individual tasks which are part of the `stack` task-prefix, for example, will have the same color (which is chosen randomly from the viridis color map).

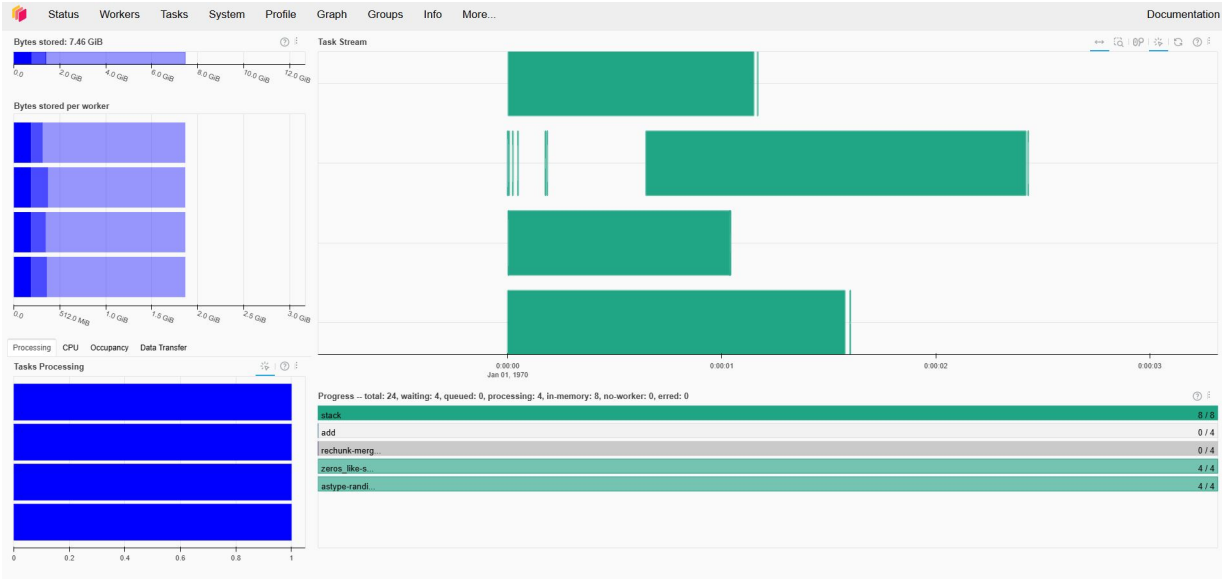

Figure 4.5: The local dask dashboard set with 4 workers, while a python script is being executed using dask.map_blocks

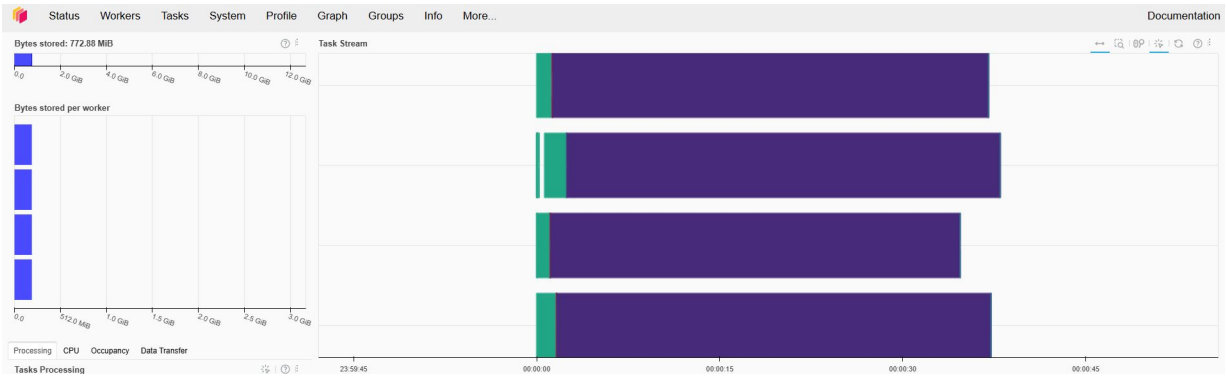

Figure 4.6: The local dask dashboard when the same python script, using dask.map_blocks, has finished execution

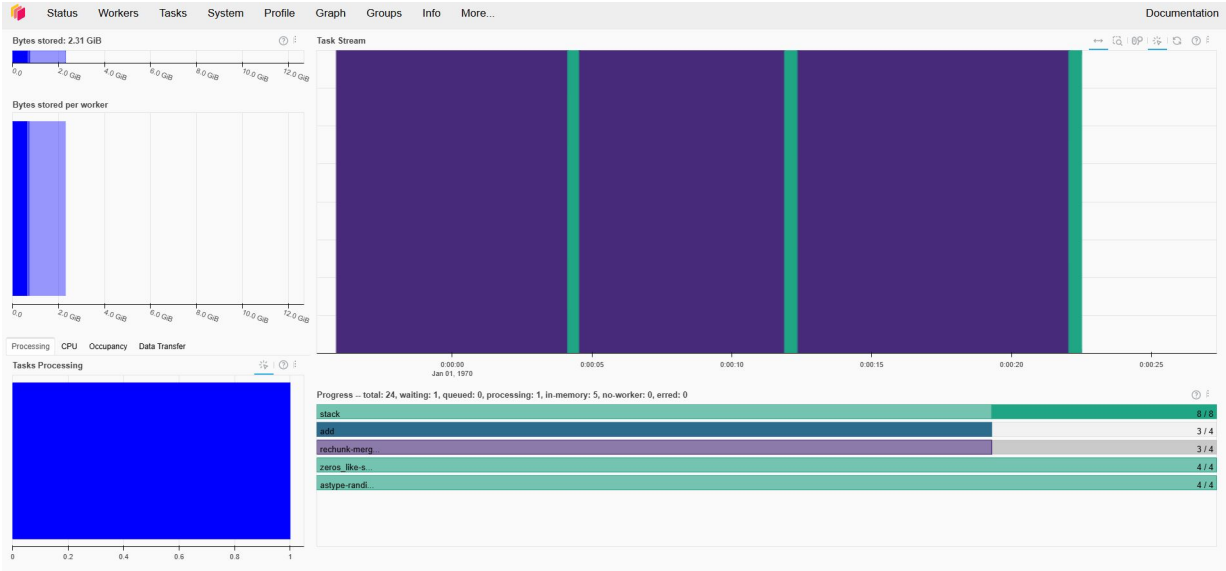

Figure 4.7: The local dask dashboard set with only 1 worker, while the same python script is being executed using dask.map_blocks

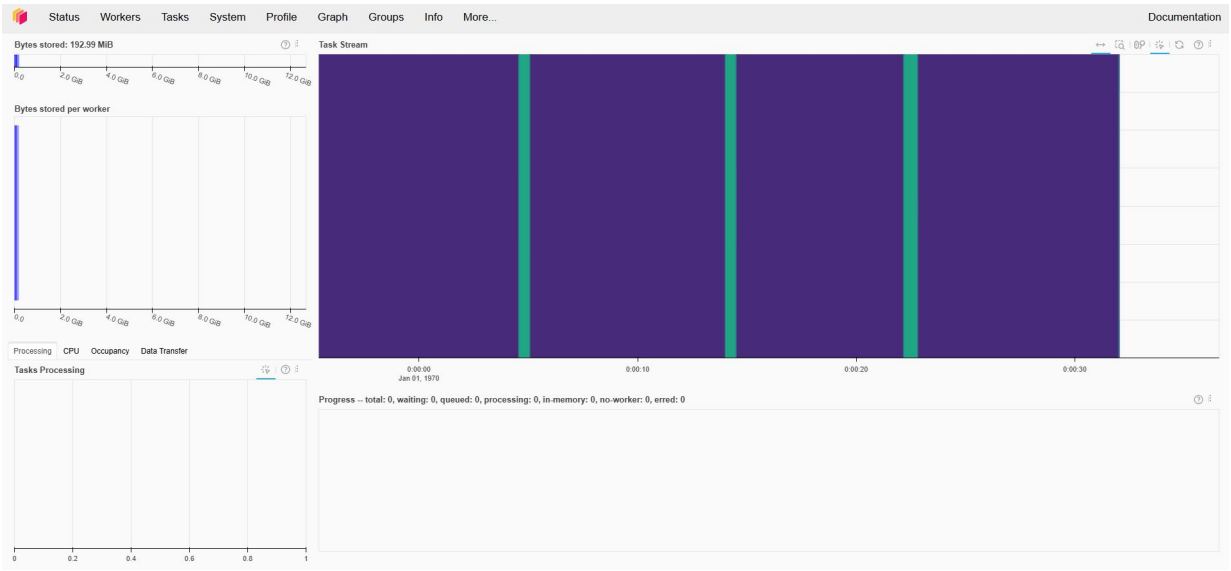

Figure 4.8: The local dask dashboard when the same python script, using dask.map_blocks, has finished execution

In Figure 4.8, it can be noticed that the example with one worker takes longer to execute because all four data chunks are processed sequentially on a single core. In contrast, in Figure 4.6, the example with four workers assigns one chunk to each worker, allowing all four chunks to be processed in parallel.

Chapter 5

Tensorflow

TensorFlow [52] is an open-source machine learning framework. It is widely used for building and training neural networks across various platforms, including CPUs, GPUs, and TPUs. TensorFlow supports multiple programming languages, such as Python, JavaScript, C++, and Java, facilitating its use in a range of applications in many sectors.

5.1 Core Concepts and Architecture

5.1.1 Tensors and Operations

At the heart of TensorFlow are **tensors**, which are multi-dimensional arrays, and **operations** (ops), which are nodes in a computational graph that represent mathematical computations. These tensors flow through the graph, hence the name "TensorFlow."

Coding Example 5.1: Two multi-dimensional tensors and an operation (multiplication)

```
import tensorflow as tf

matrix1 = tf.constant([[1, 2], [3, 4]])
matrix2 = tf.constant([[5, 6], [7, 8]])
product = tf.matmul(matrix1, matrix2)
print(product)
```

Output of coding example 5.1:

```
tf.Tensor(
[[19 22]
 [43 50]], shape=(2, 2), dtype=int32)
```

5.1.2 Computational Graphs

TensorFlow uses computational graphs [32] to represent and execute operations. In TensorFlow 2.x, eager execution is enabled by default, which allows operations to execute immediately as they are called from Python.

Creating Graphs with `@tf.function`

For performance optimization, TensorFlow provides the `@tf.function` [12] decorator to convert Python functions into graph-executed functions [32], which enables faster execution, as it enables performance optimizations to environments without a Python interpreter.

Coding Example 5.2: A characteristic example of `@tf.function`'s functionality

```
import tensorflow as tf

# Create some large matrices
a = tf.random.normal([1000, 1000])
b = tf.random.normal([1000, 1000])

# Non-decorated version
def matmul_loop():
    for _ in range(100):
        tf.matmul(a, b)

# Decorated version using @tf.function
@tf.function
def matmul_loop_graph():
    for _ in tf.range(100):  # tf.range is needed for graph tracing
        tf.matmul(a, b)
```

In coding example 5.2, the decorated version (`matmul_loop_graph()`) takes 0.002 secs to execute, while the non-decorated version (`matmul_loop()`) takes 4.68 secs to execute. These measurements were taken on Google Colab's standard virtual environment using CPU only (no GPU/TPU). `@tf.function` is almost 2,340 times faster because TensorFlow builds a static computation graph which eliminates Python function call overhead and it also enables runtime optimization like better memory planning.

5.2 Building Neural Networks with Keras

TensorFlow integrates tightly with Keras [34], a high-level API for building and training deep learning models. Keras provides two main ways to build models: the Sequential API and the Functional API.

5.2.1 Sequential API

The Sequential API [55] is suitable for models that have a single input and output, with layers stacked in sequence.

Coding Example 5.3: A Sequential API example

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Define Sequential model with 3 layers
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu", name="layer1"),
        layers.Dense(3, activation="relu", name="layer2"),
        layers.Dense(4, name="layer3"),
    ]
)
# Call model on a test input
x = tf.ones((3, 3))
y = model(x)
```

Model: "sequential_8"

Layer (type)	Output Shape	Param #
layer1 (Dense)	(3, 2)	8
layer2 (Dense)	(3, 3)	9
layer3 (Dense)	(3, 4)	16

Figure 5.1: Model Summary of coding example 5.3.

In Figure 5.1, "Model: "sequential_8"" is just the name of the model (the 8th sequential model created in the current environment). There are 3 layers, and all of them are `Dense` (fully connected) layers. This is the breakdown of these layers:

- Output Shape: For instance, layer1 has (3,2), which means the batch size is 3, and the output dimension of the layer is 2.
- Param #: Formula for Dense layer parameters. For layer1: $(3 * 2) + 2 = 8$ (input dimensions must be 3). For layer2: $(2 * 3) + 3 = 9$, which means input to this layer is 2 (from the output of layer1), and it outputs 3 values per sample.

5.2.2 Functional API

The Functional API [54] is more flexible and allows for models with multiple inputs and outputs, or models with non-linear topology.

Coding Example 5.4: A Functional API example

```
inputs = keras.Input(shape=(784,))
dense = layers.Dense(64, activation="relu")
x = dense(inputs)
x = layers.Dense(64, activation="relu")(x)
outputs = layers.Dense(10)(x)
model = keras.Model(inputs=inputs, outputs=outputs, name="functional_model")
```

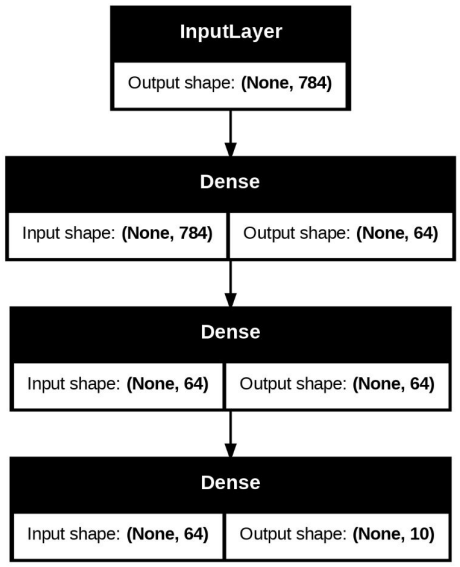

Figure 5.2: The graph of Coding example's 5.4 model

5.3 Advanced Topics

5.3.1 Convolutional Neural Networks (CNNs)

For image data, CNNs [14] are commonly used due to their ability to capture spatial hierarchies. TensorFlow provides layers like `Conv2D` and `MaxPooling2D` to build CNNs.
Example of :

Coding Example 5.5: A CNN model trained with images of the CIFAR10 dataset [53]

```
(train_images, train_labels), (test_images, test_labels) = ...
    datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

```
model = models.Sequential() # convolutional base
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten()) # adding dense layers
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))

model.compile(optimizer='adam',
              loss= tf.keras.losses.SparseCategoricalCrossentropy ...
                  (from_logits=True),
              metrics=['accuracy']) # compiling

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels)) # training

test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2) # ...
    evaluation
```

Coding example 5.5 demonstrates loading the CIFAR10 dataset, normalizing the data, building a simple feedforward neural network (using `model.add()`, compiling it with an optimizer and loss function (`model.compile()`), training the model (`model.fit()`), and evaluating its performance on test data (`model.evaluate()`).

Model: "sequential_3"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 64)	36,928
flatten (Flatten)	(None, 1024)	0
dense_3 (Dense)	(None, 64)	65,600
dense_4 (Dense)	(None, 10)	650

Figure 5.3: Model Summary of coding example 5.5

Similarly with Figure 5.1, Figure 5.3 represents the summary of a sequential CNN model produced from coding example 5.5 and it holds eight different layers. This is the breakdown of these layers:

1. **Conv2D**: Has an input of size: $32 * 32 * 3$, an output of: $(30, 30, 32)$ (Output image is slightly smaller due to kernel size). This layer converts 32 filters into 32 channels. It has parameters: $(3 * 3 * 3) * 32 + 32 = 896$ (3×3 kernel, 3 input channels, 32 filters, +32 biases).
2. **MaxPooling2D**: Has output shape: $(15, 15, 32)$. This layer reduces each spatial dimension by half (using 2×2 pooling) and it holds zero parameters.
3. **Conv2D**: Has output shape: $(13, 13, 64)$. It holds 32 input channels from previous layer. It has parameters: $(3 * 3 * 32) * 64 + 64 = 18,496$.
4. **MaxPooling2D**: Has output shape: $(6, 6, 64)$ and it again halves spatial size.
5. **Conv2D**: Has output shape: $(4, 4, 64)$. It has parameters: $(3 * 3 * 64) * 64 + 64 = 36,928$.
6. **Flatten**: Converts the 4×4×64 feature map into a 1D vector: $4 * 4 * 64 = 1024$. Has output shape: (1024).
7. **Dense**: Is a fully connected layer with 64 units. It has parameters: $1024 * 64 + 64 = 65,600$.
8. **Dense**: Is the final output layer with 10 units. It has parameters: $64 * 10 + 10 = 650$.

5.3.2 Recurrent Neural Networks (RNNs)

For sequential data, such as time series, RNNs [59] and their variants like LSTM and GRU are utilized. TensorFlow's Keras API includes these layers for building models that can capture temporal dependencies.

Here is a simple example of a Sequential model that processes sequences of integers, embeds each integer into a 64-dimensional vector and then processes the sequence of vectors using a LSTM layer:

Coding Example 5.6: A Sequential model using LSTM layer

```
model = keras.Sequential()
# Add an Embedding layer expecting input vocab of size 1000, and
# output embedding dimension of size 64.
model.add(layers.Embedding(input_dim=1000, output_dim=64))

# Add a LSTM layer with 128 internal units.
model.add(layers.LSTM(128))

# Add a Dense layer with 10 units.
model.add(layers.Dense(10))
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 64)          64000

 lstm (LSTM)                 (None, 128)               98816

 dense (Dense)               (None, 10)                1290

=================================================================
Total params: 164106 (641.04 KB)
Trainable params: 164106 (641.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 5.4: Model Summary of coding example 5.6

Figure 5.4 shows that the model has 3 layers, which are:

- **Embedding**: Converts integer-encoded tokens (like words or characters) into dense vectors of fixed size (embeddings). It has parameters: $64 * \frac{64000}{64} = 1000$ and output Shape: `(batch_size, sequence_length, 64)`.
- **LSTM**: Processes the sequence of embeddings and captures temporal dependencies. Has output shape: `(batch_size, 128)` (This is a single 128-dimensional vector per sequence). Parameters are counted as: $4 * (128 * (128 + 64 + 1)) = 98,816$. $units = 128 \quad \& \quad input\_dim = 64$.
- **Dense**: Fully connected layer, usually for classification or regression. Has output shape: `(batch_size, 10)`. Parameters are counted as: $128(\texttt{input}) * 10(\texttt{units}) + 10(\texttt{bias}) = 1290$.

Chapter 6

Our Toolkit

Our software architecture builds a parallel, Python-based stream synopsis engine designed to operate efficiently on large-scale datasets using **Dask** for parallelism and **TensorFlow** for learning. At its core lies a framework of abstracted synopsis classes (which act as abstract interfaces from which all concrete synopses inherit) (`ICardinality, Synopsis, IQuantileEstimator`) that define three important functions that are used across all different synopsis types.

6.1 Generic Functions' Descriptions

These essential functions are the following:

- `add_item`: Adds a data item into the synopsis structure.
  - In probabilistic sketches: it typically increases internal counts or sets bits.
  - In sampling: it adds the item into a heap.
- `add`: Similar with `add_item`, with the difference that it adds a chunk of data into the synopsis structure, rather than a single item.
- `estimate`: It retrieves an approximate answer depending on the synopsis type (e.g. presence of an item, cardinality estimation)
- `merge`: Combines two or more synopses with compatible configurations into one.

6.2 Classes Used Within the Synopses

6.2.1 BloomCalculations

The `BloomCalculations` class provides methods and data for optimizing and evaluating **Bloom Filter** configurations, particularly focusing on minimizing false positive rates.

Summary

- **Constants**: Defines bounds on the number of hash functions (`minK & maxK`) and buckets per element (`minBuckets & maxBuckets`), with a precomputed table (`optKPerBuckets`) giving the optimal number of hash functions for various bucket counts.
- `probs` Table: A 2D list of empirically precomputed false positive probabilities for combinations of bucket counts and hash functions.
- `computeBestK(bucketsPerElement)`: Returns the optimal number of hash functions (`K`) for a given number of buckets per element using the `optKPerBuckets` table.
- `computeBucketsAndK(maxFalsePosProb)`: Finds the smallest configuration (buckets and hash functions) that ensures the false positive probability stays below a given threshold.

- `getFalsePositiveProbability(bucketsPerElement, hashCount)`: Calculates the theoretical false positive probability for a **Bloom Filter** given the number of buckets per element and hash functions.
- Inner Class `BloomSpecification`: A simple container for the **Bloom Filter** parameters `K` and `bucketsPerElement`.

6.2.2 MurmurHash

The `MurmurHash` class provides multiple static methods for computing non-cryptographic hash values of different data types (integers, floats, strings, byte arrays, NumPy arrays, and Dask arrays), primarily implementing 32-bit and 64-bit versions of the **MurmurHash** algorithm [40]. Key methods include:

- `hash1(o, seed)`: Determines the type of input `o` and delegates hashing to the appropriate method (`hash_long()`, `hash_long_array()`, `hash_long_array_np()` or `hash()`).
- `hash_long()`, `hash_long_array()`, `hash_long_array_np()`: These methods perform 32-bit hashing on numbers, Dask and NumPy arrays respectively, using a simplified MurmurHash.
- `hash(data, length, seed)`: Implements the core 32-bit MurmurHash for byte arrays.
- `hash64(obj)` and `hash64_bytes(data, length, seed)`: Implement the full 64-bit **MurmurHash2** algorithm, designed for hashing strings, bytes, or any object (by converting it to string).

6.2.3 Lookup3Hash

The `Lookup3Hash` class implements a 64-bit variant of Bob Jenkins' Lookup3 hash algorithm [13], adapted for both single strings and batches of strings using Dask arrays.

- `lookup3ycs64(s, start=0, end=None, initval=-1)`: Computes a signed 64-bit hash of a single Unicode string, using bit shifts and `XOR`s.
- `lookup3ycs64_dask(strings, start=0, ends=None, initval=-1)`: Extends the same hashing logic to a Dask array of strings, instead of a single Unicode string.

6.2.4 RegisterSet

The `RegisterSet` class manages a compact dask array which represents the packed register values (`self.M`). Each register is 5 bits (`REGISTER_SIZE = 5`), and multiple registers are packed into $2^6 = 64$-bit words (`LOG2_BITS_PER_WORD = 6`).

- `__init__(count, initial_values=None)`: Initializes register storage for a given count. Uses either zeroed or provided initial values.
- `get_bits(count)`: Computes required bit count.
- `get_size_for_count(count)`: Computes required number of storage words.
- `set(position, value)`: Sets a 5-bit register at a position using bit manipulation.
- `get(position)`: Retrieves a register value using bit masking and shifting.
- `retrieve_registers(positions_count)`: Retrieves all registers' values for a range of positions efficiently.
- `update_if_greater(position, value)`: Updates a register in `position` if the new `value` is greater than the current one.
- `merge(that)`: Merges another `RegisterSet` into the current one, keeping maximum values at each register.
- `read_only_bits()`: Returns the Dask array for external, read-only use.
- `bits()`: Returns the current Dask array representing the state.

6.3 Interfaces implemented by the Synopses

6.3.1 Synopsis

`Synopsis` is implemented by two sketching data structures (`CountMinSketch` & ConservativeAddSketch), one probabilistic data structure used for fast membership testing (`BloomFilter`), two data structures which perform different types of sampling (`WeightedPrioritySampler & PrioritySampler`) and by one data structure which accumulates and merges Discrete Fourier Transforms (DFTs) of fixed-length signal chunks (`PartialDFTAccumulator`). All of these classes are Python implementations of the respective algorithms with the same names explained in **Chapter 2**.

Its purpose is to serve as a base interface for those data synopsis classes and to support them with the following functions:

- `generate_synopsis_id()`: provides a unique `synopsis_id` for each instance, using a global counter: `count`.
- `__init__(self, key_index = None, value_index = None)`: Initializes `synopsis_id`, `key_index`, and `value_index`. `value_index` is a value that a `Synopsis`'s subclass may add into its memory and `key_index` is used to determine the index where it may add that value into using hash functions.
- `get/set_synopsis_id, get/set_key_index & get/set_value_index`: getter and setter functions for the variables initialized in `__init__()`.
- `get_hash_count()`: returns `hash_count` which is defined in Synopsis' subclasses.

Abstract Core Methods

The functionality of abstract classes `add_item(key_index, value_index)`, `add(key_index, value_indexes), estimate(key_index) & merge(other_synopsis)` is explained in section **6.1**.

6.3.2 ICardinality

`ICardinality` is implemented by Python implementations (classes) of probabilistic cardinality estimation algorithms used to estimate the number of distinct elements in large datasets using minimal memory. These classes are `LinearCounting, LogLog, HyperLogLog, HyperLogLogPlus` and `AdaptiveCounting` which is `LogLog`'s subclass and they have the same name with their respective probabilistic cardinality estimation algorithms explained in **Chapter 2**.

`ICardinality` provides methods to add a data item into the classes' memory (`add(o: object)` & `add_hashed(hashed)`), merge many instances of compatible classes with each other(`merge(*estimators: 'ICardinality')`) and to estimate the total number of unique elements in the stream (cardinality estimation)(`estimate()`). It also provides methods to calculate the size of the classes' memory in bytes (`sizeof()`) and serialized representations of the classes (`get_bytes()`)

6.3.3 IQuantileEstimator

`IQuantileEstimator` is implemented by the `QDigest` class, which is a data structure used for approximate quantile estimation over a stream of integers. It is a Python implementation of **QDigest** algorithm explained in **Chapter 2**. `IQuantileEstimator` provides methods to insert data (`add(value)`), merge two instances (`merge(qdigest_a, qdigest_b)`) and to estimate the total inserted items (`estimate()`).

6.4 Methods Descriptions per Synopsis

The incorporated synopses and the specific functionality of their core methods are as follows:

6.4.1 Bloom Filter

- **Initialization** (`_init_()`): Can be created with a target false positive rate (`maxFalsePosProbability`) or a specific number of `bucketsPerElement`. It uses `BloomCalculations`'s functions: `computeBestK() & computeBucketsAndK()` to determine the optimal number of hash functions and buckets.
- **Bit Array**: Uses a Dask boolean array (`da.zeros`) as the filter bit array.
- `get_hash_buckets(key, hash_count = None, max_value = None)`: uses double hashing (`MurmurHash.hash()`) to compute the bit positions for storing/checking elements in the filter bit array.
- `add_item(key), add(keys: np.ndarray)`: hashes key / keys k times (using `get_hash_buckets()`) and sets the corresponding bits in the filter.
- `estimate(key)`: checks all k corresponding bit positions. Returns `True` if all are set, `False` otherwise. If `True` is returned, then the element is assumed to be present.
- `merge(other)`: merges Bloom filters using bitwise `OR`.
- `clear()`: Resets all filter bits to 0.
- `empty_buckets()`: Counts the number of unset bits (empty buckets).
- `always_matching_bloom_filter()`: creates a special filter (using `da.ones`) that always returns true (for testing use).

6.4.2 Count Min Sketch

- Maintains a 2D array (`table`) of counters with `depth` being the number of hash functions it holds and `width` being the number of buckets per row.
- Can be **initialized** using error tolerance (`eps`) and confidence level (`confidence`) or directly by using `depth` and `width`.
- `get_hash_buckets(key)`: Similarly as **Bloom Filter**, it returns the hash bucket indices for a key using double hashing with `MurmurHash.hash()`.
- `add_item(key_index, value_index), add(key_index, value_indexes)`: it adds `value / values` in d hashed positions (using `get_hash_buckets()` again) of the array of counts the sketch holds.
- `estimate(key)`: returns the minimum count across all rows for the hashed positions.
- `merge(other)`: it merges Count Min Sketch instances using element-wise addition of counter arrays.

6.4.3 Conservative Add Sketch

- **Initialization** (`_init_()`): inherits all the initialization procedures of `CountMinSketch`, including variables `table, depth, width, eps & confidence`. `add_item(key_index, value_index), add(key_index, value_indexes)`: updates only the counters whose current values match the minimum across all hash functions (`get_hash_buckets()`) for a key / keys.
- `get_hash_buckets()`, `estimate(key)` and `merge(other)` methods are inherited from `CountMinSketch`.

6.4.4 Partial DFT Accumulator

- `add(chunk, value=None)`: Computes the DFT of a chunk and adds it to the internal sum (`dft_sum`). It also checks that all chunks are of the same length.
- `estimate()`: returns the summed DFT of the entire signal.
- `merge(other)`: merges other **PartialDFTAccumulator** instances by summing their `dft_sum` values. Verifies that all accumulators have DFTs of the same length.

6.4.5 Weighted Priority Sampler

- Items are stored in a heap as (`priority, value, weight`) which simulates a min-heap.
- `add_item(value, weight=1.0), add(chunk)`: computes priority / priorities as $\frac{-log(U)}{weight}$, assigns it into an item / a chunk of items and it keeps in the heap the k items with the smallest priorities.
- `estimate()`: can estimate the weighted mean of the sampled values using the stored weights and values.
- `merge(other)`: combines samples and keeps top-k based on priority.

6.4.6 Priority Sampler

- Similarly with its superclass `WeightedPrioritySampler`, items are stored in a heap as (`neg_priority, item`), which simulates a min-heap.
- `add_item(self, item), add(chunk)`: assigns random priority into an item / a chunk of items and it keeps in the heap the k items with the largest priorities.
- `estimate()`: returns the indices of sampled items.
- `merge(other)`: is inherited from Weighted Priority Sampler.
- `reservoir_priority_sampling()`: splits large datasets into chunks and applies sampling on them, in parallel using `dask.array.map_blocks`, and returns the indices of sampled items.
- `get_sample()`: Returns the sampled items themselves.

6.4.7 Linear Counting

- Uses a Dask boolean bit array (`da.zeros`) to track hashed objects.
- `add_item(obj), add(obj)`: it hashes *obj* using `MurmurHash`'s 32-bit hashing (`MurmurHash.hash1()` function), then it maps the result / results into a position / positions in the bit array $map[m]$ and it sets it / them to 1.
- `estimate()`: it counts the number of zeros in the bit array (V) and then it computes the cardinality of the distinct elements: $\hat{n} = -m * ln(V/m)$, where m is the number of bits.
- `merge(other)`: merges instances of Linear Counting performing bitwise `OR` between their bitmaps.
- `get_utilization()`: Measures how full the bit array is (proportion of bits set).
- `map_as_bit_string()`: returns a human-readable binary string of the bitmap.
- `get_bytes()`: returns the map as a NumPy byte array
- `sizeof()`: returns the size of the map (`self.map`) in bytes.

6.4.8 LogLog

- Uses a bit-based register array M of size 2^k, where k controls accuracy and memory usage.
- Maintains a gamma correction factor (C_m) from a predefined table to improve estimation accuracy.
- `add(obj)`:
  - Hashes `obj` to 32 bit integers $h(x)$, using `MurmurHash`,
  - Uses the first k bits to select indexes j,
  - Through `rho(x, k)`, it computes $r = \rho(h(x))$, the positions of the first 1 in the remaining $32 - k$ bits of the hashed values.
  - Updates the table in positions j where $r > M[j]$: $M[j] = \max(M[j], r)$.

- `estimate()`: it computes $\bar{R} = \frac{1}{m}\sum_{j=0}^{m-1} M[j]$, which is the average of the register values and then the estimated cardinality: $\hat{n} = C_m \cdot 2^{\bar{R}}$.
- `merge(other)`: merges instances of LogLog by computing the element-wise maximum of their arrays.
- `get_bytes()`: returns the raw byte array.
- `sizeof()`: returns the size of the register array M.

6.4.9 Adaptive Counting

- `rho(x, k)`: computes the position of the first 1 in the remaining $64 - k$ bits of a the hashed value x.
- `add_item(obj), add(obj)`: follows similar procedure with its superclass: `LogLog`, with the difference that it uses `Lookup3Hash`'s 64-bit hashing (`Lookup3Hash.lookup3ycs64()` function), and it maintains count of empty buckets $\beta = b_e/m$ and that `add_item()` does this procedure for a single item rather than a chunk of items.
- `estimate()`:
  - If the ratio of empty buckets (β) exceeds a threshold ($\beta_s = 0.0513$), it uses Linear Counting's estimate: $\hat{n} = -m * ln(\beta)$
  - Otherwise, it uses LogLog estimate: $\alpha_m * m^2 * 2^{\frac{1}{m} * \sum_{j=1}^{m} M(j)}$
- `merge(other)`: Same as LogLog: register-wise max merge

6.4.10 Hyper Log Log

- **Initialization** (`__init__(self, rsd = None, log2m = None)`): accepts either `rsd` (relative standard deviation) or `log2m` (precision level); calculates internal parameters like number of registers (`m`) and correction factor (`alpha_mm`) using `_get_alpha_mm(log2m,m)` function.
- `add_item(o), add(o)`: follows similar procedure with **LogLog**, with the difference that M is a set of registers (`RegisterSet`) rather than an array and the set of registers is updated using `RegisterSet.update_if_greater()` function.
- `estimate()`: computes the harmonic mean of the transformed register values to estimate cardinality, applying linear counting (via `_linear_counting()` function) for small cardinalities.
- `merge(other)`: for each register, it takes max value between current and other, using `add_all(other)` as well internally.
- `get_bytes()`: Serializes `RegisterSet` and `log2m` to bytes
- `sizeof()`: Returns `RegisterSet`'s size in bytes needed for serialization.

6.4.11 Hyper-Log-Log-Plus

The `HyperLogLogPlus` class is a Python implementation of the **HyperLogLog++** algorithm, a probabilistic cardinality estimation algorithm used to estimate the number of distinct elements in large datasets using minimal memory. It implements the `ICardinality` interface.

- **Initialization** `__init__(self, p: int, sp: int)`: Determines initial storage mode (sparse or normal), depending on the values of normal (`p` (defines register count = 2^p)) and sparse precision and (`sp` (if it is negative, `Format` switches into normal mode and into sparse otherwise)). Also, it sets up registers (`RegisterSet`), thresholds and other data structures like temporary arrays (e.g. `tmp_set`).
- `add_item(obj), add(chunk)`: follows similar procedure with **HyperLogLog**, with the difference that it uses `MurmurHash`'s 64-bit hashing (`MurmurHash.hash64()`). Also when **HLL++** is on `Sparse Format`, the additions are done in `sparse_set`. Only when `Format` is switched into `Normal`, the `RegisterSet` is used for updates.

- `estimate()`:
  - Computes raw estimate $E := \alpha_m * m^2 * \left(\sum_{j=1}^{m} 2^{-M[j]}\right)^{-1}$ ($\alpha_m * m^2$ constant is calculated via **HLL**'s `_get_alpha_mm(p,m)`),
  - for small cardinalities raw estimate tends to be biased, therefore it is corrected accordingly: $E^* = E - bias(E)$, using `get_estimate_bias(estimate, p)` and `get_bias(nearest_neighbors, p)` functions,
  - **Linear Counting** is selected to estimate cardinality (calculated via **HLL**'s `_linear_counting()`) when there are empty registers and the empirically defined threshold is: $T > E$. Otherwise, E^* is used.
  - Large cardinalities are calculated this way: $\hat{n} = -2^{64} \cdot \ln\left(1 - \frac{E}{2^{64}}\right)$
- `merge(other)`: Same as **HyperLogLog**.
- `convert_to_normal(self)`: switches `Sparse` into `Normal` mode by converting `tmp_set` and `sparse_set` into registers.
- `merge_temp_list`: Decodes all items from `tmp_set` into `sparse_set` and it converts into `Normal` mode if the `sparse_set` exceeds the threshold. It is invoked in `estimate()` if `Format` is `Sparse`.
- `decode_run_length(k)`: It is used in `convert_to_normal()` to parse a hashed value stored in the old `sparse_set` and add it into the new `RegisterSet` in order to successfully convert into `Normal` mode.
- `get_nearest_neighbors(distance_map)`: It is used in `get_estimate_bias()` to locate the closest precomputed bias estimates for interpolation.
- `get_bytes()`: Returns the serialized representation of the estimator.
- `sizeof()`: Same as **HLL**.

6.4.12 QDigest

Key Variables

- `compression_factor`: balances memory usage and accuracy.
- `capacity`: Determines the range of values (power of 2).
- `node2count`: A binary tree (array-based) storing counts per node.
- `size`: Total number of elements added.

Main Functionalities

- `value2leaf(x), leaf2value(id)`: These functions convert between data values and leaf node IDs.
- `is_root, is_leaf, sibling, parent, left_child, right_child`: These functions are tree navigation helpers.
- `compress_upward(node)`: merges nodes' counts (`node2count`) from leaves up the tree when under threshold.
- `compress_fully()`: traverses the tree bottom-up to perform complete compression.
- `rebuild_to_capacity(new_capacity)`: resizes the tree using `new_capacity` and it remaps existing counts (`node2count`).
- `add_item(value), add(values)`:
  - `value, values` is / are converted into a leaf node / leaf nodes,

  - the count at the corresponding node / nodes is / are incremented, the structure then performs upward compression: it pushes small counts to parent - parents / sibling - siblings if their combined total is low

- `estimate()`: it returns the sum of all node counts (the total inserted items)
- `merge(other)`:
  - it aligns capacities using `rebuild_to_capacity()`,
  - sums counts for each node,
  - calls `compress_fully()` to restore **Q-Digest** invariants
- `print_node2count()`: Prints the dictionary of non-zero node counts (`node2count`).

The following is the full class diagram of our architecture:

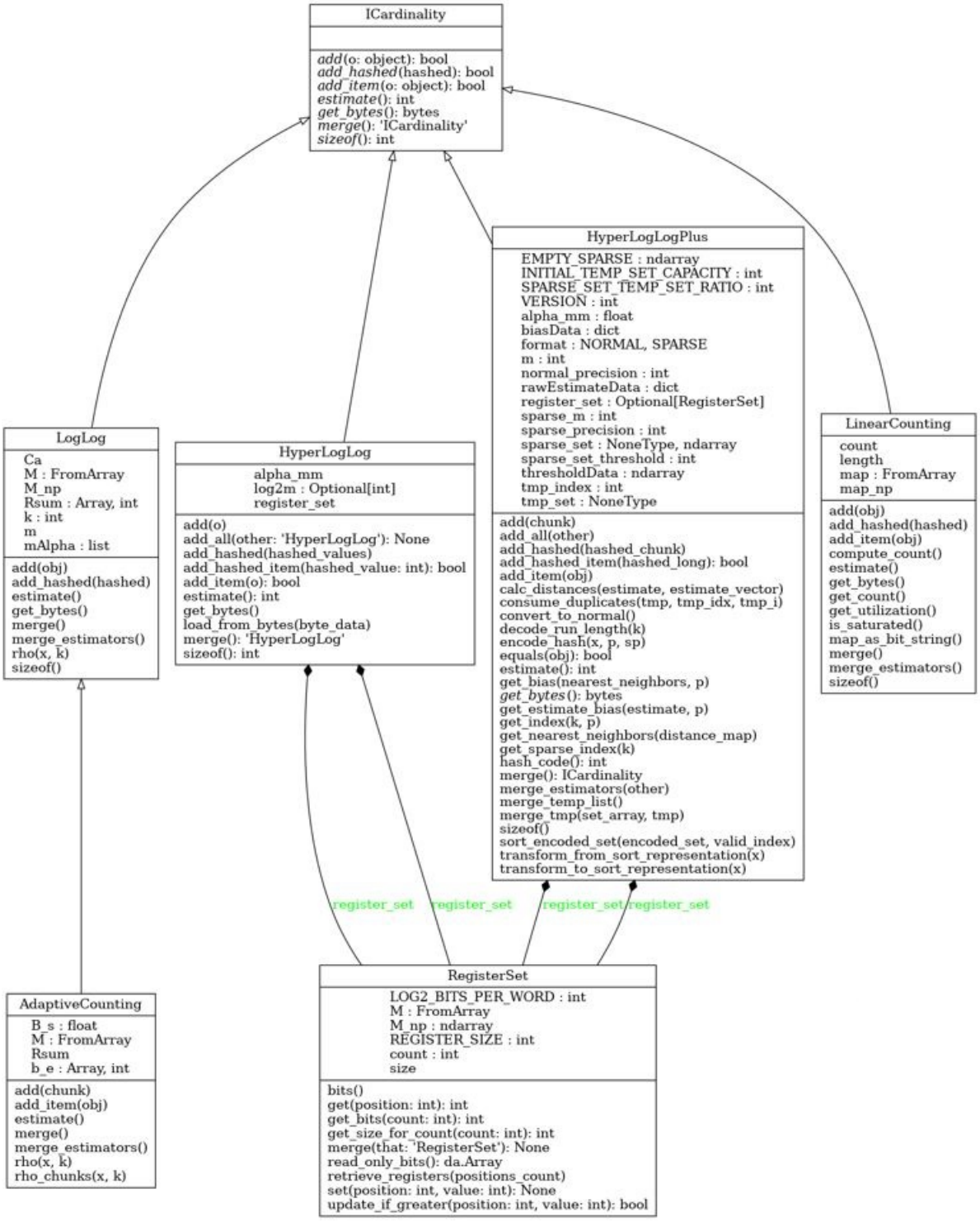

Figure 6.1: `ICardinality`'s subclasses

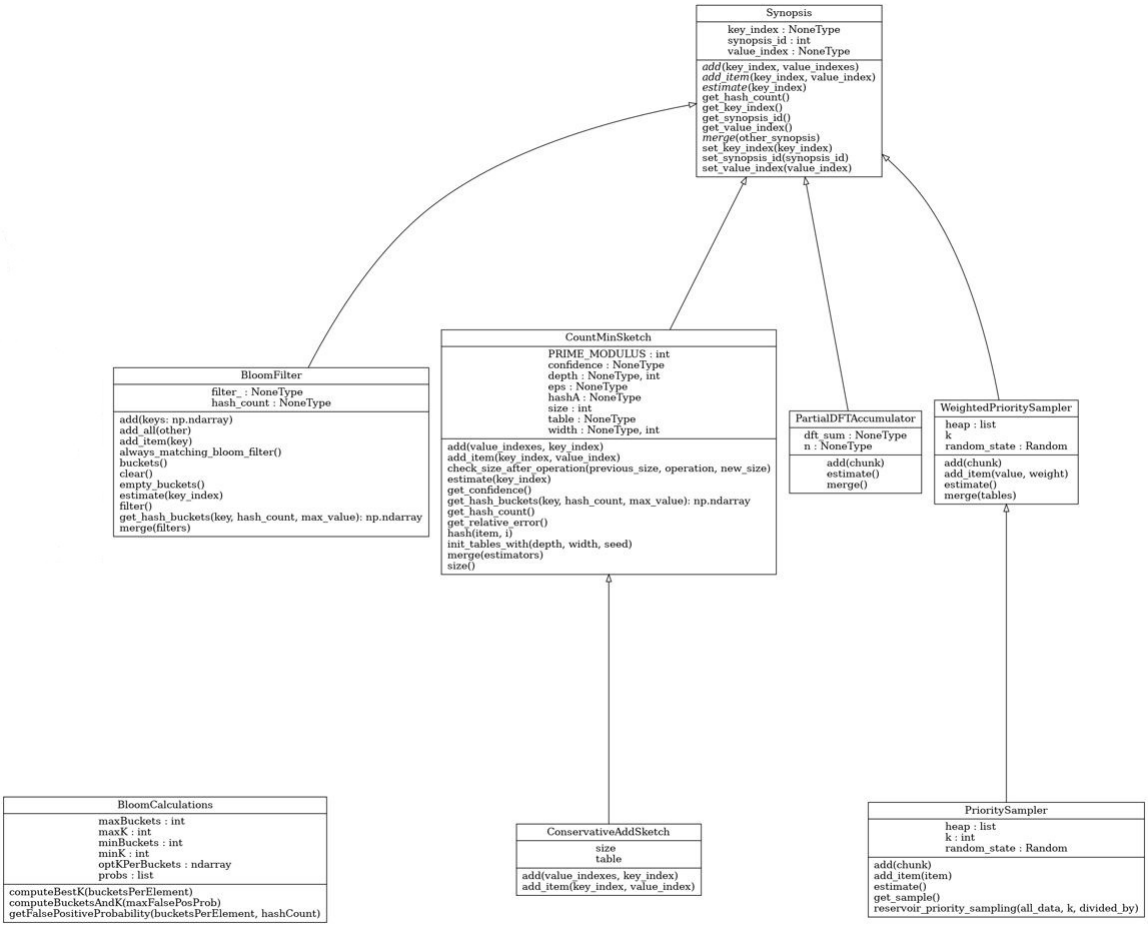

Figure 6.2: `Synopsis`'s subclasses

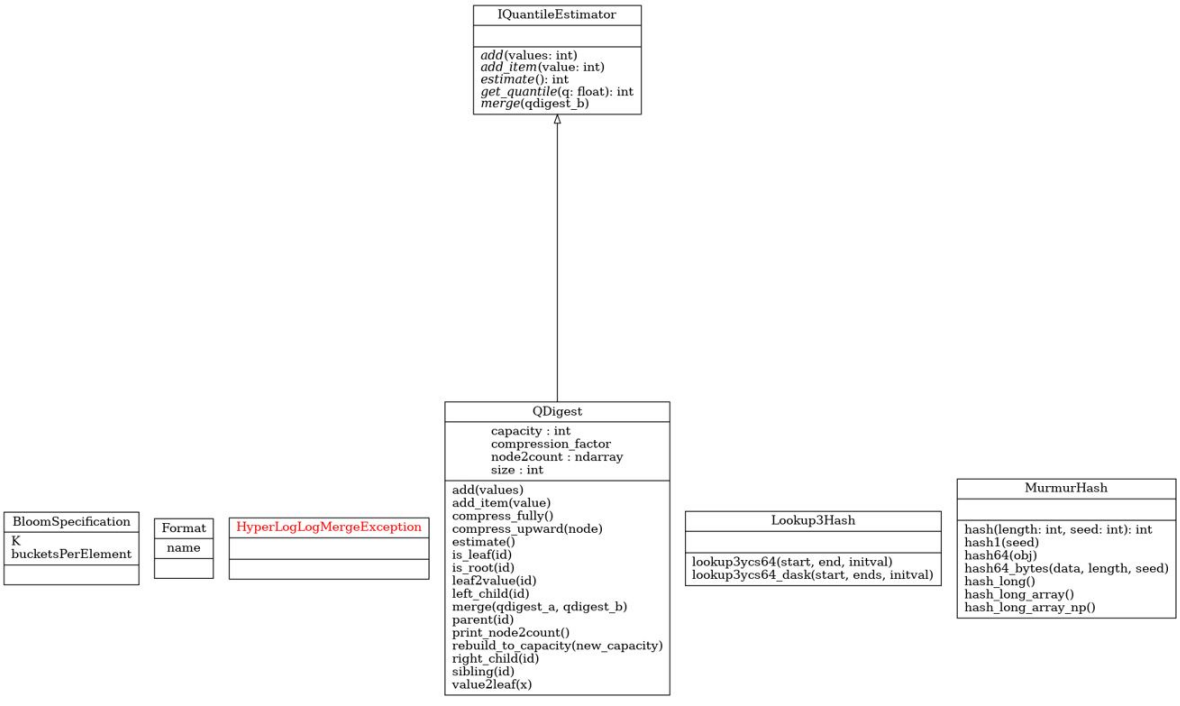

Figure 6.3: `QDigest` & smaller classes

6.5 How parallelism is achieved in core Synopses

Parallelism is achieved through the testing code of the stream synopsis engine, where large portions of data are added into the synopses.

6.5.1 Parallelism in HyperLogLog

Here is a breakdown of how parallelism is achieved and utilized in `HyperLogLog`:

Coding Example 6.1: `HyperLogLog`**'s Random Data Generation and Chunking**

```
elements_dask = da.arange(num_elements, chunks=num_elements // 4)
```

Dask arrays (`da`) are used to generate an array of: `num_elements`. Then, the array is split into four chunks, in order to distribute each chunk into a worker (CPU core). This setup allows Dask to distribute these four chunks across Google Colab's available CPU cores, which are four as well, in the next step.

Coding Example 6.2: `HyperLogLog`**'s Parallel Block Processing**

```
M_nps_splitted = elements_dask.map_blocks(hll.add, ...
    dtype=elements_dask.dtype).compute()
```

`map_blocks` applies `hll.add` in parallel on each chunk. Each chunk is processed independently and concurrently using one thread/process per worker under the hood. Then, each worker returns the synopsis' table after its respective chunk of data is added into it. This is the most time-efficient step, as `dask.array.map_blocks` distributes chunks of data items into multiple cores (workers), which significantly reduces execution times. This is how it looks in Dask's Dashboard:

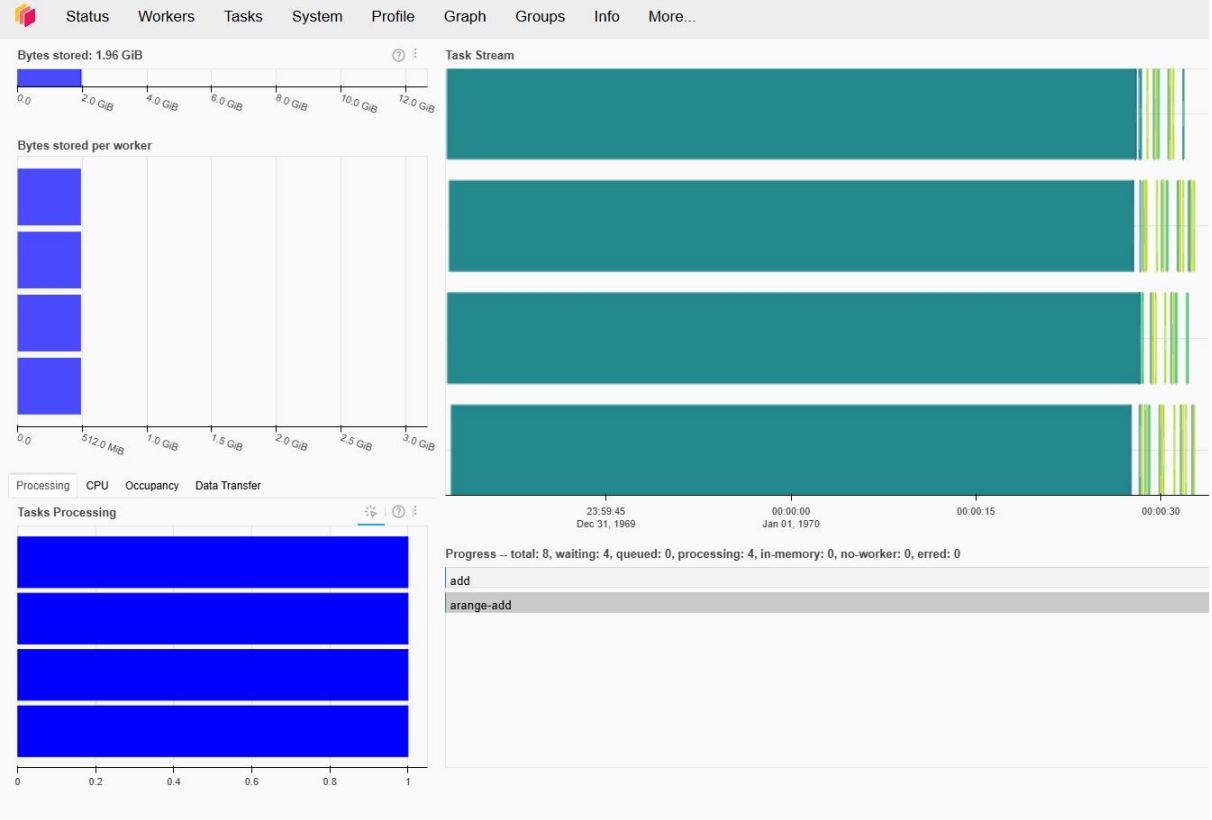

Figure 6.4: The Dask dashboard during the addition process of `HyperLogLog`

In Figure 6.4, it can be noticed that this step is memory efficient as well, because the 1.96 GB that are stored, are distributed across each chunk in the "Bytes stored per worker" field (four chunks are used, so 512 MB are stored in each one, as it is seen in Figure 6.4). Moreover, via the "Tasks Processing" field, it is proved that each worker executes independently and concurrently, as all of them execute at the same time.

Then, each output table: `M_nps_splitted` is added into separate partial sketches:

Coding Example 6.3: `HyperLogLog`'s `M_nps_splitted` is added into separate partial sketches

```
M_nps_size = len(M_nps_splitted) // 4

hlls = []
for i in range(0, len(M_nps_splitted), M_nps_size):
    hlls.append(HyperLogLog(rsd=k))
    print("M_np ", hlls_index, " : ", M_nps_splitted[i:i+M_nps_size])
    maps.append(M_nps_splitted[i:i+M_nps_size])
    hlls[hlls_index].register_set.M_np = M_nps_splitted[i:i+M_nps_size]
    hlls[hlls_index].register_set.M = ...
        da.from_array(hlls[hlls_index].register_set.M_np, chunks="auto")

    hlls_index += 1
```

Finally, these partial sketches are merged into one, to produce the final synopsis:

Coding Example 6.4: Mergence of `HyperLogLog`'s partial sketches

```
HLL_Final = hlls[0].merge(*hlls[1:4])
```

Coding Example 6.5: `HyperLogLog`'s Cardinality Estimation

```
estimated_cardinality = HLL_Final.estimate()
```

The function used in coding example 6.5 calculates the synopsis' cardinality as explained above in `HyperLogLog` section, using some parallel processes. Firstly, it performs parallel function mapping using `dask.array.map_blocks`, which applies `retrieve_registers()` in parallel across chunks of `self.register_set.count`, similarly as in "Parallel Block Processing" example 6.2. This wraps the function in a Dask task graph, enabling multi-core execution. When `.compute()` is called, the previously defined `map_blocks` task graph is executed in parallel, retrieving all register values.

Coding Example 6.6: Parallel `RegisterSet`'s values retrieval

```
all_values = da.map_blocks(self.register_set.retrieve_registers, ...
    self.register_set.count, dtype=int)
vals = all_values.compute()
```

Secondly, it builds and computes a Dask graph to sum transformed register values (HLL formula) by spliting the computation across chunks and by leveraging multiple cores:

Coding Example 6.7: Summation of transformed `RegisterSet`'s values (HLL formula) using Dask

```
register_sum = da.sum(1.0 / (1 << all_values)).compute()
```

6.5.2 Parallelism in Priority Sampler

Here is a breakdown of how parallelism is achieved and utilized in `PrioritySampler`:

Coding Example 6.8: `PrioritySampler`'s Dask Arrays for Chunked Computation

```
zeros = da.zeros((num_elements), chunks=(num_elements // 4))
all_data_indexes = da.arange(num_elements, chunks=(num_elements // 4))
data = da.stack([all_data_indexes, zeros], axis=1)
data = data.rechunk(num_elements // 4, 2)
```

In coding example 6.8, `da.zeros` and `da.arange` create Dask arrays with chunk sizes: `chunks=(num_elements // 4)`, which tells Dask to split the array into multiple partitions, in order to be processed in parallel across CPU cores or workers. These arrays are not computed immediately and Dask builds a task graph for deferred execution.

Coding Example 6.9: `PrioritySampler`'s Parallel Block-wise Processing

```
1  top_ks_splitted = data.map_blocks(self.add, dtype=all_data.dtype).compute()
```

`map_blocks` applies the `self.add` function to each chunk (block) of the Dask array independently. This is where parallelism actually happens: each block is processed independently and concurrently. The `.compute()` triggers execution of the task graph. Then, each worker returns the top-k data of each chunk into a list of top-ks: `top_ks_splitted`. Similarly with `HyperLogLog`, this is the most time-efficient step, as `dask.array.map_blocks` distributes chunks of data items into multiple cores (workers), which significantly reduces execution times. This is how it looks in Dask's Dashboard:

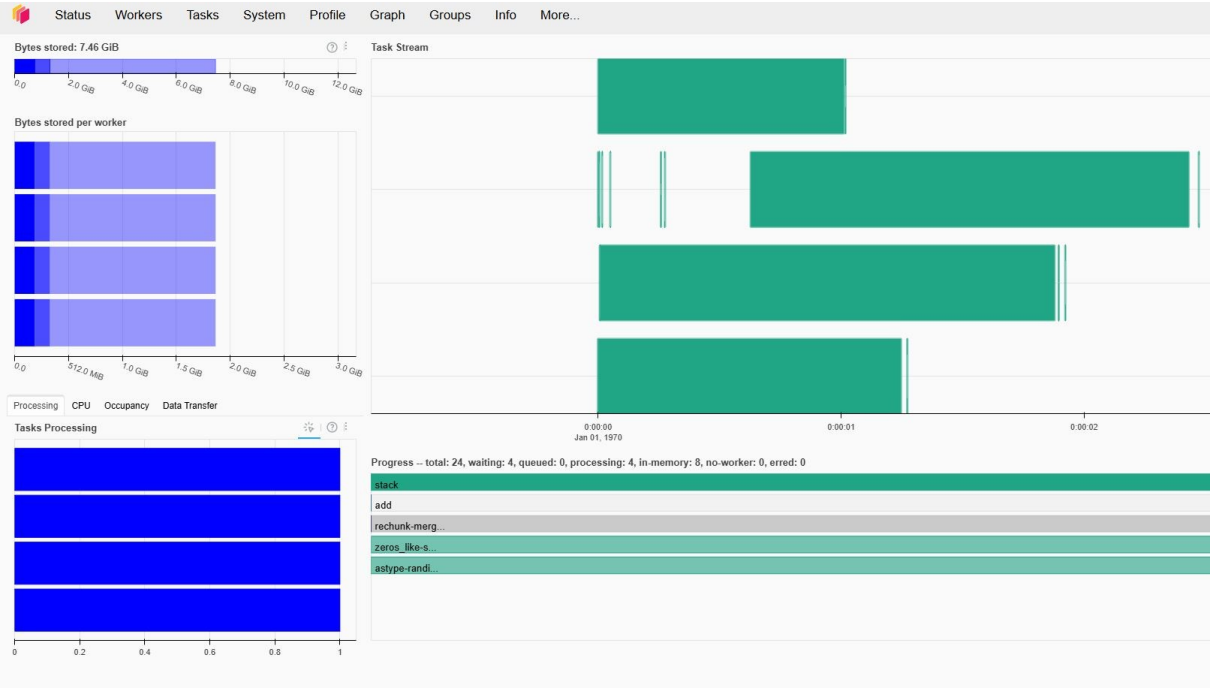

Figure 6.5: The Dask dashboard during the addition process of `Priority Sampler`

Like `HyperLogLog`, it can be noticed that this step is memory efficient as well, because the 7.46 GB that are stored, are distributed across each chunk in the "Bytes stored per worker" field (four chunks are used, so 1.87 GB are stored in each one, as it is seen in Figure 6.5). Moreover, via the "Tasks Processing" field, it is proved that each worker executes independently and concurrently, as all of them execute at the same time.

Coding Example 6.10: `PrioritySampler`'s Final top-k Data

```
1  for i in range(0, (top_ks_splitted.shape[0]), self.k):
2      top_ks.append(top_ks_splitted[i : i + self.k])
3
4  merged_PS = self.merge(top_ks)
```

Finally, using coding example 6.10, this list of top-ks is merged into one and out of them the top-k data are kept.

6.5.3 Parallelism in QDigest

Here is a step-by-step breakdown of how parallelism is implemented in `QDigest`:

Coding Example 6.11: `QDigest`'s Dask Arrays for Chunked Computation

```
1  a_samples_dask = da.from_array(a_samples, chunks=len(a_samples) // divided_by)
```

In coding example 6.11, a Dask array is created with chunk sizes: `chunks=(num_elements // divided_by)`, which tells Dask to split the array into multiple parts (With `divided_by = 4`, each sample array is split into 4 chunks) that can be processed in parallel across CPU cores or workers.

Coding Example 6.12: `QDigest`'s use of `dask.array.map_blocks` for Parallel Processing

```
1  node2counts_a = a_samples_dask.map_blocks(a.add, dtype=a.node2count.dtype).compute()
```

In coding example 6.12, `map_blocks` applies the `self.add` function to each chunk (block) of the Dask array independently. This is where parallelism actually happens: each block is processed independently and concurrently across available cores. The `.compute()` triggers actual execution - prior to this, Dask builds a task graph. Then, each worker returns the nodecount of each chunk into a list of nodecounts: `node2counts`. Similarly with `HyperLogLog` and `PrioritySampler`, this is the most time-efficient step, as `dask.array.map_blocks` distributes chunks of data items into multiple cores (workers), which significantly reduces execution times. This is how it looks in Dask's Dashboard:

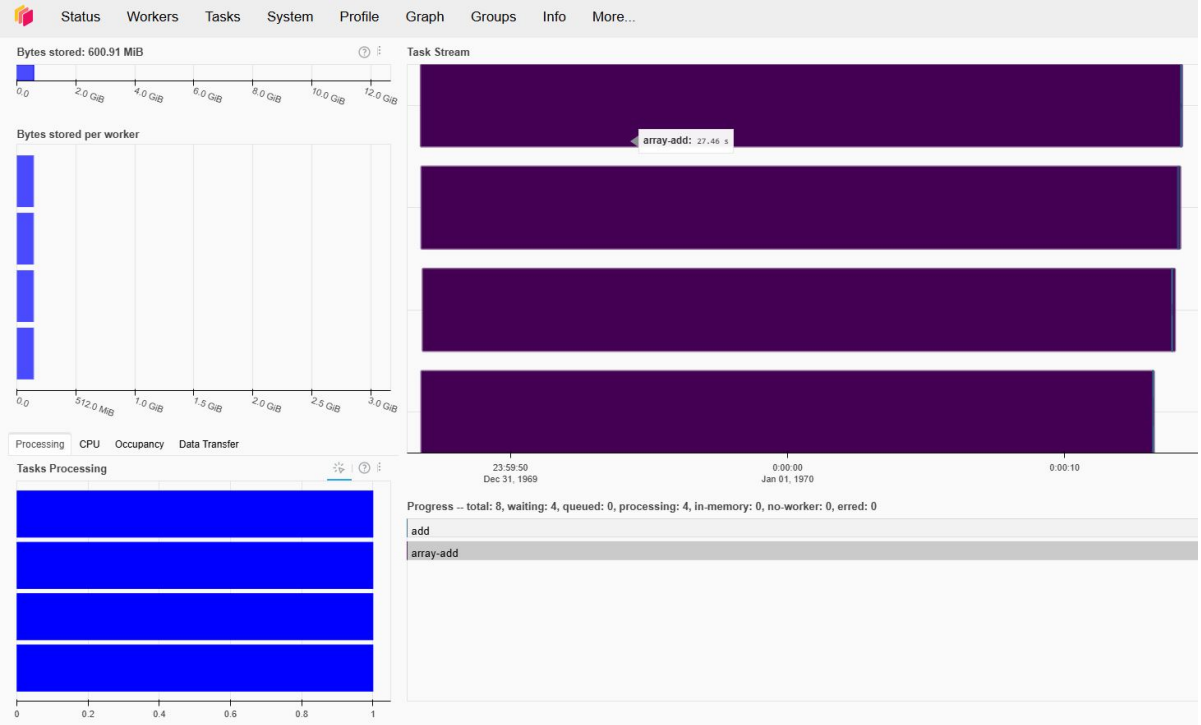

Figure 6.6: The Dask dashboard during the addition process of `QDigest`

Like `HyperLogLog` and `PrioritySampler`, it can be noticed that this step is memory efficient as well, because the 600 MB that are stored, are distributed across each chunk in the "Bytes stored per worker" field (four chunks are used, so 150 MB are stored in each one, as it is seen in Figure 6.6). Moreover, via the "Tasks Processing" field, it is proved that each worker executes independently and concurrently, as all of them execute at the same time.

After parallel processing, the data is gathered:

Coding Example 6.13: `QDigest`'s Manual Aggregation After Parallel Computation

```
qdigests_a = []
for i in range(divided_by):
    qdigests_a.append(QDigest(compression_factor))

j=0

for i in range(0, len(node2counts_a), len(a.node2count)):
    qdigests_a[j].node2count = node2counts_a[i:i + len(a.node2count)]
    j+=1
```

The results of the parallel blocks are split and assigned to individual `QDigest` instances (`qdigests_a`), in order to be set up for merging.

Coding Example 6.14: Mergence of `QDigest` instances

```
A = QDigest.merge(qdigests_a[0], qdigests_a[1])

for i in range(2,len(qdigests_a)):
    A = QDigest.merge(A, qdigests_a[i])
```

Finally, instances of `QDigest` are merged into the final instance: `A`, using `merge()`.

Chapter 7

The SuBiTO Framework

7.1 SuBiTO Preliminaries: Introduction to Bayesian Optimization

Bayesian Optimization (BO) [38] is a powerful strategy for optimizing objective functions that are expensive to evaluate and lack an analytical form. Unlike traditional optimization methods, BO treats the target function as a black-box and avoids assumptions such as smoothness or convexity. In the context of SuBiTO, BO is employed to efficiently search for the best training or parallelization configurations while minimizing the number of costly evaluations. This is particularly crucial for scenarios involving large-scale data streams or resource-constrained model training pipelines.

7.1.1 Surrogate Modeling with Gaussian Processes

Rather than evaluating the true objective function $f(x)$ exhaustively, BO builds a surrogate model that approximates $f(x)$ based on a limited number of observations. The most common surrogate used is the **Gaussian Process (GP)**, a non-parametric model that defines a distribution over functions:

$$f(x) \approx GP(\mu(x), k(x, x'))$$

Here, $\mu(x)$ is the mean value of the GP and $k(x, x')$ is the kernel function. The kernel function is the most crucial component of a Gaussian Process model, as the similarity between two data points is defined by it. A widely used choice is the **Radial Basis Function (RBF)** kernel:

$$k(x, x') = \sigma^2 * exp(\frac{-||x - x'||^2}{2l^2})$$

Here, the value of lengthscale l defines the velocity with which the function changes. Also, the output variance σ^2 quantifies how far the function diverges from its mean value. Therefore, this kernel defines smoothness and similarity between function evaluations.

7.1.2 Acquisition Functions

Once a surrogate is constructed, instead of uniformly observing the actual function, BO uses an **acquisition function** $\alpha(x)$ to determine the most promising next point to evaluate. Several acquisition functions are supported offering particular advantages, such as balancing **exploration** (evaluating points where there is large uncertainty about $f(x)$) and **exploitation** (evaluating points where the surrogate model predicts the actual value of $f(x)$ will be high), including:

- Expected Improvement (EI):

$$EI(x; \xi) = (\mu - f(x^+) - \xi)\Phi(\frac{\mu - f(x^+) - \xi}{\sigma}) + \sigma\phi(\frac{\mu - f(x^+) - \xi}{\sigma})$$

  where $f(x^+)$ is the best observed value so far, Φ is the standard normal CDF and ϕ the PDF. Hyperparameter ξ balances the exploration and exploitation of Expected Improvement. By assigning $\xi = 0$, the optimizer takes an exploitative approach, and for large values of ξ there is more exploration as if we have a smaller current best observation.

- Upper Confidence Bound (UCB):

$$\alpha_{UCB} = \mu_{GP}(x) + \kappa\sigma_{GP}(x)$$

- Lower Confidence Bound (LCB):

$$\alpha_{LCB} = \mu_{GP}(x) - \kappa\sigma_{GP}(x)$$

In UCB and LCB, the tunable parameter κ balances exploration and exploitation. $\mu_{GP}(x)$ is the mean of the prior and $\sigma_{GP}(x)$ is the marginal standard deviation of $f(x)$. In UCB, when κ is small, BO will choose points with high μ and when κ is large, it will favor the exploration of the search space.

Each of these acquisition functions guides the optimization loop by proposing points where the model is uncertain or expects improvement.

7.1.3 Bayesian Optimization Algorithm

The BO algorithm iteratively refines its estimate of the optimum using the following steps:

1. Initialization of the Gaussian Process with a few evaluations of $f(x)$.
2. Fitting of the GP to the observed data.
3. Maximizing of the acquisition function to select the next point:

$$x_{t+1} = argmax_x \alpha(x)$$

4. Evaluation of $f(x_{t+1})$ and update of the GP model
5. Repetition of steps 2–4 until the budget is exhausted or convergence is reached.

This cycle is illustrated in Figure 7.1, which shows how the surrogate model and acquisition function evolve over iterations to hone in on the global optimum.

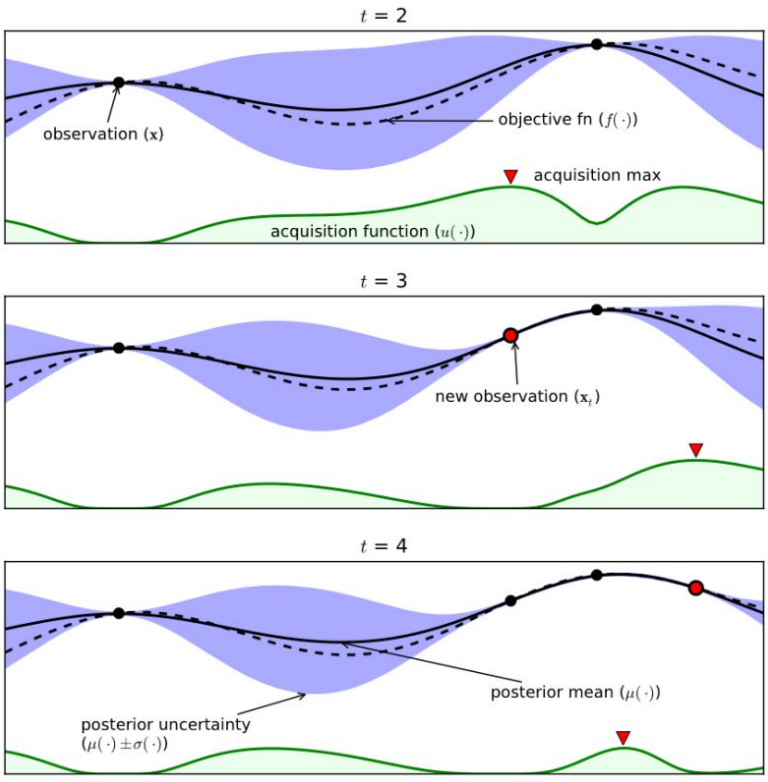

Figure 7.1: Bayesian Optimization progress in timesteps [28] [38]

7.2 SuBiTO Overview

SuBiTO [48] [50], which stands for Synopsis-Based Training Optimization, is a machine learning framework made for continuous real-time training of neural networks over big streaming data. It is designed to balance the training speed and accuracy of neural networks, enabling timely predictions without compromising model quality.

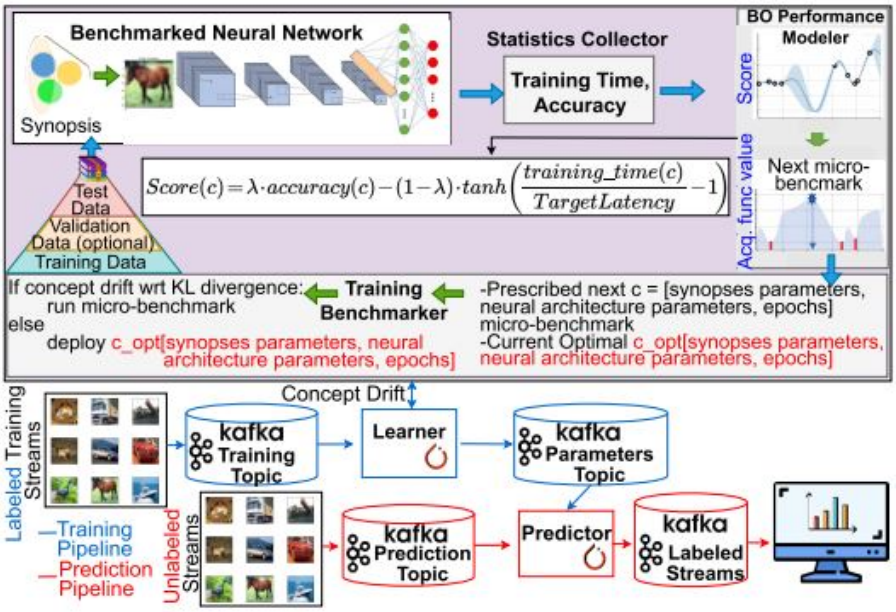

Figure 7.2: The SuBiTO architecture [48]

7.3 Key Concepts and Innovations

Traditional neural network training strategies struggle in streaming contexts due to:

- Unbounded data volumes,
- The need for frequent model updates,
- The volatility of input data distributions (concept drift),
- Limited time for retraining between updates.

SuBiTO addresses these issues by optimizing the following core components:

- Neural Network Architecture: Number, size, and type of layers (e.g., CNN, RNN),
- Data Input using Synopses of data: Compact summaries of data using sampling, dimensionality reduction, and sketching to reduce load,
- Training Epochs: Number of training iterations.

It operates in real time, using streaming frameworks (e.g., Apache Kafka) for ingesting labeled and unlabeled data. SuBiTO develops continuously updated models for predictions and detects concept drift to adapt on-the-fly.

7.4 Production Training and Prediction Pipelines

7.4.1 Training Pipeline (Blue Path in the Architecture)

It continuously trains neural network models using newly arriving labeled streaming data in near real-time from streaming platforms like Apache Kafka.

In more detail, it executes the following procedure:

1. The **Learner** receives streaming data from **Apache Kafka's Training Topic** that contains labeled examples (e.g., images with labels for moderation).
2. Instead of using all raw data, the **Learner** applies synopsis techniques (e.g., sampling, sketches, dimensionality reduction) to reduce data volume and speed up training.
3. The **Learner** trains the model using: a neural architecture (type, depth), a configured synopsis (e.g., sample size) and a set number of epochs, which are all determined by the **SuBiTO Optimizer** (after a concept drift or human-operator trigger).
4. After training, the updated model is published to **Apache Kafka's Parameters Topic**, which is then picked up by the **Prediction Pipeline** for inference.

7.4.2 Prediction Pipeline (Red Path in the Architecture)

Performs live inference using the most recently trained neural network model from the **Training Pipeline** on unlabeled streaming data.

1. Receives a real-time stream (e.g., images or videos) from **Apache Kafka's Prediction Topic**.
2. When the **Training Pipeline** finishes training, it shares the new model with the **Prediction Pipeline** through **Apache Kafka's Parameters Topic** and it immediately adopts this new model.
3. Then, the **Predictor** applies the current model to the incoming data received from **Apache Kafka's Prediction Topic**, to generate outputs like image classifications, content moderation flags or any other task relevant to the application.
4. The **Predictor**'s results (prediction accuracy, latency, etc.) are sent to the dashboard to let human operators monitor how the model is performing over time.

7.4.3 Summary of Production Training and Prediction Pipelines

Component	Role	Data Input	Key Actions
Training Pipeline	Trains & produces new models in real-time	Labeled stream (Kafka)	Applies synopses, trains NN & updates the model
Prediction Pipeline	Uses the latest model immediately for inference	Unlabeled stream (Kafka)	Loads the model, makes predictions & reports stats

Table 7.1: A summary of the 2 Pipelines (Roles at a Glance)

7.5 The SuBiTO Optimizer: The Computational Bottleneck

The SuBiTO Optimizer is a Bayesian Optimization (BO) engine which is triggered **automatically** when a concept drift (changes of data distribution) is detected (currently using the detection function: KL-divergence) or **manually** when a user intervenes (e.g. suspicious performance). Its purpose is to execute multiple model training trials under different parameter configurations to explore and suggest the best training configuration.

7.5.1 Optimization Strategy

Parameter Selection Using Bayesian Optimization

- The SuBiTO Optimizer uses Bayesian Optimization (BO) to intelligently select parameter sets that are likely to perform well.

- BO avoids brute-force grid search by being sample-efficient. Specifically, it chooses configurations probabilistically using one out of three BO acquisition functions, namely Lower Confidence Bound (LCB), Expected Improvement (EI) and Probability of Improvement (PI).

These guide the choice of configurations to test only limited trials, avoiding unnecessary computations.

Training and Evaluation Trials

For each selected trial, the optimizer trains a model using:

- The chosen architecture,
- Stream synopsis setting (e.g., 30% of data),
- A number of epochs.

After the training, the optimizer records the time the training took (**training time**) and how accurate the model got trained (**validation accuracy**).

Modeling the Trade-off Space

The results of these trials are fed into a Gaussian Process Regressor (GPR), which models the function:

$$f(architecture, synopses, epochs) \rightarrow (accuracy, training\_time)$$

GPR acts as a surrogate model that can estimate how any new parameter set will perform without retraining.

7.5.2 Scoring Function

SuBiTO evaluates parameter sets (c) with a composite score function:

$$Score(c) = \lambda * accuracy(c) - (1 - \lambda) * tanh(\frac{training\_time(c)}{TargetLatency} - 1)$$

where $\lambda \epsilon [0, 1]$ is the balancing factor between accuracy and training time.
This function provides explicit control over the accuracy vs delay trade-off.

7.5.3 Computational Bottleneck

The **SuBiTO Optimizer** is the computational bottleneck of the **SuBiTO Framework** for the following reasons:

- Running the optimizer involves retraining models multiple times under varying configurations, in order to evaluate performance, which is computationally expensive.
- Even with BO limiting the number of trials, these retraining runs remain costly.
- Each trial can involve frequent concept drifts, large data, and long epochs. As a result, this part dominates SuBiTO's computational workload.

Therefore, reducing the number of trials and speeding up GPR-based prediction are key to improving SuBiTO's overall responsiveness.

7.6 SuBiTO Dashboard

A web-based dashboard (built with Streamlit [47]) allows operators to:

- Set the system's parameter bounds, such as synopses compression ratio for load shedding, possible ranges of epochs, valid types of NN layers,
- Monitor live training/prediction metrics,
- Review the top-3 candidate models suggested by SuBiTO based on the current data stream,

- Manually or automatically (through a concept drift) trigger optimization runs.

The system is integrated with both TensorFlow [52] (for the **SuBiTO Optimizer**) and PyTorch [44](for the **Training and Prediction Pipelines**), and has been validated on real-world datasets like NSFW Detect [42] and UCF50 [46].

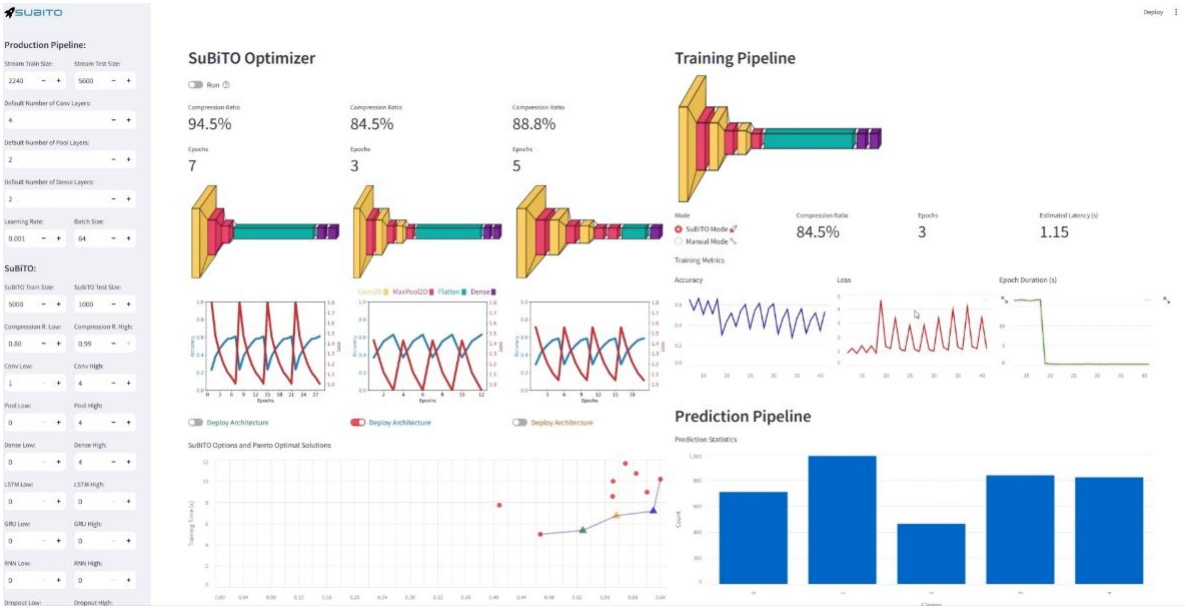

Figure 7.3: The SuBiTO Dashboard

Chapter 8

Applications - Experimental Evaluation

8.1 Maintaining Generic Stream Synopses

To evaluate the practical performance of the **Stream Synopses** developed in **Chapter 6** (Our Toolkit), we conducted a series of experiments using their corresponding testing codes, as described in **Section 6.5** (How parallelism is achieved). These experiments utilize artificially generated data and focus on measuring the impact of parallelism on execution time. Specifically, for each synopsis, we varied the number of parallel workers involved in the computation, leveraging `Dask`'s [11] distributed processing via `dask.array.map_blocks` [16], which splits the artificially generated data into chunks and assigns them to available workers.
The results in the following figures clearly demonstrate that increasing the level of parallelism significantly reduces execution time.

In each experiment, we utilized as many artificial data as the **Local Cluster** could handle, taking into consideration that we had limited computational resources on our disposal. We conducted the experiments in a simulated environment using **Google Colab Pro**, which provided 50GB of RAM and four cores.

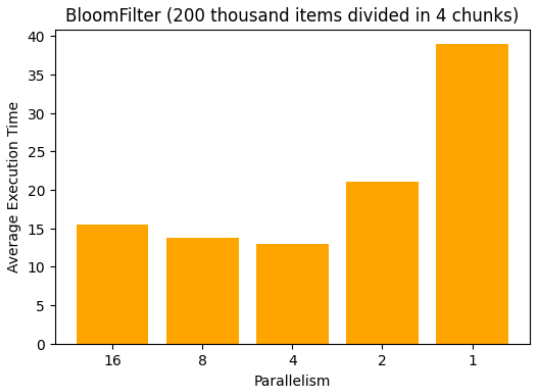

Figure 8.1: Execution times of experiments conducted with `BloomFilter`, leveraging increasing number of workers

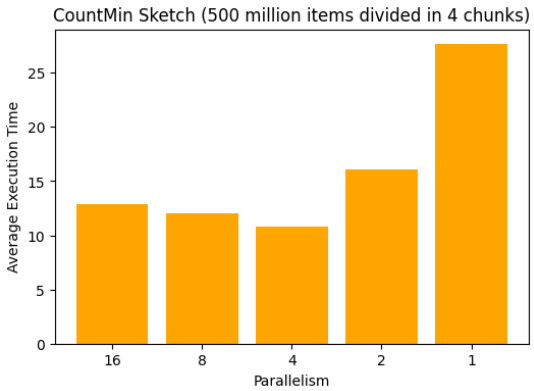

Figure 8.2: Execution times of experiments conducted with `CountMinSketch`, leveraging increasing number of workers

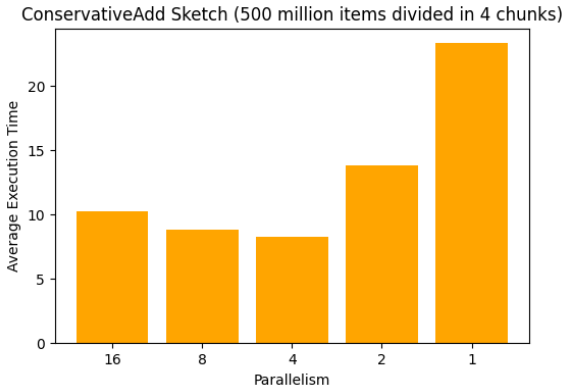

Figure 8.3: Execution times of experiments conducted with `ConservativeAddSketch`, leveraging increasing number of workers

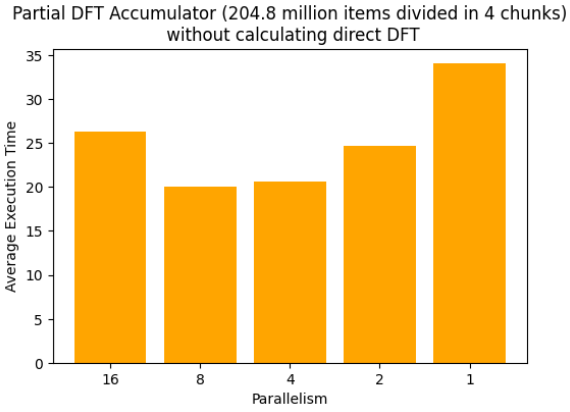

Figure 8.4: Execution times of experiments conducted with `PartialDFTAccumulator`, leveraging increasing number of workers

In Figure 8.4 a direct DFT is computed only for validation against the `PartialDFTAccumulator`'s result. Its execution time is excluded from the measurements, as it does not contribute to parallelism.

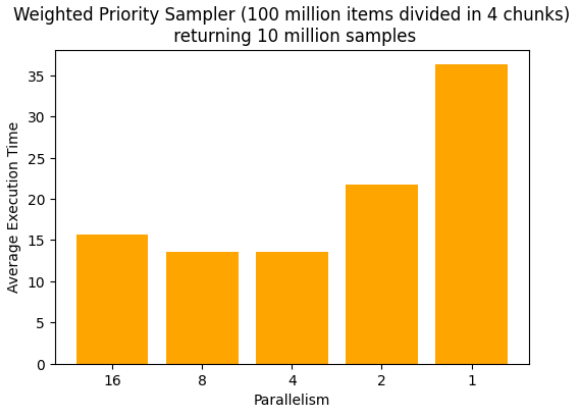

Figure 8.5: Execution times of experiments conducted with `WeightedPrioritySampler`, leveraging increasing number of workers

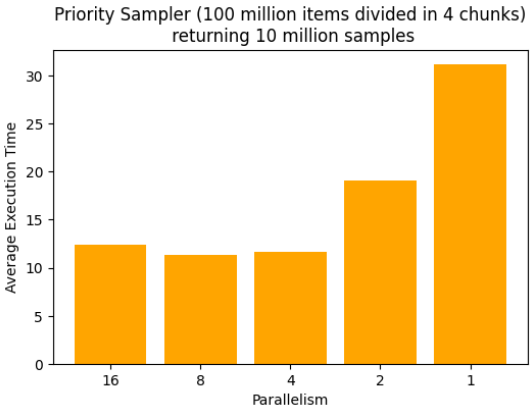

Figure 8.6: Execution times of experiments conducted with `PrioritySampler`, leveraging increasing number of workers

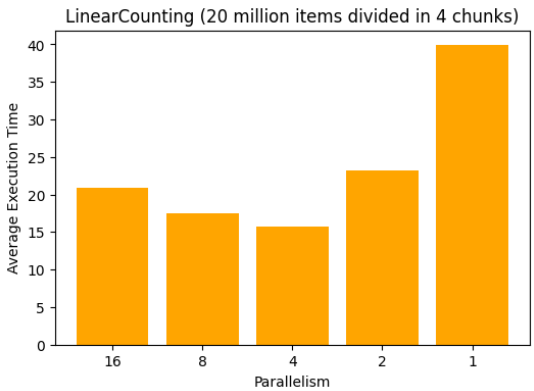

Figure 8.7: Execution times of experiments conducted with `LinearCounting`, leveraging increasing number of workers

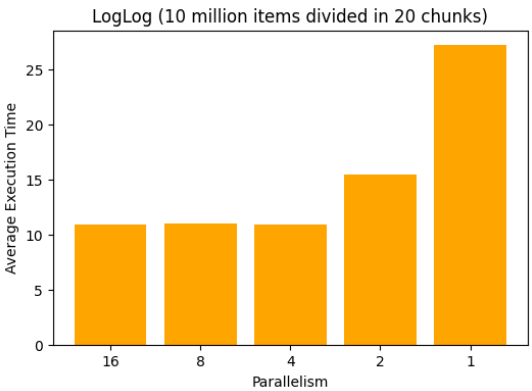

Figure 8.8: Execution times of experiments conducted with `LogLog`, leveraging increasing number of workers

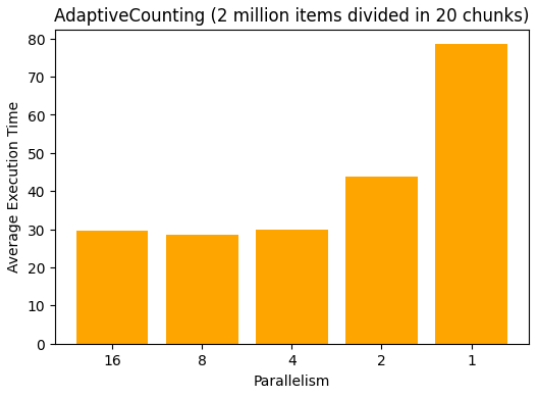

Figure 8.9: Execution times of experiments conducted with `AdaptiveCounting`, leveraging increasing number of workers

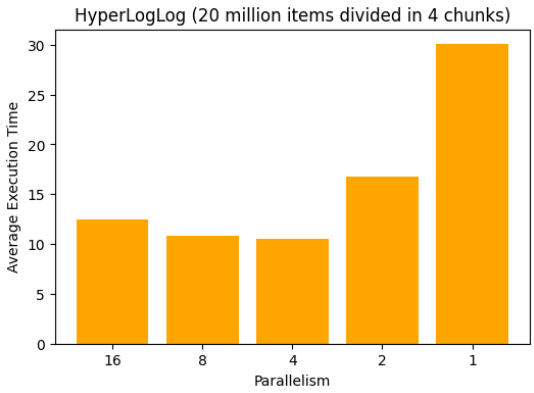

Figure 8.10: Execution times of experiments conducted with `HyperLogLog`, leveraging increasing number of workers

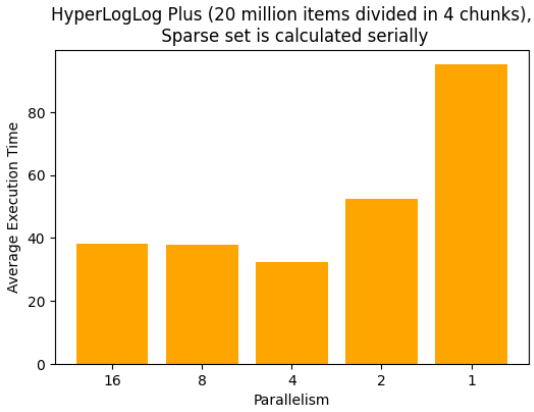

Figure 8.11: Execution times of experiments conducted with `HyperLogLogPlus`, leveraging increasing number of workers

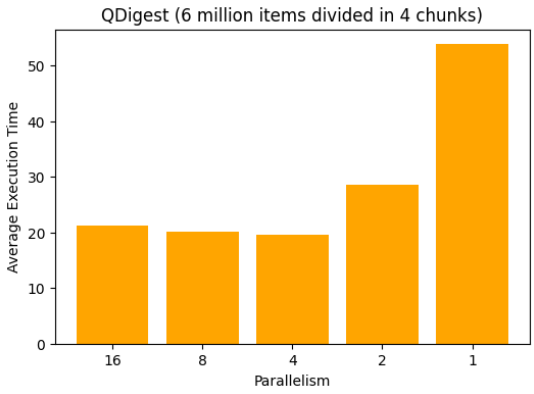

Figure 8.12: Execution times of experiments conducted with `QDigest`, leveraging increasing number of workers

In Figure 8.11, the first 200,000 artificially generated data (representing 1% of the dataset) are inserted serially into the synopsis, while in `Sparse` mode, triggering a transition of `HyperLogLogPlus` to `Normal` mode, which enables parallel processing.

In most figures, it can be observed that using more than four workers (eight or sixteen) actually increases execution time. This is due to two main factors: first, Google Colab supports only up to four active workers, limiting parallelism beyond that point; and second, the cost of merging partial results into one merged synopsis, grows with the number of workers. Since the merging procedure involves only reductions, combining more partial synopses becomes increasingly costly.
However, in `AdaptiveCounting` and in `PartialDFTAccumulator` (Figures 8.4, 8.9) it can be noticed that when using eight workers, execution time continues to slightly reduce.

From Figures 8.1 to 8.12, we observe that the execution time is approximately halved when moving from 1 to 2 workers ($T_2 = T_1/2$), indicating ideal linear and parallel scaling, as our workload is almost perfectly distributed into the 2 workers. However, when moving from 2 to 4 workers, the execution time decreases only to about two-thirds, rather than another halving ($T_4 = \frac{2*T_2}{3} \neq \frac{T_2}{2}$). This is sublinear scaling, and it occurs due to:

1. **Merging Cost of Partial Synopses**: In Section 6.5 we describe how our synopses (e.g. `HyperLogLog`, `PrioritySampler`) are merged after being built in parallel. This merging step:
    - Increases in complexity with more workers.

- Often requires element-wise operations (e.g., `max()` across registers in `HyperLogLog` or bitwise `OR` in `Bloom Filter`) and their cost accumulates with more workers.

2. **More workers cause more scheduling overhead**: `Dask` introduces synchronization **delays** in task graph execution and inter-worker communication as worker count (parallel tasks) grows.

8.2 Improving SuBiTO performance

To enhance SuBiTO's performance, we modified its data preprocessing approach. In more detail, we replaced its original sampling function (`reservoir_sampling`) with our `PrioritySampler`'s `reservoir_priority_sampling`, as explained in section **6.4.6**. This change enables SuBiTO to preprocess data in parallel using `Dask`. The impact of this improvement is demonstrated in the following experiments, where SuBiTO is evaluated on image (CIFAR-10 [53]), video (UCF-50 [46]), and weather time series (Jena Climate 2009-2016 [33]) datasets. In these experiments, 20 BO micro-benchmarks were computed three times with different parallelization degrees in each time (number of workers) and each benchmark had different configurations than the others (sampling percentage and epochs number). In these micro-benchmarks, sampling percentage varied between $0.1 - 0.2$, in order to compare similar examples. Also, data where divided into 4 chunks, for them to be processed optimally in parallel across the available CPU cores.

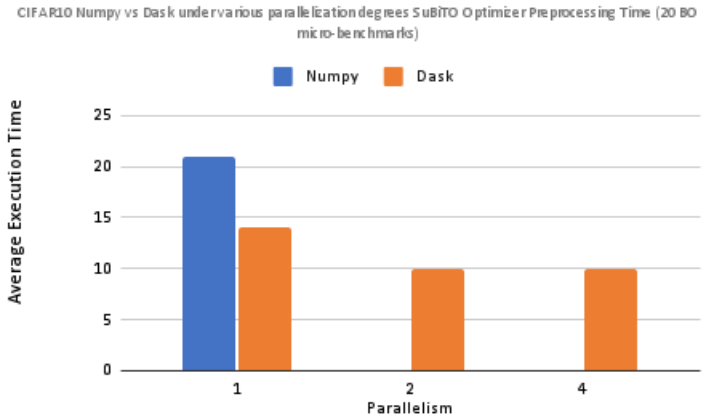

Figure 8.13: `CIFAR10 Numpy` vs `Dask` replicated 4 times, under various parallelization degrees, **SuBiTO Optimizer** Total Preprocessing (Sampling) Time (20 BO micro-benchmarks)

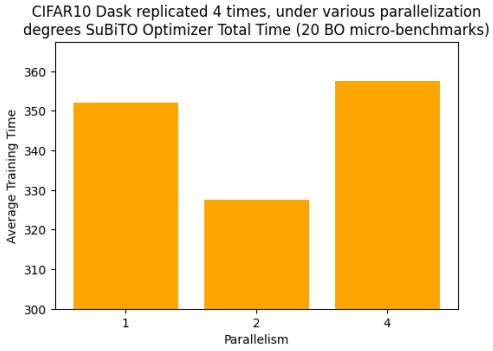

Figure 8.14: `CIFAR10 Dask` replicated 4 times, under various parallelization degrees, **SuBiTO Optimizer** Total Time (20 BO micro-benchmarks)

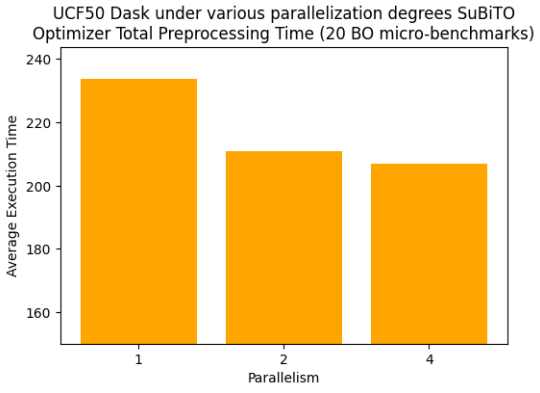

Figure 8.15: `UCF50 Dask` under various parallelization degrees **SuBiTO Optimizer** Total Preprocessing (Sampling) Time (20 BO micro-benchmarks)

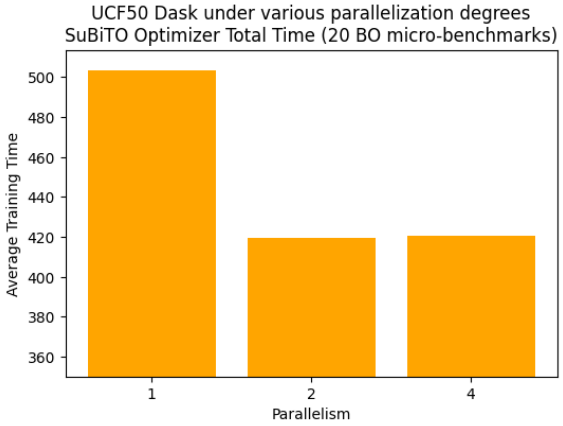

Figure 8.16: `UCF50 Dask` under various parallelization degrees, **SuBiTO Optimizer** Total Time (20 BO micro-benchmarks)

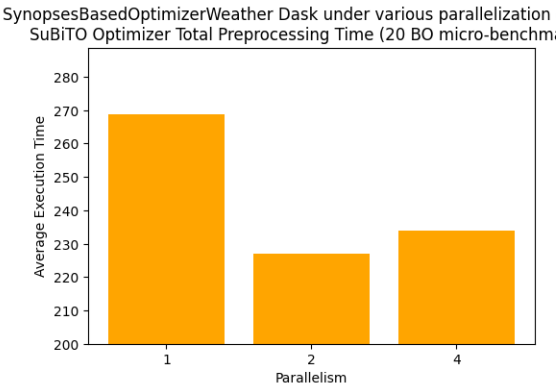

Figure 8.17: `jena_climate_2009_2016 Dask` under various parallelization degrees **SuBiTO Optimizer** Total Preprocessing (Sampling) Time (20 BO micro-benchmarks)

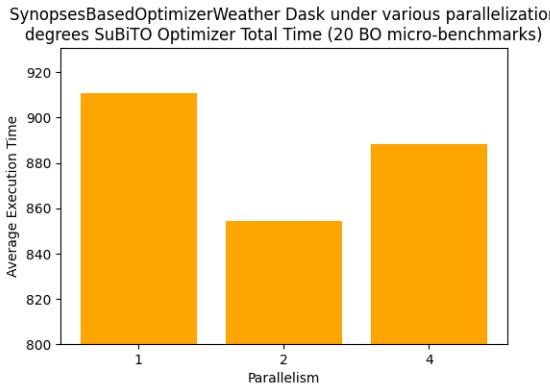

Figure 8.18: `jena_climate_2009_2016 Dask` under various parallelization degrees, **SuBiTO Optimizer** Total Time (20 BO micro-benchmarks)

We save 25 secs in `CIFAR10`, 85 secs in `UCF50` and 57 secs in Jena Climate timeseries forecasting datasets. Setting parallelism to 2 in these datasets is sufficient. No major improvement above parallelism = 2. Observations:

- The training trials of the **Bayesian Optimization** dominate the entire execution time,
- increasing parallelism introduces synchronization delays produced by merging partial synopses of >1 workers to produce the final sample that will be fed in the NN.

This performance relegation is likely produced by the nature of the **SuBiTO** pipeline, where not all stages benefit equally from parallelization. While synopses computation (e.g., sampling) can leverage parallelism, Bayesian optimization, model training and evaluation may not benefit from more workers, as they are inherently sequential or involve iterative dependencies. Also, adding more workers could lead to idle CPU cycles.
Finally, the increased task scheduling and communication overhead caused by the usage of more workers, may outweigh the gains from additional parallelism. In more detail, If the actual computation per task is small (which might be true for SuBiTO microbenchmarks), then the overhead dominates.

In Figures 8.14 - 8.18, y-axis starts from a larger value to better demonstrate the execution time that is saved by using 2 workers.

The data used in these experiments are not really big enough for `Dask` to perform clearly better than `Numpy`. Only in the `CIFAR10` example, shown in Figure 8.13, `Dask` preprocesses data faster than `Numpy` because 200,000 images where used, which is a big dataset.

In general, SDEaaS [22] demonstrates faster performance in terms of raw stream synopsis preprocessing operations. However, the time it gains in the data preprocessing step, is largely dominated by the overhead introduced during the conversion of Flink DataStreams [5] into NumPy-compatible structures that can be consumed by TensorFlow [52].
This transformation step adds significant latency, especially in high-throughput scenarios. Consequently, SDEaaS's advantages in speed at the data preprocessing level are dominated by its difficulty to seamlessly integrate with machine learning pipelines.

Chapter 9

Conclusion & Future Work

In this thesis, we presented a parallel Python-based Synopses Data Engine built on `Apache Dask` [11], specifically implemented to support efficient and scalable data preprocessing for neural learning applications. Our engine provides a unified framework that implements and maintains a wide variety of stream synopsis algorithms (including `Count-Min Sketch, Weighted PrioritySampler, QDigest, Partial DFTAccumulator` and others). It integrates seamlessly with `TensorFlow` [52] pipelines by leveraging `Dask`'s capabilities. This integration addresses a significant limitation in current data engines, which are mostly Java-based and therefore not smoothly compatible with tensor operations.

We demonstrated the utility of our system by conducting experiments in two similar domains: (i) in some generic stream summarization scenarios, where we used distributed and mergeable probabilistic structures, and (ii) to enhance the performance of the SuBiTO framework for scalable learning over streaming data, where our toolkit preprocessed and summarized real-time and memory-efficient parallel streams. Experimental results confirmed that parallel maintenance of synopses significantly improves computational efficiency while maintaining statistical accuracy.

Our future work will focus on the following directions:

- **Expansion of Synopsis Library and Estimation Techniques**: We aim to enrich our engine with additional data summarization techniques to broaden the supported functionality, including more algorithms for approximate top-k querying, set membership, and advanced quantile summarization (more algorithms like `PrioritySampler, BloomFilter` and `QDigest` respectively).
- **Reevaluate our Experiments using Enhanced Computational Resources**: The current experimental setup, constrained by the limitations of **Google Colab Pro** (50GB RAM and four CPU cores), restricted our ability in some cases to produce really big data volumes to a level where `Dask`'s parallelism could offer clear performance advantages over `NumPy`. Therefore, future experiments should be conducted in environments with higher memory capacity and more cores at their disposal, to more accurately prove `Dask`'s performance benefits in large-scale processing scenarios.

Bibliography

[1] AddThis. *Stream-lib: Stream summarizer and cardinality estimator*. 2019. URL: https://github.com/addthis/stream-lib/.

[2] Nisheeth Shrivastava & Chiranjeeb Buragohain & Divyakant Agrawal. "Medians and Beyond: New Aggregation Techniques for Sensor Networks". In: *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems* (2004), pp. 239–249. URL: https://dl.acm.org/doi/10.1145/1031495.1031524.

[3] *Apache DataSketches*. 2020. URL: https://github.com/apache/datasketches-website/.

[4] *Apache DataSketches functions - Distinct Counting with HyperLogLog Sketches*. URL: https://docs.firebolt.io/sql_reference/functions-reference/datasketches/.

[5] *Apache Flink®*. URL: https://flink.apache.org/.

[6] *Apache Geode*. URL: https://geode.incubator.apache.org/.

[7] *Apache Kafka*. URL: https://kafka.apache.org/.

[8] *Apache Spark - Unified engine for large-scale data analytics*. URL: https://spark.apache.org/.

[9] *Apache Spark Streaming*. URL: https://spark.apache.org/streaming/.

[10] Nikos Giatrakos & Elias Alevizos & Antonios Deligiannakis & Ralf Klinkenberg & Alexander Artikis. "Proactive Streaming Analytics at Scale: A Journey from the State-of-the-art to a Production Platform". In: *CIKM '23: Proceedings of the 32nd ACM International Conference on Information and Knowledge Management* (2023), pp. 5204–5207. URL: https://dl.acm.org/doi/10.1145/3583780.3615293.

[11] Gourav Singh Bais. *Parallel computing with Dask: a step-by-step tutorial*. 2023. URL: https://domino.ai/blog/dask-step-by-step-tutorial.

[12] *Better performance with tf.function*. URL: https://www.tensorflow.org/guide/function.

[13] *C++: Lookup3 hash/Jenkins hash functions*. URL: https://asecuritysite.com/hash/smh_lookup3?val1=abcdefghijklmnopqrstuvwxyz&seed1=4.

[14] *Convolutional Neural Network (CNN)*. URL: https://www.tensorflow.org/tutorials/images/cnn.

[15] *Dask Array Documentation*. URL: https://docs.dask.org/en/stable/array.html.

[16] *Dask Array Map_Blocks Documentation*. URL: https://docs.dask.org/en/stable/generated/dask.array.map_blocks.html.

[17] *Dask Dashboard Diagnostics*. URL: https://docs.dask.org/en/latest/dashboard.html.

[18] *Dask DataFrame Documentation*. URL: https://docs.dask.org/en/stable/dataframe.html.

[19] *Dask Delayed Documentation*. URL: https://docs.dask.org/en/stable/delayed.html.

[20] *DataSketches - An introduction*. 2020. URL: https://archive.fosdem.org/2020/schedule/event/apache_datasketches/attachments/slides/3547/export/events/attachments/apache_datasketches/slides/3547/DataSketches_An_introduction.pdf.

[21] *DataSketches extension*. URL: https://druid.apache.org/docs/latest/development/extensions-core/datasketches-extension/.

[22] Antonios Kontaxakis & Nikos Giatrakos & Antonios Deligiannakis. "A Synopses Data Engine for Interactive Extreme-Scale Analytics". In: *CIKM '20: Proceedings of the 29th ACM International Conference on Information & Knowledge Management* (2020), pp. 2085–2088. URL: https://doi.org/10.1145/3340531.3412154.

[23] Antonios Kontaxakis & Nikos Giatrakos & Dimitris Sacharidis & Antonios Deligiannakis. "And synopses for all: A synopses data engine for extreme scale analytics-as-a-service". In: *Information Systems* 116 (2023). URL: https://www.sciencedirect.com/science/article/pii/S0306437923000571.

[24] *Discrete Fourier Transform*. URL: https://en.wikipedia.org/wiki/Discrete_Fourier_transform#.

[25] *Discrete Fourier Transform - Linearity*. URL: https://en.wikipedia.org/wiki/Discrete_Fourier_transform#Properties.

[26] *Distributed - spread your data and computation across a cluster*. URL: https://tutorial.dask.org/04_distributed.html.

[27] Marianne Durand & Philippe Flajolet. "LogLog counting of large cardinalities". In: *ESA03* 2832 (2003), pp. 605–617. URL: https://algo.inria.fr/flajolet/Publications/DuFl03.pdf.

[28] E. Brochu & V. M. Cora & N. de Freitas. *A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning*. 2010. URL: https://arxiv.org/abs/1012.2599.

[29] *Frequent Items*. URL: https://apache.github.io/datasketches-python/5.1.0/frequency/frequent_items.html.

[30] Stefan Heule & Marc Nunkesser & Alex Hall. "HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm". In: *EDBT '13: Proceedings of the 16th International Conference on Extending Database Technology* (2013), pp. 683–692. URL: https://dl.acm.org/doi/10.1145/2452376.2452456.

[31] Min Cai & Jianping Pan & Yu K. Kwok & Kai Hwang. "Fast and accurate traffic matrix measurement using adaptive cardinality counting". In: *MineNet '05: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data* (2005), pp. 205–206. URL: https://dl.acm.org/doi/10.1145/1080173.1080185.

[32] *Introduction to graphs and tf.function*. URL: https://www.tensorflow.org/guide/intro_to_graphs.

[33] *Jena Climate Dataset 2009-2016 - Time Series Forecasting Dataset*. URL: https://www.kaggle.com/datasets/mnassrib/jena-climate.

[34] *Keras: The high-level API for TensorFlow*. URL: https://www.tensorflow.org/guide/keras.

[35] Antonios Kontaxakis. "Design and implementation of a distributed synopsis data engine on Apache Flink". In: (2020). URL: https://dias.library.tuc.gr/view/85602.

[36] R.P. Lemaitre & M. Kiefer & J.V. Hein & J. Quiané-Ruiz & V. Markl. "In the land of data streams where synopses are missing, one framework to bring them all". In: *Proc. VLDB Endow.* 14(10) (2021), pp. 1818–1831. URL: https://dl.acm.org/doi/abs/10.14778/3467861.3467871.

[37] Philippe Flajolet & Eric Fusy & Olivier Gandouet & Frédéric Meunier. "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm." In: *Discrete Mathematics and Theoretical Computer Science. DMTCS* AH (2007), pp. 127–146. URL: https://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf.

[38] Antonios Monogyios. "Parallel Techniques for Neural Network Verification". In: (2025). URL: https://dias.library.tuc.gr/view/102611?locale=en.

[39] Barzan Mozafari. "SnappyData". In: *Encyclopedia of Big Data Technologies* (2019), pp. 1522–1531. URL: https://link.springer.com/rwe/10.1007/978-3-319-77525-8_258.

[40] *MurmurHash*. URL: https://en.wikipedia.org/wiki/MurmurHash#.

[41] Graham Cormode & S. Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications." In: *Journal of Algorithms* 55 (2005), pp. 58–75. URL: https://www.sciencedirect.com/science/article/pii/S0196677403001913.

[42] *NSFW-detect Dataset*. 2024. URL: https://huggingface.co/datasets/deepghs/nsfw_detect.

[43] *Proteus Project*. URL: https://github.com/proteus-h2020.

[44] *PyTorch*. URL: https://pytorch.org/.

[45] *Rapidminer studio, streaming extension*. URL: https://docs.rapidminer.com/10.2/studio/connect/streaming/index.html.

[46] Kishore K. Reddy & Mubarak Shah. "Recognizing 50 human action categories of web videos". In: *Machine Vision and Applications* 24 (2013), pp. 971–981. URL: https://dl.acm.org/doi/10.1007/s00138-012-0450-4.

[47] *Streamlit*. 2024. URL: https://streamlit.io/.

[48] Errikos Streviniotis, George Klioumis, and Nikos Giatrakos. "SuBiTO: Synopsis-based Training Optimization for Continuous Real-Time Neural Learning over Big Streaming Data (Demo Paper)". In: *Proceedings of the 39th Annual AAAI Conference on Artificial Intelligence (AAAI'25)*. Philadelphia, Pennsylvania, USA, Mar. 2025. URL: http://users.softnet.tuc.gr/~ngiatrakos/papers/aaai25.pdf.

[49] Do Le Quoc1 & Ruichuan Chen2 & Pramod Bhatotia3 & Christof Fetzer1 & Volker Hilt2 & Thorsten Strufe1. "StreamApprox: Approximate Computing for Stream Analytics". In: *Middleware '17: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (2017), pp. 185–197. URL: https://dl.acm.org/doi/10.1145/3135974.3135989.

[50] *SuBiTO*. 2024. URL: https://subito-ai-for-bigdata.github.io/.

[51] Kyu Y. Whang & Brad T. Vander Zanden & Howard M. Taylor. "A linear-time probabilistic counting algorithm for database applications." In: *ACM Transactions on Database Systems (TODS)* 15(2) (1990), pp. 208–229. URL: https://dl.acm.org/doi/10.1145/78922.78925.

[52] *Tensorflow basics*. URL: https://www.tensorflow.org/guide/basics.

[53] *The CIFAR-10 dataset*. URL: https://www.cs.toronto.edu/%7Ekriz/cifar.html.

[54] *The Functional API*. URL: https://www.tensorflow.org/guide/keras/functional_api.

[55] *The Sequential model*. URL: https://www.tensorflow.org/guide/keras/sequential_model.

[56] Nick Duffield & Carsten Lund & Mikkel Thorup. "Priority sampling for estimation of arbitrary subset sums". In: *Journal of the ACM (JACM)* 54 (6 2007), pp. 1–39. URL: https://doi.org/10.1145/1314690.1314696.

[57] *Using Ngrok with Kubernetes: A Practical Guide*. URL: https://blog.techiescamp.com/using-ngrok-with-kubernetes/.

[58] *Variance Optimal Sampling (VarOpt)*. URL: https://apache.github.io/datasketches-python/5.1.0/sampling/varopt.html.

[59] *Working with RNNs*. URL: https://www.tensorflow.org/guide/keras/working_with_rnns.

[60] Nick Duffield & Yunhong Xu & Liangzhen Xia & Nesreen K. Ahmed & Minlan Yu. "Stream Aggregation Through Order Sampling". In: *CIKM '17: Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (2017), pp. 909–918. URL: https://doi.org/10.1145/3132847.3133042.