Compact Storage of Data Streams in Mobile Devices

Rémy Raes, Olivier Ruas, Adrien Luxey-Bitri, Romain Rouvoy

▶ **To cite this version:**

Rémy Raes, Olivier Ruas, Adrien Luxey-Bitri, Romain Rouvoy. Compact Storage of Data Streams in Mobile Devices. DAIS'24 - 24th International Conference on Distributed Applications and Interoperable Systems, Jun 2024, Groningen, Netherlands. hal-04535716v3

HAL Id: hal-04535716
https://hal.science/hal-04535716v3

Submitted on 22 Apr 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Compact Storage of Data Streams in Mobile Devices

Rémy Raes[1], Olivier Ruas[2], Adrien Luxey-Bitri[1], and Romain Rouvoy[1]

[1] Inria, Univ. Lille, CNRS, UMR 9189 CRIStAL, France
[2] Pathway, France

Abstract. Data streams produced by mobile devices, such as smartphones, offer highly valuable sources of information to build ubiquitous services. However, the diversity of embedded sensors and the resulting data deluge makes it impractical to provision such services directly on mobiles, due to their constrained storage capacity, communication bandwidth and processing power. Unfortunately, the improving hardware capabilities of devices are unlikely to resolve these structural issues. We, therefore, believe that mobile data management systems should, instead, handle data streams efficiently and compactly, to provision services directly at the edge, while accounting for the limits of existing assets and network infrastructures. This paper introduces the FLI framework, which leverages a piece-wise linear approximation technique to capture compact representations of data streams in mobile devices. Our experiments, performed on Android and iOS devices, show that FLI outperforms the state of the art both in memory footprint and I/O throughput. Our Flutter implementation of FLI can store stream datasets in mobile devices, which is a prerequisite to processing big data from ubiquitous devices *in situ*.

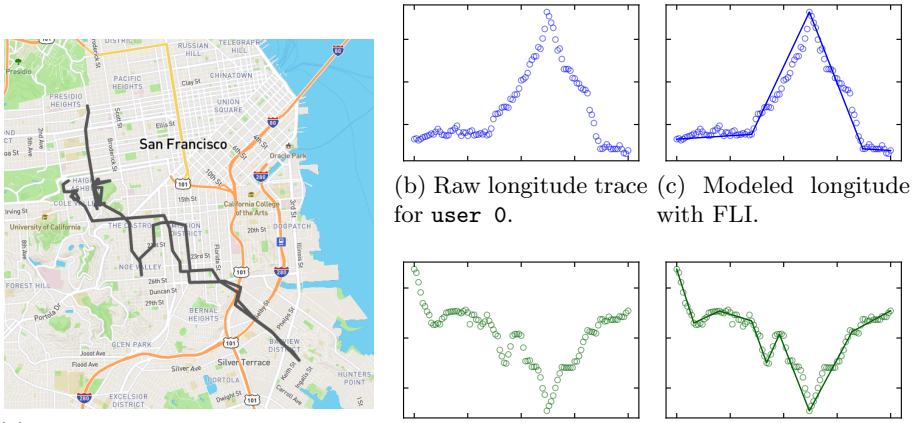
1 Introduction

With the advent of smartphones and more generally the *Internet of Things* (IoT), ubiquitous devices are mainstream in our societies and widely deployed at the edge of networks. Such constrained devices are not only consuming data and services, such as content streaming, restaurant recommendations or more generally *Location-Based Service*s (LBSs), but are also key producers of data streams by leveraging a wide variety of embedded sensors that capture the surrounding environment of end-users, including their daily routines. The data deluge generated by a connected user is potentially tremendous: according to preliminary experiments, a smartphone can generate approximately 2 pairs of *Global Positioning System* (GPS) samples and 476 triplets of accelerometer samples per second, resulting in more than 172,800 location and 41,126,400 acceleration samples daily.

In this context, the storage and processing of such data streams in mobile devices are challenges that cannot only be addressed by assuming that the hardware capabilities will keep increasing. In particular, sustainability issues call for increasing the lifespan of legacy devices, thus postponing their replacement. This

implies that software-defined solutions are required to leverage the shortenings of hardware resources.

This paper, therefore, demonstrates that modeling data streams successfully address these device-level storage & processing challenges. More specifically, to address the storage challenge, we introduce *Fast Linear Interpolation* (FLI): a novel algorithm leveraging a *Piece-wise Linear Approximation* (PLA) technique to model and store data streams under memory constraints. Figure 1 illustrates FLI's behavior: to capture the trajectory displayed on Fig. 1a, FLI does not store raw data samples (Fig. 1b & 1d) but, instead, models their evolution as linear interpolations (Fig. 1c & 1e) —thus offering a much bigger storage capacity at the cost of a controlled approximation error.



(a) Cabspotting mobility trace of `user 0`.

(b) Raw longitude trace for `user 0`.

(c) Modeled longitude with FLI.

(d) Raw latitude trace for `user 0`.

(e) Modeled latitude with FLI.

Fig. 1. FLI compacts any location stream as a sequence of segments.

In the following, we first discuss the related works (Sec. 2), before diving into the details of FLI (Sec. 3). We then present our experimental setup (Sec. 4) and the results we obtained (Sec. 5). Finally, we discuss the limitations of our approach (Sec. 6), before concluding (Sec. 7).

2 Related Work

Overcoming the memory constraints of mobile devices to store data streams usually implies the integration of efficient temporal databases. To take the example of Android: few databases are available, such as SQLite and its derivative Drift [3], the cloud-supported Firebase [21], the NoSQL hive, and ObjectBox [5]. The situation is similar on iOS.

Relational databases (*e.g.*, SQL) are typically designed for *OnLine Transactional Processing* (OLTP) and *OnLine Analytical Processing* (OLAP) workloads, which widely differ from time-series workloads. In the latter, reads are mostly contiguous (as opposed to the random-read tendency of OLTP); writes are most

often inserts (not updates) and typically target the most recent time ranges. OLAP is designed to store big data workloads to compute analytical statistics, while not putting the emphasis on read or write performances. Finally, in temporal workloads, it is unlikely to process writes & reads in the same single transaction [22].

Time series databases (TSDB). Despite these deep differences, several relational databases offer support for temporal data with industry-ready performance—*e.g.*, TimescaleDB [9] is a middleware that exposes temporal functionalities atop a relational PostgreSQL foundation. InfluxDB [10] is one of the most widely used temporal databases. Unfortunately, when facing memory constraints, its retention policy prevents the storage from scaling in time: the oldest samples are dumped to make room for the new ones. Furthermore, on mobile, memory shortages often cause the operating system to kill the TSDB process to free the memory, which is opposed to the very concept of in-memory databases.

Moving objects databases (MOD). Location data storage is an issue that has also been studied in the MOD community, where a central authority merges trajectory data from several sensors in real-time. To optimize storage and communication costs, it does not store the raw location data, but rather trajectory approximations. *Linear Dead Reckoning* (LDR) [24] limits data exchange between sensors and server by sending new location samples only when a predefined *accuracy bound* ϵ (in meters) is exceeded. A mobility prediction vector is additionally shared every time a location sample is sent. Even so, this class of solutions requires temporarily storing modeled locations to ensure they fit the ϵ bound and exclusively focuses on modeling location data streams, while we aim at storing any type of real-valued stream.

Modeling data streams. While being discrete, the streams sampled by sensors represent inherently continuous signals. Data modeling does not only allow important memory consumption gains, but also flattens sensors' noise, and enables extrapolation between measurements. In particular, *Piece-wise Linear Approximation* (PLA) is used to model the data as successive affine functions. An intuitive way to do linear approximation is to apply a bottom-up segmentation: each pair of consecutive points is connected by interpolations; the less significant contiguous interpolations are merged, as long as the obtained interpolations introduce no error above a given threshold. The bottom-up approach has low complexity, but usually requires an offline approach to consider all the points at once. The *Sliding Window And Bottom-up* (SWAB) algorithm [11], however, is an online approach that uses a sliding window to buffer the latest samples on which a bottom-up approach is applied. emSWAB [2] improves the sliding window by adding several samples at the same time instead of one. Instead of interpolation, linear regression can also be used to model the samples reported by IoT sensors [8]. For example, Greycat [13] adopts polynomial regressions with higher degrees to further compress the data. Unfortunately, none of those works have been implemented on mobile devices to date.

Sprintz [4] proposes a mobile lossless compression scheme for multi-modal integer data streams, along with a comparison of other compression algorithms.

They target streaming of the compressed data to a centralized location from IoT devices with minimal resources. This work is orthogonal to ours, as FLI intends to model floating-point unimodal streams on one's devices for further local computation, instead of streaming it to a third-party server.

Closer to our work, FSW [12] and the ShrinkingCone algorithm [6] attempt to maximize the length of a segment while satisfying a given error threshold, using the same property used in FLI. FSW is not a streaming algorithm as it considers the dataset as a whole, and does not support insertion. The ShrinkingCone algorithm is a streaming greedy algorithm designed to approximate an index, mapping keys to positions: it only considers monotonic increasing functions and can produce disjoints segments. FLI models non-monotonic functions in a streaming fashion, while providing joint segments.

Limitations. To the best of our knowledge, state-of-the-art storage solutions for unbounded data streams either require storing raw data samples or triggering *a posteriori* data computations, which makes them unsuitable for mobile devices.

3 Storing Data Streams in the Small

3.1 Leveraging Piecewise Linear Approximations

To overcome the memory constraint of mobile devices, we claim that efficient temporal storage solutions must be ported onto ubiquitous environments. In particular, we advocate the use of data modeling, such as *Piece-wise Linear Approximation* (PLA) [11,8] or Greycat [13], to increase the storage capacity of mobile devices. Therefore, we introduce FLI, a time series modeling algorithm based on an iterative and continuous PLA to store approximate models of data streams on memory-constrained devices, instead of storing all the raw data samples as state-of-the-art temporal databases do. FLI models one-dimensional points (or samples) p as piece-wise linear segments (or interpolations) $\boldsymbol{s}$. It enforces the following invariant: *all samples modeled by an interpolation maintain an error below the configuration parameter* ϵ. Its data structure $\mathcal{D}$ is composed of *i)* a list of selected historical points $\mathcal{P}$, *ii)* the latest segment's gradient α_M, and *iii)* the two bounding gradients $\alpha_{\min}$ and $\alpha_{\max}$ used for insertion:

$$\mathcal{D} = (\mathcal{P}, \alpha_M, \alpha_{\min}, \alpha_{\max}), \text{ s.t.}$$
$$\mathcal{P} = [\ldots, p_i, p_{i+1}, \ldots, p_M] \subset \mathbb{R}^2 \ \&\ (\alpha_M, \alpha_{\min}, \alpha_{\max}) \in \mathbb{R}^3$$

Historical segments are captured as tuples of consecutive samples: $\boldsymbol{s}_i = [p_i, p_{i+1}]$. The latest interpolation $\boldsymbol{s}_M$ takes the last inserted sample p_M as its origin and the gradient α_M as its slope, as depicted in Fig. 2. We first present how observed points are inserted, before explaining how reading a value is performed.

3.2 Inserting Data Samples

Data samples are inserted sequentially: the current interpolation is adjusted to fit new samples until it cannot satisfy the invariant. Upon insertion of a new sample

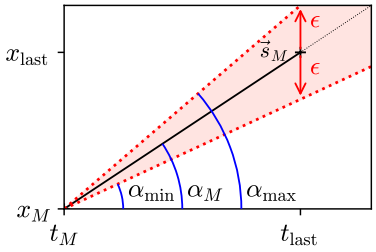
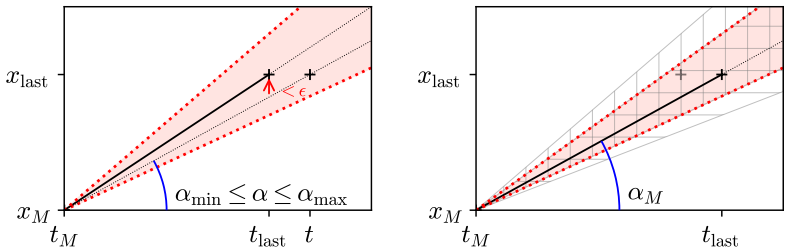


Fig. 2. A new FLI interpolation $\boldsymbol{s}_M$ begins with 2 samples: the point $p_M = (t_M, x_M)$ as origin, and the latest sample $p_{\text{last}} = (t_{\text{last}}, x_{\text{last}})$ as slope α_M. 2 bounding gradients $\alpha_{\min}$ and $\alpha_{\max}$ are derived from ϵ, and used to assert if future inserts fit $\boldsymbol{s}_M$.

p, the slope α of the segment $[p_M, p]$ is compared to the interval $[\alpha_{\min}, \alpha_{\max}]$. If it falls within (cf. Fig. 3), p is added to the current interpolation: $\alpha_{\min}$ and $\alpha_{\max}$ are updated to encompass p, α_M is updated to α (for reading), and the previous sample is dropped. If p is outside the interval (cf. Fig. 4), a new interpolation begins from the 2 last points.



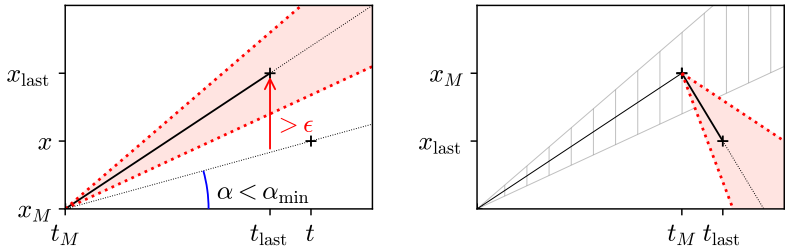
(a) Gradient α of sample p remains within $[\alpha_{\min}, \alpha_{\max}]$.

(b) α_M, $\alpha_{\min}$ & $\alpha_{\max}$ are updated to include p in the current model.

Fig. 3. When a new sample fits within $[\alpha_{\min}, \alpha_{\max}]$, it is added to the current model by updating $\alpha_{\min}$ and $\alpha_{\max}$.α_M is also updated for read queries (see Alg. 1).

The value of ϵ has an important impact on the performances of FLI. If ϵ is too small, none of the inserted samples fits the current model at that time, thus initiating a new model each time. In that case, there will be one model per sample, imposing an important memory overhead. The resulting model overfits the data. On the other hand, if ϵ is too large, then all the inserted samples fit, and a single model is kept. While it is the best case memory-wise, the resulting model simply connects the first and last point and underfits the data.

As long as the newly inserted data samples fit the existing model, the memory footprint of FLI remains unchanged. This potentially unlimited storage capacity makes FLI a key asset for mobile devices, for example drastically increasing the storage capacity for user mobility traces. While FLI is designed for the modeling



(a) Error of $p = (t, x)$ exceeds ϵ.

(b) A new model is created from the last 2 points.

Fig. 4. When an error $> \epsilon$ is reported, a new model is created using p_{last} as p_M.

of univariate data streams, it straight-forwardly generalizes to multivariate data streams by combining several instances of FLI. We, therefore, claim that the use of FLI alleviates the memory constraint of mobile devices, hence opening new opportunities to process unbounded data streams locally.

3.3 Reading Data Streams

In FLI, reading a value at time t is achieved by estimating its image using the appropriate interpolation, as is shown in Algorithm 1. If t is ulterior or equal to t_M, the current interpolation $\boldsymbol{s}_M$ is used (line 3), defined by $p_M = (t_M, x_M)$ and α_M. When t is anterior to t_M, FLI reconstructs the interpolation $\boldsymbol{s}_i$ in charge of approximating t by picking 2 consecutive points from $\mathcal{P}$ (lines 5–6). In practice, the segment is found with a dichotomy search, as $\mathcal{P}$ stores points in insertion order. Using that model, the interpolation of t is computed on line 7.

Algorithm 1 Approximate read implemented by FLI

Require: $\mathcal{D} = (\mathcal{P}, \alpha_M, \alpha_{\min}, \alpha_{\max})$
1: **function** READ($t \in \mathbb{R}$)
2: **if** $t_M \leq t$ **then**
3: **return** $\alpha_M \times (t - t_M) + x_M$
4: **end if**
5: **select** i **s.t.** $((t_i, x_i), (t_{i+1}, x_{i+1})) \in \mathcal{P} \wedge t_i \leq t < t_{i+1}$
6: $\alpha_i \leftarrow (x_{i+1} - x_i) / (t_{i+1} - t_i)$
7: **return** $\alpha_i \times (t - t_i) + x_i$
8: **end function**

4 Experimental Setup

4.1 Key Performance Metrics

We consider state-of-the-art system metrics to evaluate the performance of FLI:

Memory footprint. The key objective of FLI is to reduce the memory footprint required to store an unbounded stream of samples. We explore two metrics: *(i)* the number of 64-bit variables required by the model and *(ii)* the size of the model in the device memory. To do so, we compare the size of the persistent file with the size of the vanilla SQLite database file. We consider the number of 64-bit variables as a device-agnostic estimation of the model footprint.

I/O throughput. Another key system metric is the I/O throughput of the temporal databases. In particular, we measure how many write and read operations can be performed per second (IOPS).

4.2 Input Datasets & Parameter Tuning

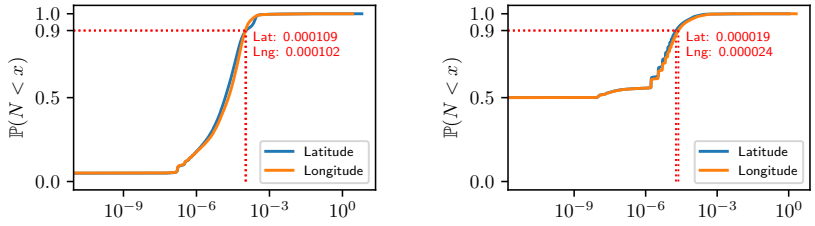
FLI is data-agnostic: any data stream can be modeled using it (cf. Section 5.4). However, the ϵ parameter depends on the underlying data distribution. In the following, we propose a protocol to tune an ϵ value according to the modeled data stream. We use mobility traces as a representative example of data streams that can be stored and processed by modern mobile devices. In particular, we believe that mobility traces are a good candidate for FLI as the storage of sampled user locations may require a lot of storage space. A mobility trace is defined as an ordered sequence T of pairs (t, g) where t is a timestamp and g is a geolocation sample, a latitude-longitude pair for example. The trace is ordered in chronological order and we assume that reported timestamps are unique.

Location datasets. Cabspotting [15] is a mobility dataset of 536 taxis in the San Francisco Bay Area. The data was collected during a month and is composed of 11 million records, for a total of 388 MB. PrivaMov [14] is a multi-sensors mobility dataset gathered during 15 months by 100 users around the city of Lyon, France. We use the full GPS dataset, which includes 156 million records, totaling 7.2 GB. Compared to Cabspotting, PrivaMov is a highly-dense mobility dataset.

Parameter tuning. The choice of an ϵ value is of major importance and plays a central role in FLI's performances: a poorly-chosen value has a strong impact on FLI's underlying segments, either degrading modeled data quality or filling storage space up excessively. To find a compromise between the two, since the ϵ value is highly correlated to the modeled data, one has to *know* the data; more specifically, we advise studying data variation between consecutive values.

For example, in the context of location data, Fig. 5 characterizes—as a *Cumulative Distribution Function* (CDF)—the evolution of longitude and latitude samples for all the traces stored in the Cabspotting and PrivaMov datasets. In particular, we plot the CDF of the drift d observed between 2 consecutive values (x_1, y_1) and (x_2, y_2), which we compute as $d = |y_2 - y_1|/|x_2 - x_1|$. One can observe that Cabspotting and PrivaMov datasets report on a drift lower than 1×10^{-4} and 2×10^{-5} for 90% of the values, respectively. Furthermore, due to the high density of locations captured by PrivaMov, half of the drifts are equal to `0`, meaning several consecutive longitudes or latitudes are unchanged.

This preliminary analysis of both datasets highlights that mobility traces are highly relevant data streams for FLI, and demonstrates $\epsilon = 10^{-3}$ is a conservative



(a) CDF of CABSPOTTING locations' drifts.

(b) CDF of PRIVAMOV locations' drifts.

Fig. 5. *Cumulative Distribution Function* (CDF) of latitude and longitude drifts of successive location samples in CABSPOTTING and PRIVAMOV datasets. One can observe that, from one location sample to the next, latitude or longitude deviations are small.

choice to model location data. To automate the tuning of ϵ, FLI comes with a script that takes a sample input to report on candidate ϵ values to capture 90%, 95% and 99% of the sampled data. The following sections will focus on the evaluation of FLI on those datasets to study the benefits of adopting FLI to capture real-world metrics in mobile devices.

4.3 Storage Competitors

SQLITE is the state-of-the-art solution to persist and query large volumes of data on Android devices. SQLITE provides a lightweight relational database management system. SQLITE is not a temporal database, but is a convenient and standard way to store samples persistently on a mobile device. Insertions are atomic, so one may batch them to avoid one memory access per insertion.

Sliding-Window And Bottom-up (SWAB) [11] is a linear interpolation model. As FLI, the samples are represented by a list of linear models. In particular, reading a sample is achieved by iteratively going through the list of models until the corresponding one is found and then used to estimate the requested value. The bottom-up approach of SWAB starts by connecting every pair of consecutive samples and then iterates by merging the less significant pair of contiguous interpolations. This process is repeated until no more pairs can be merged without introducing an error higher than ϵ. Contrarily to FLI, this bottom-up approach is an offline one, requiring all the samples to be known. SWAB extends the bottom-up approach by buffering samples in a sliding window. New samples are inserted in the sliding window and then modeled using a bottom-up approach: whenever the window is full, the oldest model is kept and the captured samples are removed from the buffer.

One could expect that the bottom-up approach delivers more accurate models than the greedy FLI, even resulting in a slight reduction in the number of models and faster readings. On the other hand, sample insertion is more expensive than FLI due to the execution of the bottom-up approach when storing samples. Like

FLI, SWAB ensures that reading stored samples is at most ϵ away from the exact values.

GREYCAT [13] aims at compressing even further the data by not limiting itself to linear models. GREYCAT also models the samples as a list of models, but these models are polynomials. The samples are read the same way.

When inserting a sample, it first checks if it fits the model. If so, then nothing needs to be done. Otherwise, unlike FLI and SWAB which directly initiate a new model, GREYCAT tries to increase the degree of the polynomial to make it fit the new sample. To do so, GREYCAT first regenerates $d + 1$ samples in the interval covered by the current model, where d is the degree of the current model. Then, a polynomial regression of degree $d + 1$ is computed on those points along the new one. If the resulting regression reports an error lower than $\frac{\epsilon}{2^{d+1}}$, then the model is kept, otherwise, the process is repeated by incrementing the degree until either a fitting model is found or a maximum degree is reached. If the maximum degree is reached, the former model is stored and a new model is initiated. The resulting model is quite compact, and thus faster to read, but at the expense of an important insertion cost.

Unlike FLI and SWAB, there can be errors higher than ϵ for the inserted samples, as the errors are not computed on raw samples but on generated ones, which may not coincide. Furthermore, the use of higher-degree polynomials makes the implementation subject to overflow: to alleviate this effect, the inserted values are normalized.

4.4 Experimental Settings

For experiments with univariate data streams—*i.e.* memory and throughput benchmarks—we set $\epsilon = 10^{-2}$. The random samples used in those experiments follow a uniform distribution in $[-1{,}000; 1{,}000]$: it is very unlikely to have two successive samples with a difference lower than ϵ, hence reflecting the worst case conditions for FLI. For experiments on location data, and unless said otherwise, we set $\epsilon = 10^{-3}$ for FLI, SWAB and GREYCAT. For GREYCAT, the maximum degree for the polynomials is set to 14. The experiments evaluating the throughput were repeated 4 times each and the average is taken as the standard deviation was low. All the other experiments are deterministic and performed once.

4.5 Implementation Details

We implemented FLI using the Flutter *Software Development Kit* (SDK) [7]. Flutter is Google's UI toolkit, based on the Dart programming language, that can be used to develop natively compiled apps for Android, iOS, web and desktop platforms (as long as the project's dependencies implement cross-compilation to all considered platforms). Our implementation includes FLI and its storage competitors.This implementation is publicly available [18].

For our experiments, we also implemented several mobile applications based on this library. To demonstrate its capability of operating across multiple environments (models, operating systems, processors, memory capacities, storage

capacities), all our benchmark applications were successfully installed and executed in the devices listed in Table 1. Unless mentioned otherwise, the host device for the experiments is the Fairphone 3.

Table 1. Mobile devices used in the experiments.

Model	OS	CPU	Cores	RAM	Storage
Lenovo Moto Z	Android 8	Snapdragon 820	4	4 GB	32 GB
Fairphone 3	Android 11	Snapdragon 632	8	4 GB	64 GB
Pixel 7 Pro	Android 13	Google Tensor G2	8	12 GB	128 GB
iPhone 12	iOS 15.1.1	A14 Bionic	6	4 GB	64 GB
iPhone 14 Plus	iOS 16.0.1	A15 Bionic	6	6 GB	128 GB

5 Experimental Results

In this section, we evaluate our implementation of FLI on Android and iOS to show how it enables efficient data stream storage on mobile devices.We first perform several benchmarks (memory, throughput & stability), before evaluating the performance of FLI beyond location streams. Finally, we perform a *Point Of Interest* (POI) mining experiment directly on mobile devices, thus showcasing how FLI enables *in-situ* big data processing.

5.1 Memory Benchmark

As there is no temporal database (e.g. InfluxDB), available on Android, we compare FLI's performances with SQLite, the only database natively available on Android.

Synthetic data. 2 identical operations are performed with SQLite and FLI: *(i)* the incremental insertion of random samples and *(ii)* the incremental insertion of constant samples. The memory footprint of both solutions on disk is compared when storing timestamped values. As FLI models the inserted samples, random values are the worst-case scenario it can face, while inserting constant values represents the ideal one. One million samples are stored and, for every 10,000 insertion, the size of the file associated with the storage solution is saved. The experiments are done with a publicly available application [19].

Fig. 6 depicts the memory footprint of both approaches. On the one hand, the size of the SQLite file grows linearly with the number of inserted samples, no matter the nature (random or constant) of the samples. On the other hand, the FLI size grows linearly with random values, while the size is constant for constant values. In particular, for the constant values, the required size is negligible. The difference between vanilla SQLite and FLI is explained by the way the model is stored: while SQLite optimizes the way the raw data is stored, FLI is an in-memory stream storage solution, which naively stores coefficients in a text file. Using more efficient storage would further shrink the difference between

the two. As expected, the memory footprint of a data stream storage solution outperforms the one of a vanilla SQLite database in the case of stable values. While random and constant values are extreme cases, in practice data streams produced by ubiquitous devices exhibit a behavior between the two scenarios which allows FLI to lower the memory required to store those data streams.

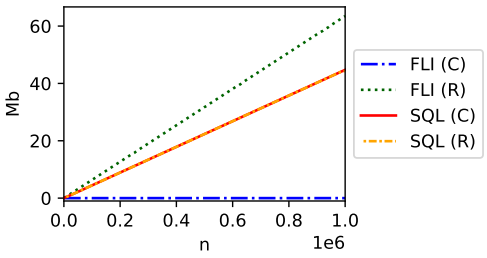


Fig. 6. Inserting $1M$ samples, random (R) or constant (C), in SQLite and FLI.

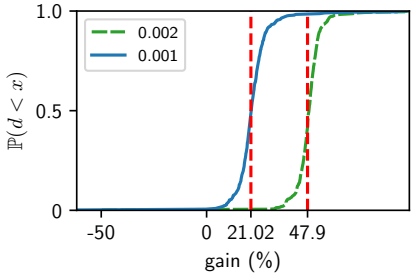


Fig. 7. Memory gain distribution when storing Cabspotting with FLI.

GPS data. We use FLI to store latitudes and longitudes of the entire Cabspotting dataset (388MB) in memory, using both $\epsilon = 10^{-3}$ and $\epsilon = 2 \times 10^{-3}$ (representing an accuracy of approximately a hundred meters). For each user, we compute the gain of memory storage as a percentage, compared to storing the raw traces. Fig. 7 reports on the gain distribution as a CDF along with the average gain on the entire dataset. Most of the user traces largely benefit from using FLI, and FLI provides an overall gain of 21% (307MB) for $\epsilon = 10^{-3}$ on the entire dataset, and a gain of 47.9% (202MB) for $\epsilon = 2 \times 10^{-3}$.

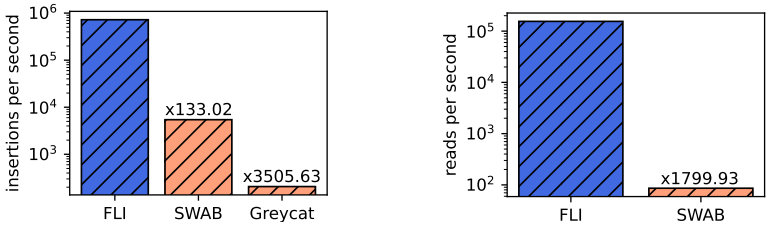
Additionally, we also compare SQLite and FLI to store the entire Privamov dataset (7.2GB). In this context, FLI only requires 25MB (gain of 99.65%) compared to more than 5GB (gain of 30.56%) for SQLite, despite the naive storage scheme used by FLI. Furthermore, with smartphones featuring limited RAM (cf. Table 1) and not allocating the whole of it to a single application, FLI enables loading complete datasets in memory to be processed: on mobile devices, loading the raw Privamov dataset in memory crashes the application (due to out-of-memory errors), while FLI succeeds in fitting the full dataset into RAM. This capability is particularly interesting to enable the deployment of data stream processing tasks on mobile devices that do not incur any processing overhead.

5.2 Throughput Benchmark

We compare FLI with its competitors among the temporal databases: SWAB and Greycat. We study the throughput of each approach in terms of IOPS. Insertion speed is computed by inserting $1M$ random samples (that is each of these solutions' worst-case scenario). For the reads, we also incrementally insert

$1M$ samples before querying $10K$ random samples among the inserted ones. Greycat is an exception: due to its long insertion time (Sect. 4.3), we only insert $10K$ random values and those values are then queried. Our experiment is done using a publicly available application [20].

Fig. 8 depicts the throughput of the approaches for sequential insertions and random reads. On the one hand, FLI drastically outperforms its competitors for the insertions: it provides a speed-up from $\times 133$ against SWAB up to $\times 3{,}505$ against Greycat. The insertion scheme of FLI is fast as it relies on a few parameters. On the other hand, Greycat relies on a costly procedure when a sample is inserted: it tries to increase the degree of the current model until it fits with the new point or until a maximum degree is reached. Greycat aims at computing a model as compact as possible, which is not the best choice for fast online insertions.



(a) Sequential Insertions (b) Random Reads

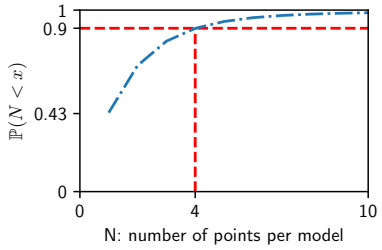
Fig. 8. Throughput for insertions and reads using FLI, SWAB, and Greycat (log scale). FLI drastically outperforms its competitors for insertions and reads.

For the reads (Fig. 8b), FLI also outperforms SWAB. Our investigation reports that FLI largely benefits from its dichotomy lookup inside the time index (see Alg. 1), compared to SWAB, which scans the list of models sequentially until the correct time index is found. SWAB reads have a complexity linear in the size of the list, while FLI has a logarithmic one. Greycat has the same approach as SWAB and this is why it is not represented in the results: with only $10K$ insertions instead of $1M$, its list of models is significantly smaller compared to the others, making the comparison unfair. Nevertheless, we expect Greycat to have a better throughput as its model list shall be shorter.

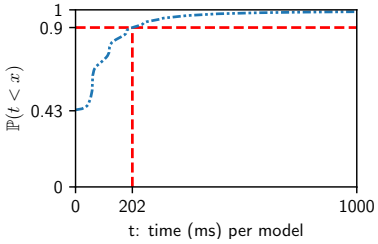
Note that those results have been obtained with the worst-case: random samples. Similarly, unfit for FLI are periodical signals, such as raw audio: our tests show a memory usage similar to random noise. Because FLI leverages linear interpolations, it performs best with signals that have a linear shape (e.g. GPS, accelerometer). We expect SWAB to store fewer models than FLI thanks to its sliding window, resulting in faster reads. However, the throughput obtained for FLI is minimal and FLI is an order of magnitude faster than SWAB for insertions, so it does not make a significant difference. We can conclude that FLI is the best solution for storing large streams of data samples on mobile devices.

5.3 Stability Benchmark

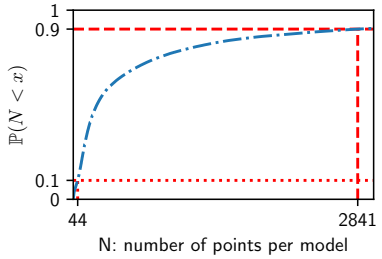
We further explore the capability of FLI to capture stable models that group as many data samples as possible for the longest possible durations. Fig. 9 reports on the time and the number of samples covered by the models of FLI for the CABSPOTTING and PRIVAMOV datasets. One can observe that the stability of FLI depends on the density of the considered datasets. While FLI only captures at most 4 samples for 90% of the models stored in CABSPOTTING (Fig. 9a), it reaches up to 2,841 samples in the context of PRIVAMOV (Fig. 9c), which samples GPS locations at a higher frequency than CABSPOTTING. This is confirmed by Fig. 9b and 9d, which report a time coverage of 202 ms and 3,602 ms for 90% of FLI models in CABSPOTTING and PRIVAMOV, respectively. Given that PRIVAMOV is a larger dataset than CABSPOTTING (7.2 GB vs. 388 MB), one can conclude that FLI succeeds to scale with the volume of data to be stored.



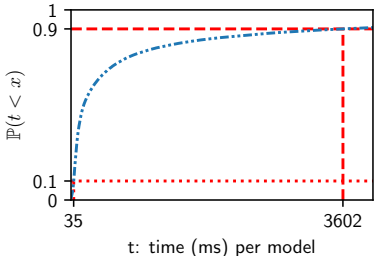
(a) Samples per CABSPOTTING model.



(b) Duration of CABSPOTTING models.



(c) Samples per PRIVAMOV model.



(d) Duration of PRIVAMOV models.

Fig. 9. Stability of the FLI models on PRIVAMOV & CABSPOTTING with $\epsilon = 10^{-3}$.

5.4 Beyond Location Streams

In this paper, FLI was mainly tested against location streams, but our proposal efficiently approximates any type of signal that varies mostly linearly: timestamps, accelerations, temperature, pressure, humidity, light, proximity, air quality, etc. This makes FLI a valuable candidate in ubiquitous contexts, as many

physical quantities captured by e.g. IoT sensors have a piecewise linear behavior. To showcase different scenarios, we benchmark the storage of timestamps and heartbeat data using FLI. In both cases, we use our ϵ-tuning script to capture the most appropriate value to store the data with a reduced error.

Storing timestamps. In all the previous experiments, the timestamps were not modeled by FLI, as we expect the user to query the time at which she is interested in the samples. However, it is straightforward to store irregular timestamps using FLI: we store couples (i, t_i) with t_i being the i^{th} inserted timestamp. The nature of the timestamps makes them a good candidate for modeling, as insertion rates are generally fixed, or vary linearly. To assess the efficiency of FLI for storing timestamps, we stored all the timestamps of the `user 1` of the PrivaMov dataset with $\epsilon = 1$—*i.e.*, we tolerate an error of one second per estimate. The 4,341,716 timestamps were stored using 26,862 models for a total of 80,592 floats and an overall gain of 98%, with a *mean average error* (MAE) of 0.246 second. Hence, not only does the use of FLI result in drastic memory savings, but it also provides accurate estimations.

Storing heartbeat pulses. We downloaded pulse-to-pulse intervals, which oscillate between 500 and 1,100 ms and are reported as the time duration between cardiac pulses to the timestamp of the original pulse, from a *Polar Ignite 2* [16] smartwatch, gathering 28,294,762 samples covering the 12 months of 2023, as a file of 259.1 MB. Using FLI to model this dataset with an accuracy of $\epsilon = 100$ reports on a non-negligible storage space gain of 26.44%, with a *mean error* of 22.74 milliseconds. FLI is thus a suitable solution to store data streams produced by various sensors of wearable and mobile devices, which could find application in e.g. human context recognition [23].

6 Threats to Validity

While FLI enables processing big data in the small by allowing local data storage, our results might be threatened by some variables we considered.

The hardware threats relate to the classes of constrained devices we considered. In particular, we focused on the specific case of smartphones, which is the most commonly deployed mobile device in the wild. To limit the bias introduced by a given hardware configuration, we deployed FLI on both recent Android and iOS smartphones for most of the reported experiments, while we also considered the impact of hardware configurations on the reported performances.

Another potential bias relates to the mobility datasets we considered in the context of this paper. To limit this threat, we evaluated our solutions on two established mobility datasets, Cabspotting and PrivaMov, which exhibit different characteristics. Yet, we could further explore the impact of these characteristics (sampling frequency, number of participants, duration and scales of the mobility traces). Beyond mobility datasets, we could consider the evaluation of other IoT data streams, such as air quality metrics, to assess the capability of

FLI to handle a wide diversity of data streams. To mitigate this threat, we reported on the storage of timestamps and heartbeats in addition to 2-dimensional locations.

Although FLI increases storage capacity through data modeling, it might still reach the storage limit of its host device if using a constant ϵ parameter (which drives the compression rate). To address this issue, we could dynamically adapt data compression to fit a storage size constraint. Toward this end, An *et al.* [1] propose an interesting time-aware adaptive compression rate, based on the claim that data importance varies with its age.

Our implementations of FLI may suffer from software bugs that affect the reported performances. To limit this threat, we make the code of our libraries and applications freely available to encourage the reproducibility of our results and share the implementation decisions we took as part of the current implementation.

Finally, our results might strongly depend on the parameters we pick to evaluate our contributions. While FLI performances (gain, memory footprint) vary depending on the value of the ϵ parameter, we considered a sensitive analysis of this parameter and we propose a default value $\epsilon = 10^{-3}$ that delivers a minimum memory gain that limits the modeling error.

7 Conclusion

Mobile devices are incredible producers of data streams, which are often forwarded to remote third-party services for storage and processing. This data processing pattern might be the source of privacy breaches, as the raw data may leak sensitive personal information. Furthermore, the volume of data to be processed may require huge storage capacity, from mobile devices to remote servers, and network capacity to deal with the increasing number of devices deployed in the wild.

To better deal with the deluge of sensor data streams continuously generated by ubiquitous devices, we proposed FLI that unlocks *in situ* data management strategies by enabling the storage of unbounded data streams. FLI comes as an open library that can be deployed on any mobile device to store multivariate data streams, like mobility traces, 3D accelerations, or air quality metrics.

Our extensive evaluations, based on real mobile applications available for Android and iOS, highlight that FLI drastically outperforms its competitors in terms of insertion throughput—FLI is more than 130 times faster than the traditional SWAB—and read throughput—FLI reads 1,800 times faster than SWAB. Beyond its relevant performances for mobile devices, we also show that the integration of FLI paves the way for the deployment of big data processing tasks on mobile devices, hence addressing the above privacy, network and storage issues in a single solution.

Acknowledgements. This research was supported in part by the Groupe La Poste, sponsor of the Inria Foundation, in the framework of the FedMalin Inria Challenge.

References

1. An, Y., Su, Y., Zhu, Y., Wang, J.: TVStore: Automatically bounding time series storage via Time-Varying compression. In: 20th USENIX Conference on File and Storage Technologies (FAST 22). pp. 83–100. USENIX Association, Santa Clara, CA (Feb 2022), https://www.usenix.org/conference/fast22/presentation/an
2. Berlin, E., Van Laerhoven, K.: An on-line piecewise linear approximation technique for wireless sensor networks. In: IEEE Local Computer Network Conference. pp. 905–912. IEEE (2010)
3. Binder, S.: Drift library. https://pub.dev/packages/drift (2019), last accessed on April 21st, 2024
4. Blalock, D., Madden, S., Guttag, J.: Sprintz: Time Series Compression for the Internet of Things. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies **2**(3) (Sep 2018). https://doi.org/10.1145/3264903
5. Dollinger, V., Junginger, M.: Objectbox database. https://objectbox.io (2014), last accessed on April 21st, 2024
6. Galakatos, A., Markovitch, M., Binnig, C., Fonseca, R., Kraska, T.: Fiting-tree: A data-aware index structure. In: Proceedings of the 2019 International Conference on Management of Data. pp. 1189–1206 (2019)
7. Google: Flutter framework. https://flutter.dev/ (2018), last accessed on April 21st, 2024
8. Grützmacher, F., Beichler, B., Hein, A., Kirste, T., Haubelt, C.: Time and memory efficient online piecewise linear approximation of sensor signals. Sensors **18**(6), 1672 (2018)
9. Inc, T.: Timescale database. https://www.timescale.com (2018), last accessed on April 21st, 2024
10. InfluxData: Influxdb. https://www.influxdata.com/products/influxdb-overview/ (2013), last accessed April 21st, 2024
11. Keogh, E., Chu, S., Hart, D., Pazzani, M.: An online algorithm for segmenting time series. In: Proceedings 2001 IEEE international conference on data mining. pp. 289–296. IEEE (2001)
12. Liu, X., Lin, Z., Wang, H.: Novel online methods for time series segmentation. IEEE Transactions on Knowledge and Data Engineering **20**(12), 1616–1626 (2008)
13. Moawad, A., Hartmann, T., Fouquet, F., Nain, G., Klein, J., Le Traon, Y.: Beyond discrete modeling: A continuous and efficient model for iot. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS). pp. 90–99. IEEE (2015)
14. Mokhtar, S.B., Boutet, A., Bouzouina, L., Bonnel, P., Brette, O., Brunie, L., Cunche, M., D'Alu, S., Primault, V., Raveneau, P., et al.: Priva'mov: Analysing human mobility through multi-sensor datasets. In: NetMob 2017 (2017)
15. Piorkowski, M., Sarafijanovic-Djukic, N., Grossglauser, M.: Crawdad data set epfl/mobility (v. 2009-02-24) (2009)
16. Polar: Ignite 2. https://www.polar.com/en/ignite2 (2021), last accessed on April 21st, 2024
17. Raes, R., Ruas, O., Luxey-Bitri, A., Rouvoy, R.: *Fast Linear Interpolation* accelerometer example application. https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/temporaldb_apps.git&path=temporaldb/example (2022), last accessed on April 21st, 2024

18. Raes, R., Ruas, O., Luxey-Bitri, A., Rouvoy, R.: *Fast Linear Interpolation* implementation. https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/temporaldb_apps.git&path=temporaldb (2022), last accessed on April 21st, 2024
19. Raes, R., Ruas, O., Luxey-Bitri, A., Rouvoy, R.: Memory space benchmarking application. https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/temporaldb_apps.git&path=benchmarking_memory_space (2022), last accessed on April 21st, 2024
20. Raes, R., Ruas, O., Luxey-Bitri, A., Rouvoy, R.: Throughput benchmarking application. https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://gitlab.inria.fr/Spirals/temporaldb_apps.git&path=benchmarking_throughput (2022), last accessed on April 21st, 2024
21. Tamplin, J., Lee, A.: Firebase services. https://firebase.google.com (2012), last accessed on April 21st, 2024
22. Timescale: Building a distributed time-series database on PostgreSQL (Aug 2019), https://www.timescale.com/blog/building-a-distributed-time-series-database-on-postgresql/, last accessed on May 12th 2023
23. Vaizman, Y., Ellis, K., Lanckriet, G.: Recognizing Detailed Human Context in the Wild from Smartphones and Smartwatches. IEEE Pervasive Computing **16**(4) (Oct 2017). https://doi.org/10.1109/MPRV.2017.3971131
24. Wolfson, O., Chamberlain, S., Dao, S., Jiang, L., Mendez, G.: Cost and imprecision in modeling the position of moving objects. In: Proceedings 14th International Conference on Data Engineering. pp. 588–596 (1998). https://doi.org/10.1109/ICDE.1998.655822

A Methodological Transparency & Reproducibility Appendix

A.1 Tools setup

Our benchmarking applications are developed with Flutter, an open-source framework by Google for building multi-platform applications (installation instructions: `https://docs.flutter.dev/get-started/install`); we used Flutter version 3.19.6: to ensure you are using the correct version, check the `flutter --version` command output:

```
user@computer:~$ flutter --version
Flutter 3.19.6 • channel stable • https://github.com/flutter/flutter.git
Framework • revision 54e66469a9 • 2024-04-17 13:08:03 -0700
Engine • revision c4cd48e186
Tools • Dart 3.3.4 • DevTools 2.31.1
```

To run the applications, you can use smartphones (*i.e.* real Android or iPhone devices) or emulators, though performances will be poorer with the latter. In both cases, you will need to install the Android Studio IDE: `https://developer.android.com/studio`.

If you own an Android smartphone, it can be used to test our applications: you need to plug it into your computer using USB and set up development mode: `https://developer.android.com/studio/run/device`. Otherwise, Android Studio contains the Android emulator component: if you need to set up one emulator, instructions can be found here: `https://developer.android.com/studio/run/emulator`.

In both situations, your device should appear in the `adb devices` command output:

```
user@computer:~$ adb devices
List of devices attached
13241JEC208547  device
```

A.2 Applications

Results and figures presented in this paper were obtained using three experimental applications:

- Accelerometer [17]
- Benchmarking memory space [19]
- Benchmarking throughput [20]

TemporalBDDFlutter *(installation time: 2 minutes, run experiment duration: 2 minutes)*

The core library of our contribution, `temporalbddflutter` includes all classes used to model data; this package also includes a toy application modeling accelerometer data with FLI in real-time.

To run the experiment, click the "Launch XP: no movement" button (and do not move your phone if it is a physical device). This will start listening to your device's accelerometer and store its values in FLI models. The "no movement" part of this experiment shows that FLI saves memory space by modeling data instead of storing discrete records, providing an important space gain. You can also try the "move" experiment while moving your phone around: you will see that size gain is lower than the previous experiment, due to data randomness. This experiment has values count and time bounds to stop it automatically after 10k inserted values or 2 elapsed minutes, depending on which limit is reached first.

Benchmarking memory space *(installation time: 2 minutes; run experiment duration: 5 minutes)*

This application allows comparing data sizes of random or constant values, using an SQLite database or a FLI model.

This application allows you to store 1 million values in the phone's memory, using either random or constant values, and either using SQLite or FLI; once an experiment is finished, you can read the size of the file in which values are stored. It demonstrates that FLI modeling performances are approximately the same as SQLite regarding random values, but are superior while storing constant values.

Results obtained by this benchmark are reported in Figure 6.

Benchmarking throughput *(installation time: 2 minutes; run experiment duration: 10 minutes)*

This application allows comparing speeds of inserting or reading values, using FLI, SWAB or Greycat modeling.

Results obtained by this benchmark are reported in Figure 8.