

École polytechnique de Louvain

Differentially Private Stream Processing in Apache Flink

Authors: **Luca HAFHOUF, Alex KHALIL**
Supervisors: **Tom BARBETTE, Étienne RIVIÈRE**
Readers: **Cyrille DUBOIS, Alessandro MAGGI**
Academic year 2025–2026
Master [120] in Computer Science and Engineering

Generative AI Use

As non-native English speakers, we decided to employ generative AI tools during the drafting of this master's thesis strictly for language assistance. Specifically, they helped us polish our sentences and paragraphs to ensure that the main ideas we wanted to convey were expressed clearly and correctly in English.

We emphasize that all the work presented in this thesis is entirely our own. We take full responsibility for the entirety of the content and claims written in this document.

Furthermore, no private or proprietary data was inputted into any generative AI prompts during the course of this research.

Abstract

Privacy protection in the context of real-time applications that continuously process user data, such as IoT sensor networks, healthcare monitoring, or financial fraud detection, is extremely valuable to protect the user. In recent years, Differential privacy (DP) has emerged as the gold standard for formal privacy guarantees, yet its practical deployment in distributed stream processing engines remains largely unexplored. Recent theoretical work aims to close the gap by providing a state-of-the-art DP mechanism for infinite streams. This thesis investigates the integration of the state-of-the-art user-level Differential Privacy algorithm into Apache Flink, an open-source distributed stream processing engine. We deliver a fault-tolerant implementation of the mechanism, coupled with Apache Kafka, and evaluate it on eighteen real-world datasets spanning multiple domains. Through this evaluation, we identify the failure point of the algorithm, caused by skewed user participation in the stream. This leads to stalling in the algorithm’s internal mechanics, which in turn can yield extremely high error and very low utility. We conduct a comprehensive evaluation covering both utility and performance. On the performance side, our system achieves a sustained throughput of approximately 595,000k events/s, incurs a CPU overhead of only 30%, and adds fewer than 5 μ s of operator latency per event, demonstrating that strong user-level privacy guarantees are compatible with high-speed production environments. While the mechanism yields good utility on datasets with large and balanced user populations, it breaks down entirely on datasets where a small number of users dominate the stream. In those cases, the relative error can approach 100%, rendering the released counts practically useless. This reveals a fundamental gap between theory and practice: the formal utility guarantees of the mechanism hold under data conditions that some real-world streams do not satisfy.

Acknowledgements

First, we would like to express our deepest gratitude to our supervisors, Professor Tom Barbette and Professor Étienne Rivière. Their continuous guidance, expertise, and support have been invaluable throughout the realization of this thesis. We are especially grateful for their insightful feedback, as well as the practical advice and tips they provided to help us navigate around roadblocks whenever we felt stuck. Their vision and encouragement pushed us to continually refine and elevate our work.

A very special thank you goes to our direct assistants, Alessandro Maggi and Maxime Vanliefde. We truly appreciate the time and effort they invested in organizing our weekly meetings and consistently following our progress throughout the year. Their dedication to reading our drafts and providing detailed, constructive comments significantly improved the quality and clarity of this manuscript. We could not have navigated the complexities and nuances of this project without their constant help and availability.

We would also like to extend our sincere thanks to Cyrille Dubois for serving as the external reader for this thesis. We are particularly grateful for his guidance during the early stages of our research, his recommendations and pointers toward highly valuable papers in the field of differential privacy gave us a crucial head start in our literature review.

Contents

Generative AI Use	1
Abstract	2
Acknowledgements	3
1 Introduction	7
1.1 Context and Motivation	7
1.1.1 The Importance of Data	7
1.1.2 The Privacy Problem in Data Processing	7
1.1.3 Challenges of Differentially Private Streaming	8
1.1.4 Motivation of This Thesis	9
2 Theoretical Background	11
2.1 Data Stream Processing	11
2.1.1 Distributed Stream Processing	11
2.1.2 Time Semantics and Watermarks	12
2.1.3 Stream Partitioning	12
2.1.4 State Management and Fault Tolerance	13
2.1.5 Streaming Infrastructure	14
2.2 Privacy Before Differential Privacy	17
2.2.1 Classical Privacy Models	17
2.2.2 Why Traditional Models Fail in Adversarial Settings	19
2.3 Differential Privacy	19
2.3.1 Historical Background	19
2.3.2 Intuition Behind Differential Privacy	20
2.3.3 Formal Definition of Differential Privacy	20
2.3.4 ϵ -Differential Privacy	23
2.3.5 Why Differential Privacy is Considered the Gold Standard	26
2.4 Core Mechanisms of Differential Privacy	27
2.4.1 Global Sensitivity (Δf)	27
2.4.2 Laplace Mechanism	28
2.5 Fundamental Properties of Differential Privacy	29
2.5.1 Composition Theorems	29
2.5.2 Group Privacy	30
2.6 Differential Privacy in Streaming Settings	30
2.6.1 Why Streaming Makes DP Hard	30

2.6.2	Event-Level vs User-Level Privacy in Streams	31
2.6.3	Data Hardness	32
2.7	Tools for Differentially Private Streams	32
2.7.1	Binary Tree Mechanism	32
2.7.2	The Sparse Vector Technique (SVT)	33
2.7.3	Telescoping Series for Infinite Epochs	35
2.8	State of the art	36
2.8.1	Continual Observation under Event-Level DP	36
2.8.2	Practical DP Systems	37
2.8.3	Weaker Privacy Relaxations for Streams	38
2.8.4	User-Level DP on Static Datasets	39
2.8.5	Toward User-Level DP for Infinite Streams	39
2.8.6	Closing the Gap: Dong, Luo, and Yi (2023)	40
2.8.7	Summary	40
3	Testing the Limits	43
3.1	Overview	43
3.2	System Architecture and Orchestration	44
3.2.1	Data Ingestion and Event-Time Semantics	44
3.2.2	The Unified DP Coordinator	46
3.2.3	Fault Tolerance and Privacy Budget Integrity	48
3.3	Dataset Selection	49
3.4	Intuition: The Impact of User Cardinality and Distribution	50
3.5	Mathematical Formulation of the Limitation	53
3.5.1	Deconstructing the Baseline Noise Penalty (d_1)	55
3.5.2	Deconstructing the Temporal Penalty (d_2)	56
3.5.3	Combining d_1 and d_2	57
3.5.4	The Consequence: Unbounded Truncation Bias	58
3.6	Estimating the relative error	59
3.6.1	The Halting Epoch k	59
3.6.2	The Concentration Ratio	60
3.6.3	Relative error	61
3.6.4	The Pre-Execution Prediction Algorithm	62
4	Evaluation	63
4.1	Experimental Setup	63
4.1.1	Hardware and Software Setup	63
4.1.2	Metrics	64
4.1.3	Justification of Privacy Hyperparameters	64
4.2	Utility Evaluation	65
4.2.1	Behavior Under Favorable and Adverse Regimes	65
4.2.2	Predicting the Relative Error	69
4.3	Performance Evaluation	72
4.3.1	CPU Overhead	72
4.3.2	Processing Throughput	74
4.3.3	Infrastructure Processing Latency	75
4.4	Discussion	78

5	Conclusion and Future Work	80
5.1	Conclusion	80
5.2	Future Work	81
5.2.1	Extending the Implementation to the Full Query Portfolio	81
5.2.2	Mechanisms Robust to Truncation Lock	82
5.2.3	Distributed and Parallelized Architecture	82
5.2.4	Windowed Differentially Private Aggregations	82
5.2.5	Privacy Budget Management Across Concurrent Queries	82
5.2.6	The Fully-Dynamic Setting: Handling Deletions	83
A	Appendix	92
A.1	Windowing	92
A.2	Privacy Before Differential Privacy	93
A.2.1	Classical Privacy Models	93
A.3	Differential Privacy Mechanisms	95
A.3.1	Gaussian Mechanism	95
A.3.2	Zero-Concentrated Differential Privacy (zCDP) Mechanism	95
A.3.3	w -event Differential Privacy	95
A.4	Hyperparameter Sweep: ϵ vs. θ	96
A.5	Dataset Selection	96
A.6	Utility Evaluation	101

Chapter 1

Introduction

1.1 Context and Motivation

The digitalization of the world has led to an enormous amount of data, which can be collected, analyzed and exploited [1][2]. Modern systems rely on continuous ingestion and process of streams of events coming from different online platforms, devices, IoT sensors[3][4][5]. The ability to process and use this data in real time has become a key competitive factor for most platforms [6]. The processing of large scale personal and sensitive data raises a lot of concerns about security, privacy and ethical data use [7][8].

1.1.1 The Importance of Data

Today, data has become one of the most powerful assets in many cases, especially for decision making and predictions [9]. In the case of streams, organizations can leverage continuous data streams for fraud detection, recommendation algorithms, healthcare and insurance, and monitoring among other cases [10]. In some cases, real-time systems with low-latency pipelines are capable of predicting an outcome as events are taking place [4].

Stream processing frameworks have emerged to attend to those needs, such as Apache Flink [11, 12] and Apache Spark [3, 13], which benefit from scalable and fault-tolerant processing of continuous and unbounded data streams [5]. These frameworks can handle high throughput, stateful computations, and exactly-once processing guarantees.

1.1.2 The Privacy Problem in Data Processing

The problem with the heavy reliance on data analytics means that user personal and sensitive data are continuously being analyzed and processed [14][15]. Interests, location, browsing history, financial decisions and health data are often collected and stored [16], raising serious privacy concerns, as numerous studies have documented cases where such data has been exposed, breached, or even sold to third parties without user consent [17] [18].

To tender to those concerns, anonymization was long considered a practical way to protect privacy by removing explicit identifiers such as names or social security numbers and by

applying heuristic techniques such as **k-anonymity** [19]. In practice, however, privacy risks do not disappear once identifiers are removed, because individuals can often be re-identified by linking seemingly anonymous records with other available information. This was demonstrated by Sweeney [20], who matched voter registration data with anonymized medical records, and by Narayanan and Shmatikov [21], who showed that users in the Netflix Prize dataset could be identified by comparing their movie ratings with external public information, like IMDB user profiles. More broadly, Dinur and Nissim [22] showed that even the release of aggregate statistics can leak sensitive information if insufficient noise is added, while Ganta *et al.* [23] further highlighted the fundamental limitations of anonymization methods. These examples show that the privacy problem is not limited to data collection itself, but also arises during data storage, sharing, and processing, where seemingly harmless releases can still expose sensitive personal information.

The fundamental inability of traditional methods to protect against these types of inference attacks motivated the development of Differential Privacy, introduced by Dwork *et al* [24][25], which provides a mathematically provable privacy guarantee on the data. Namely, Differential Privacy ensures that the presence or absence of a single individual in a dataset does not, to a certain extent, influence the output of a computation. This bounds the risk of inference attacks, regardless of the adversary's access to auxiliary information.

Differential Privacy has become widely regarded as the gold standard for privacy-preserving data analysis. It has been adopted in various industrial and governmental contexts[26][27][28]. However, as stated in the opening sections of Fan and Xiong [29], most work related to that field has focused on static datasets and batch-processing scenarios.

1.1.3 Challenges of Differentially Private Streaming

While Differential Privacy provides a robust foundation for static analysis, real-time streaming environments introduce a qualitatively different set of challenges that have only recently begun to be addressed in the literature [30, 31, 29, 32].

Unbounded Privacy Budget Consumption. In a static setting, a private query consumes a fixed portion of the privacy budget ϵ . In a streaming system, however, results are released repeatedly as new data arrives. Since streams are in principle infinite, a naive application of DP would cause the cumulative privacy cost to diverge over time [25, 33]. Any practical streaming mechanism must therefore "spread" a finite total budget across a potentially infinite sequence of future releases, a problem significantly more complex than the one-shot static case.

Event-level vs. User-level Privacy. A critical distinction arises regarding the unit of protection in the context of Differential Privacy [16]. At the *event level*, the goal is to hide a single record (e.g., one transaction). At the *user level*, the entire history of an individual's contributions must be protected, even if that history spans millions of events accumulated over years. User-level privacy is substantially harder to achieve because the required noise must scale with a user's maximum possible contributions quantity that is not naturally bounded in an infinite stream [31]. Achieving this level of protection without

destroying the utility of a query remains a central open problem in the field.

Temporal Correlation and Differencing Attacks. Streaming aggregations, such as prefix sums or running counts, are inherently correlated. An adversary observing two consecutive outputs at times t and $t + 1$ can compute their exact difference, potentially isolating a single user’s activity [31]. Countering these attacks requires structured mechanisms, such as the Binary Tree Mechanism [30, 31](see Section 2.7.1), which carefully limit how many times any single event can influence the sequence of published results.

Systemic and Engineering Constraints. Integrating formal privacy guarantees into production-grade frameworks like Apache Flink introduces engineering overhead [32]. Flink is built around event-time semantics, watermarks, and stateful processing [5, 11]. A primary challenge is the alignment of privacy accounting with Flink’s fault-tolerance. For instance, a checkpoint recovery that replays data and re-samples noise could lead to "privacy budget double-spending," which would silently invalidate the mathematical guarantees of Differential Privacy.

1.1.4 Motivation of This Thesis

The motivation of this thesis lies in addressing the gap between the significant theoretical progress in differentially private stream processing and the lack of a practical, production-grade implementation within the Apache Flink ecosystem. With the emergence of large-scale data ecosystems in which user behaviors are continuously analyzed and individual-level predictions are generated in real time [34, 35], the demand for principled privacy protection at the stream processing layer has never been greater.

We investigate how the state-of-the-art user-level DP mechanism for continual observation, proposed by Dong, Luo, and Yi [36], can be integrated into Apache Flink [11] as a production-ready module. Our goal is not merely to implement the algorithm, but to stress-test it against the realities of distributed stream processing and real-world data distributions. By doing so, we aim to identify its practical limits and formally characterize the conditions under which it succeeds or fails.

This work makes three concrete contributions to the field of privacy-preserving data analytics:

1. **Production-ready Apache Flink module:** We deliver a complete, fault-tolerant implementation of the Dong *et al.* mechanism within Apache Flink, integrated with Apache Kafka. The system is evaluated on an extensive corpus of eighteen real-world datasets spanning multiple domains and varying levels of contribution skew.
2. **Characterization of *Truncation Lock*:** We identify and formally characterize a previously undescribed failure mode of the mechanism, which we term *Truncation Lock*. We prove that this failure arises pseudo-deterministically under conditions that are common in real-world datasets, particularly those with stabilizing user populations.

3. **Pre-execution prediction algorithm:** We derive a lightweight mathematical model that enables practitioners to estimate the expected utility, or predict the failure, of the mechanism on a given dataset. This algorithm operates using only the user contribution frequency distribution, providing a forecast without the computational cost of a full pipeline deployment.

Ultimately, this work contributes to the ongoing effort of making privacy-preserving real-time data analytics both principled in its guarantees and practical in its application.

Chapter 2

Theoretical Background

2.1 Data Stream Processing

Traditional data processing treats data as a finite, static dataset that is loaded, queried, and discarded. Stream processing, instead, treats data as an unbounded, continuously arriving sequence of records that must be processed incrementally [4]. Modern streaming applications often impose strict timing requirements, processing data as it arrives rather than upon completion of a full dataset scan [5]. To build robust, distributed streaming systems, several foundational concepts must be defined:

- **Record (or Event):** A single, immutable unit of data, typically containing a payload and a timestamp.
- **Stream:** An unbounded, continuously arriving sequence of records.
- **Source:** The entry point of data into the system, responsible for ingesting records from external systems.
- **Sink:** The exit point where the final processed records are emitted to external systems, such as databases or dashboards.
- **Operator:** A continuous processing function applied to the stream to transform, filter, or aggregate the data (see Figure 2.6).
- **Query:** The complete logical pipeline composed of sources, operators, and sinks, which defines the data transformations to be executed.

Figure 2.5 illustrates how these components interact in a typical streaming pipeline.

2.1.1 Distributed Stream Processing

To handle the high throughput of modern data streams, processing engines are deployed as *distributed systems*. Rather than running on a single machine, a streaming query is partitioned and executed in parallel across multiple independent nodes communicating over a network. This architecture provides essential scalability and fault tolerance, but it also introduces physical complexities. Because nodes operate with independent hardware clocks and communicate over networks with variable latency, data is subject to distributed

buffering and transmission delays. As a result, events generated at the same time in the real world rarely arrive at the processing engine simultaneously or in the exact order they were generated.

2.1.2 Time Semantics and Watermarks

Two notions of time are central to the correctness of any stream processing system [5].

Event Time. The timestamp embedded in the record itself, reflecting the exact moment the event actually occurred in the real world.

Processing Time. The wall-clock time of the machine executing the computation at the moment the system receives and processes the record.

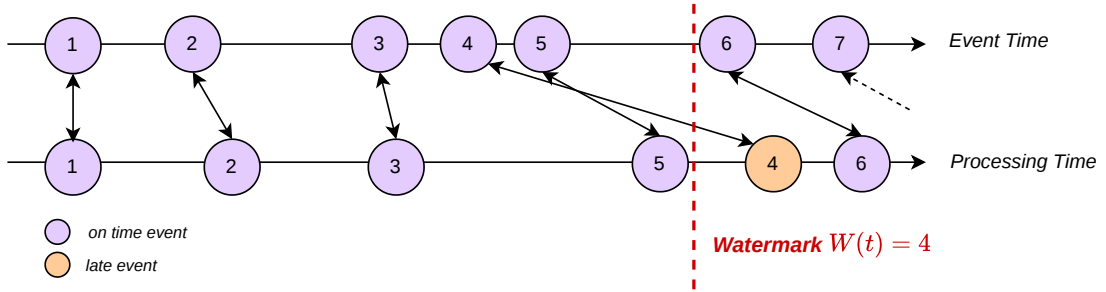


Figure 2.1: Event time vs. processing time with a watermark of value 4

In practice as shown in Figure 2.1, these two timelines continuously diverge due to network latency, device clock skew, and distributed buffering. Consequently, records rarely arrive at the processing engine in the exact order they were generated. To process data correctly according to *event time* while dealing with unbounded out-of-order arrivals, streaming systems rely on a concept called *watermarks* [5, 37].

A watermark is a metadata marker injected into the data stream that signals temporal progress. A watermark with a timestamp t declares that the system assumes all events with an event time $t' \leq t$ have already arrived. An operator can hence perform its time-based computations based on that watermark.

Late arrival events are then taken care of by a heuristic, often *bounded out-of-orderness* where a certain additional time is given after the watermark before dropping the event.

2.1.3 Stream Partitioning

To process unbounded data efficiently across a distributed cluster, systems must divide the stream both logically via partitioning to distribute the workload, and also temporally via windowing (detailed in Appendix A.1) [4].

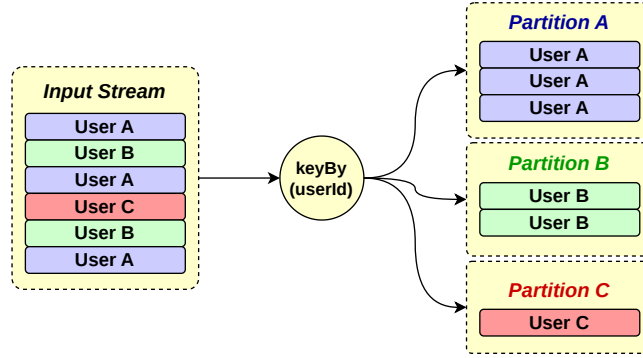


Figure 2.2: Logical stream partitioning via `keyBy(userId)`

Logical Partitioning To parallelize stateful computations and reduce memory pressure on individual nodes, streams are frequently partitioned by a routing key. For instance, partitioning a stream by a `userId`, as illustrated in Figure 2.2, ensures that all events sharing that specific key are deterministically routed to the same physical machine and processed by the same operator instance. This mechanism is a prerequisite for tracking state over time (e.g., aggregating user-level metrics) without relying on centralized, blocking databases.

2.1.4 State Management and Fault Tolerance

Computations over streams are rarely stateless. Answering questions like *how many events has this user generated today?* requires the system to maintain a continuous memory of past events, known as *state*. In distributed streaming, state must be locally available to the processing operator to maintain low latency, but it must also be resilient to machine failures [38].

To prevent data loss or double-counting during crashes, streaming systems must guarantee *exactly-once semantics*. This means that even in the event of a failure, the final computed state reflects each event exactly one time. This is typically achieved through distributed snapshots using algorithms like the Chandy-Lamport algorithm [39], which periodically capture a globally consistent view of the entire stream’s state (including operator variables and stream offsets). If a node fails, the system rolls back to the most recent successful snapshot and replays the stream from that exact point.

As detailed in Figure 2.3, this snapshotting mechanism operates continuously without halting the system. The processing engine periodically injects special control records called *epoch markers* into the data streams. These markers flow inline with normal events and logically divide the infinite stream into finite, numbered chunks (epochs). As a marker flows through the directed acyclic graph, it triggers state persistence:

¹This figure is heavily inspired by the course *LINFO2145 - Cloud Computing* taught by Prof. Étienne Rivière

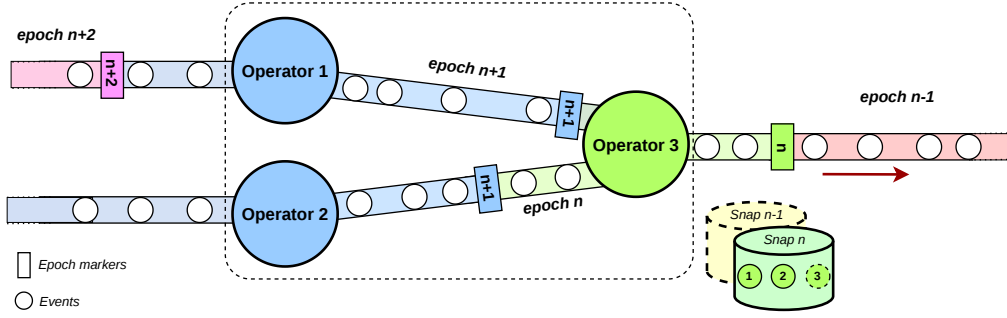


Figure 2.3: Distributed snapshotting in a stream processing engine. Snapshot markers are injected into the stream to separate data logically into epochs¹

- When an upstream operator (e.g., Operator 1 or 2) processes a marker, it saves its current local state to durable storage and forwards the marker downstream.
- When a downstream operator with multiple inputs (e.g., Operator 3) receives a marker, it must wait to receive that same marker from all incoming channels (alignment) before taking its own snapshot and forwarding it.

In the scenario illustrated, the globally consistent snapshot *Snap n* is still in progress. It is waiting for Operator 3 to finish processing the remaining events of epoch *n* before finalizing the state capture for that epoch. Simultaneously, the upstream operators have already moved forward: the epoch markers for *n + 1* are in transit, and subsequent events are actively being processed. This continuous flow ensures that high throughput is maintained without halting the system during backups.

2.1.5 Streaming Infrastructure

To implement the theoretical streaming concepts described above in a production environment, modern architectures typically decouple the data ingestion and storage layer from the data computation layer. In this thesis, we rely on Apache Kafka [3] and Apache Flink [11].

Apache Kafka: Distributed Event Storage

Apache Kafka is a highly scalable and distributed event storage and messaging system acting as a durable and high-throughput buffer. It operates on a publish/subscribe model, decoupling the external systems that generate data from the downstream engines that retrieve it, relying on two entities:

- **Producers:** Client applications that publish (write) continuous streams of events to Kafka. They typically act as adapters, collecting metrics or logs from external systems and routing them into the appropriate categories within Kafka.
- **Consumers:** Client applications (such as a Flink source operator) that subscribe to and retrieve these events. Kafka consumers actively *pull* data at their own pace, which protects downstream systems from being overwhelmed by sudden spikes in data generation.

A Kafka cluster consists of multiple server nodes called *brokers*, which handle the storage and serving of data. Kafka categorizes event streams into *topics*, acting as distributed, append-only logs. To scale horizontally and ensure fault tolerance, a topic is split into a fixed number of *partitions* distributed across different brokers. Partitioning serves two main purposes: it allows multiple consumers to read data in parallel (maximizing throughput), and it provides the fundamental unit of replication, meaning a partition is copied across multiple brokers to prevent data loss in case of hardware failure.

When a producer sends an event, it is routed to a partition deterministically: either by a routing key (e.g., a `userId`) so that related events are appended to the same partition, or distributed round-robin if no key is provided [3]. Once appended, each message is assigned a strictly monotonic, partition-specific ID called an *offset*, as illustrated in Figure 2.4.

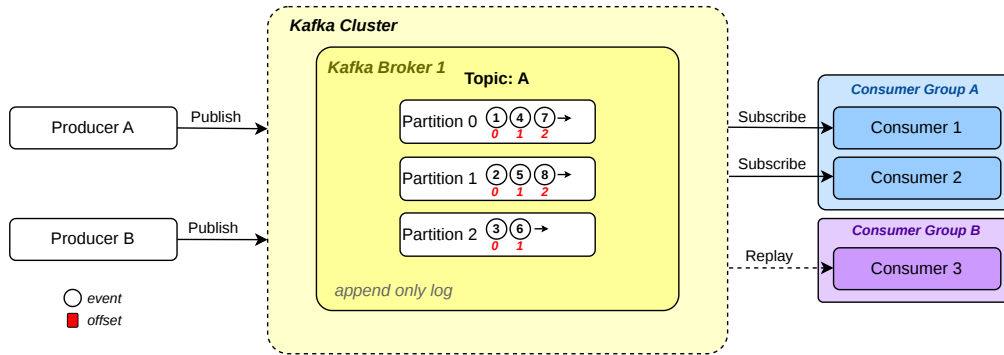


Figure 2.4: Publish and Subscribe architecture with a Kafka Cluster

The number of partitions directly caps the parallelism of consumers: with N partitions, at most N consumers within the same group can read in parallel simultaneously. Therefore, assigning one partition per consumer instance is the ideal configuration for maximum throughput.

Kafka does not delete messages immediately after they are retrieved but retains them for a configured period. This persistent log provides a critical capability: *replayability*. Because consumers explicitly track their position in the log via their offsets they can easily “rewind” time by resetting its offset. This capability is what enables the fault recovery mechanisms discussed in Section 2.1.4, allowing an engine to seamlessly replay historical data after falling back to a snapshot.

Apache Flink: Stateful Stream Processing

Apache Flink [11] is an open-source, distributed stream processing engine for stateful computations over unbounded and bounded data streams. While Kafka handles durable storage and routing of data, Flink executes the complex transformations and aggregations.

Flink’s execution model is based on a Directed Acyclic Graph (DAG) of processing operators. In practice, a Flink job is expressed as a pipeline of operators such as `sources`, `map`,

filter, **aggregations**, and **sinks**, as illustrated in Figure 2.5. While these operators are deployed across a distributed cluster to scale horizontally, the engine guarantees that each individual record flows through the pipeline and is processed sequentially by the operator instances. This subsection explains the main execution semantics for these operators.

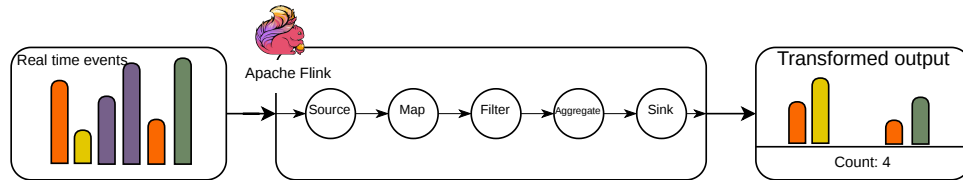


Figure 2.5: Example of a simple Flink dataflow: a source, a sequence of operators (map, filter, aggregate) and a sink.

Event-Time and Watermarks. In the dataflow of Figure 2.5, each operator processes events according to Flink’s time semantics. Flink natively supports event-time processing: it allows developers to extract timestamps from payloads and generate watermarks to handle out-of-order events [5][37]. Watermarks are assigned via a **WatermarkStrategy**. A common choice is **forBoundedOutOfOrderness(δ)**, which declares that events may arrive late by at most a fixed duration δ . In other words, an event is considered late if its event time lags behind the largest event time seen so far by more than δ . Such events are then handled according to a configured policy, typically by being dropped or routed to a side output. The parameter δ therefore is responsible for the trade-off between result latency and tolerance for out-of-order arrivals.

These time semantics apply to all operators in the dataflow graph, including common transformations such as **map**, **filter**, and **aggregations**. Figure 2.6 shows using a toy example how these operators transform the same input stream in different ways.

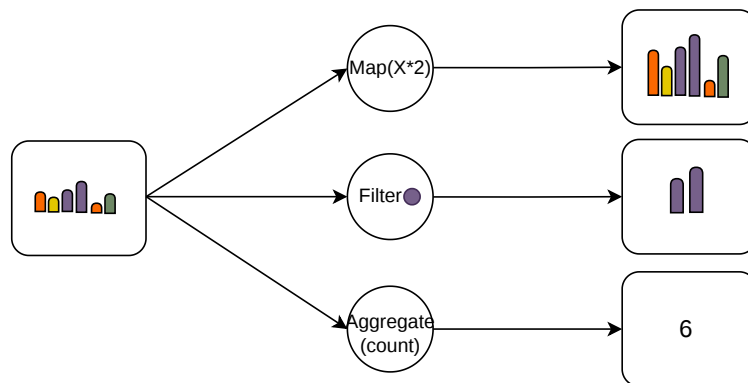


Figure 2.6: Stateless **map** and **filter**, and stateful aggregation on the same input stream.

Logical Partitioning. Using the **keyBy** primitive, Flink partitions a stream based on a routing key (such as a user ID), routing all corresponding records to a specific instance

of a `KeyedProcessFunction`. Flink guarantees that all events sharing the same key are processed strictly sequentially by a single thread. This ensures that developers can safely update state without needing complex locking mechanisms, allowing stateful operations to scale horizontally across a distributed cluster while inherently maintaining thread-safety [4]. A `KeyedProcessFunction` is Flink’s most general stateful operator: for every incoming event, its `processElement` method is invoked, giving the developer explicit access to managed state and to a timer service that can schedule callbacks at a future event-time or processing-time instant. When a registered timer fires, the corresponding `onTimer` callback is invoked. This low-level API is the correct abstraction whenever the processing logic cannot be expressed as a fixed dataflow graph and instead requires dynamic, event-driven coordination between state variables.

Managed State. Flink provides first-class primitives for local state management, such as `ValueState` for single variables and `MapState` for key-value mappings [38]. This state is kept locally in memory or an embedded database like RocksDB [40, 41] for high-performance reads and writes. Because this state is "managed" by the runtime rather than kept in raw JVM variables, Flink can automatically handle state redistribution when the parallelism of the job is changed. Under the hood, Flink divides the key space into a fixed number of buckets called *Key Groups*, defined by the `maxParallelism` parameter. Each parallel operator instance is assigned a contiguous range of these Key Groups. When a job is rescaled, Flink simply redistributes these discrete Key Groups among the new number of operator instances, ensuring state is correctly restored and efficiently rebalanced during recovery.

Fault Tolerance. Recall that Flink guarantees exactly-once processing by implementing a variant of the Chandy-Lamport distributed snapshot algorithm [38, 39]. It periodically injects barriers into the data stream, triggering operators to asynchronously snapshot their Managed State and their current Kafka offsets to durable storage. Upon recovery, the entire DAG is restored to this exact, unified state. This mechanism ensures that even in the presence of machine failures, the system produces results identical to a failure-free execution.

2.2 Privacy Before Differential Privacy

This section provides a condensed overview of traditional privacy failures to motivate the need for Differential Privacy. For a more comprehensive explanation and extended step-by-step examples of these classical models, please refer to the Appendix A.2.

2.2.1 Classical Privacy Models

Before the emergence of differential privacy, protection relied primarily on *syntactic anonymization* by removing explicit identifiers (de-identification) and generalizing attributes so individuals blend into a crowd. However, as Sweeney demonstrated [20], simply removing identifiers is insufficient. Combinations of *quasi-identifiers* (e.g., ZIP code, birth date, sex) can uniquely identify up to 87% of the population, allowing adversaries to link

supposedly anonymous records (Table 2.1) back to real people.

Name	Age	ZIP Code	Sex	Sensitive Attribute (Disease)
****	28	13053	F	Flu
****	29	13068	F	Flu
****	35	13053	M	Cancer
****	36	13068	M	Flu
****	48	14853	F	Diabetes
****	50	14850	F	Cancer

Table 2.1: De-identified Dataset (Vulnerable to Linkage Attacks via Quasi-Identifiers)

To combat linkage attacks, three progressive syntactic models were developed. Their transformations via attribute generalization and suppression, and their flaws are summarized below and visualized side-by-side in Table 2.2:

- **k -Anonymity [19]:** Ensures every record’s quasi-identifiers are indistinguishable from at least $k - 1$ other records.
Flaw: Vulnerable to *homogeneity attacks* (e.g., if all k individuals in a group share the exact same disease, privacy is entirely compromised) and background knowledge attacks.
- **ℓ -Diversity [42]:** Strengthens k -anonymity by requiring at least ℓ distinct sensitive values within each equivalence class.
Flaw: Remains vulnerable if the ℓ sensitive values are semantically similar (e.g., all are types of cancer) or if the global distribution of the dataset is highly skewed.
- **t -Closeness [43]:** Dictates that the distribution of the sensitive attribute within any group must match the global dataset distribution within a threshold t .
Flaw: Achieving this state requires such extreme generalization (grouping entirely disparate users together) that the analytical utility of the dataset is almost always destroyed.

(a) k -Anonymity ($k = 2$)				(b) ℓ -Diversity ($\ell = 2$)				(c) t -Closeness			
Age	ZIP	Sex	Disease	Age	ZIP	Sex	Disease	Age	ZIP	Sex	Disease
20-39	130**	F	Flu	20-39	13***	*	Flu	20-59	1****	*	Flu
20-39	130**	F	Flu	20-39	13***	*	Flu	20-59	1****	*	Flu
20-39	130**	M	Cancer	20-39	13***	*	Cancer	20-59	1****	*	Cancer
20-39	130**	M	Flu	20-39	13***	*	Flu	20-59	1****	*	Flu
40-59	148**	F	Diabetes	40-59	14***	*	Diabetes	20-59	1****	*	Diabetes
40-59	148**	F	Cancer	40-59	14***	*	Cancer	20-59	1****	*	Cancer

Table 2.2: The Evolution and Generalization Cost of Syntactic Anonymization Models

2.2.2 Why Traditional Models Fail in Adversarial Settings

All syntactic approaches share a fundamental conceptual flaw: privacy is treated as a static property of the dataset rather than a property of the computation itself. This leaves three critical gaps.

Vulnerability to Auxiliary Information. Syntactic models assume a “closed-world” adversary. Narayanan and Shmatikov [21] invalidated this assumption by successfully re-identifying individuals in an anonymized Netflix dataset by cross-referencing it with public IMDb ratings. External information can almost always serve as a key to de-anonymize generalized data.

The Fundamental Law of Information Recovery. Dinur and Nissim [22] mathematically proved that answering too many high-accuracy queries inevitably allows an adversary to reconstruct the majority of the original records, regardless of any prior syntactic transformation applied to the data.

Lack of Composability. Repeated releases of anonymized data from the same underlying population progressively erode protections. Ganta *et al.* [23] demonstrated that combining two individually k -anonymous datasets can break the privacy guarantees of both simultaneously.

2.3 Differential Privacy

2.3.1 Historical Background

In 2006 Cynthia Dwork [24] introduced a crucial paradigm shift: if absolute privacy is unattainable, we must redefine privacy guarantees to limit the incremental risk incurred by an individual participating in a dataset instead of preventing all information leakage.

In the paper "Calibrating Noise to Sensitivity in Private Data Analysis" [44], Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith introduced Differential Privacy as a mathematical definition of privacy, building upon the discoveries of Dinur *et al.* [22]. The term *differential privacy* itself was coined in the companion paper by Dwork [24]. Differential Privacy guarantees that the output distribution of an algorithm remains statistically indistinguishable regardless of the presence or absence of any single individual's record in the dataset. By bounding the maximum influence any one individual can have on the outcome (known as global sensitivity see Section 2.4.1), the authors provided a framework where an individual's risk is mathematically decoupled from the arbitrary auxiliary information possessed by an adversary.

While this foundational paper established the strict mathematical boundaries of what is now known as pure or ϵ -Differential Privacy, the framework needed to evolve to support practical and more complex applications. It was later generalized to approximate or (ϵ, δ) -Differential Privacy [45], among others, to allow for a broader class of highly utilitarian algorithms, such as the Gaussian mechanism. Over time, the field has evolved from theo-

retical cryptography into the gold standard for privacy-preserving data analysis, providing the foundational theory required for the continuous data environments encountered in stream processing.

2.3.2 Intuition Behind Differential Privacy

At a high level, Differential Privacy (DP) is about limiting how much the output of an analysis can change when the data of a single individual is added, removed, or modified. Intuitively, it ensures that participating in a dataset does not significantly increase the privacy risk for any one person. An observer looking only at the final (noisy) output should not be able to tell, with high confidence, whether any particular individual contributed their data or not.

The strength of this protection is controlled by a single parameter, typically denoted by ε , often called the privacy budget. In simple terms, ε measures "how private" the mechanism is: smaller values of ε correspond to stronger privacy (outputs with or without a person's data look very similar), while larger values of ε allow more information to pass through and thus offer weaker privacy but higher utility. In practice, choosing ε is a policy decision that reflects how much privacy risk is acceptable for a given application.

Differential privacy can be applied at different levels depending on what we want to protect. One can have the desire to hide whether a particular event (action) occurred or not. For example, consider a streaming system that counts how many times a specific URL was visited in a given hour: here, DP would guarantee that adding or removing a single page view (one click) does not significantly change the published count. An adversary observing the released noisy statistics cannot reliably infer whether that specific click happened.

In contrast, Differential Privacy can protect an entire user's participation, potentially across many events. For instance, in a service that records all URLs visited by a user over a week, DP can guarantee that adding or removing all of one user's visits only slightly affects the published aggregates. From the adversary's point of view, the final output is almost indistinguishable whether the user contributed thousands of events or none at all. This is a strictly stronger guarantee than event-level protection but is also harder to achieve, especially in streaming scenarios where a single user may generate an unbounded number of events.

2.3.3 Formal Definition of Differential Privacy

In order to formally define Differential Privacy, we must first mathematically establish three foundational pillars: the notion of a *dataset*, the *neighboring relation* that captures what it means for two datasets to differ by one person's data, and the notion of a *randomized mechanism* that processes such datasets. Once these are in place, the formal definition follows naturally.

Datasets and Neighboring Relations

Let $\mathcal{D}$ be the universe of all possible datasets. A dataset $D \in \mathcal{D}$ can be represented as a collection of records drawn from a data universe $\mathcal{X}$. The main objective of DP (Differential Privacy) is to conceal the participation of any single *entity*. Therefore, we evaluate the privacy guarantee by considering pairs of datasets that are identical in every respect except for the data of one *entity*. Two datasets D and D' are considered neighboring (or adjacent), denoted as $D \simeq D'$, if they differ by at most one *entity*'s data.

User	Movie	Rating
A	Inception	5
A	Titanic	4
B	Avatar	3
B	Matrix	5
C	Joker	4
A	Batman	4

Table 2.3: Base dataset D with movie ratings from three users.

To ground this mathematical abstraction in practice, consider a dataset D representing a log of movie ratings submitted by users, as shown in Table 2.3.

Two distinctions must be made as to what defines an “entity”:

- **Event-Level Differential Privacy:** The entity is an *event* in this case, and two datasets are adjacent if they differ by a single *entry*, such as one movie rating as shown in Table 2.4. This protects the occurrence of a specific isolated action, but it does not hide the overall presence of a user who may have contributed many entries.
- **User-Level Differential Privacy:** The entity is a *user* in this case, and two datasets are adjacent if they differ by *all* entries associated with a single *user*, as shown in Table 2.5. This provides a much stronger guarantee because it hides whether a user participated at all, regardless of how many records they contributed. However, it is also harder to achieve, especially when a user’s contribution may be large.

User	Movie	Rating
A	Inception	5
A	Titanic	4
B	Avatar	3
B	Matrix	5
C	Joker	4
A	Batman	4

Table 2.4: Neighboring dataset D' under event-level DP: one rating event modified

User	Movie	Rating
A	Inception	5
A	Titanic	4
B	Avatar	3
B	Matrix	5
C	Joker	4
A	Batman	4

Table 2.5: Neighboring dataset D' under user-level DP: User A’s records modified

The exact mathematical interpretation of “differing by one record” categorizes DP into two structural models as formalized by Kifer and Machanavajjhala [46] and highlighted by Near and Abuah [47]:

- **Unbounded Differential Privacy:** D' is constructed by adding or removing one record from D , as shown in Table 2.6. The overall size of the dataset $|D|$ is not public knowledge.
- **Bounded Differential Privacy:** D' is constructed by replacing one record in D with a different record from $\mathcal{X}$, as shown in Table 2.7. The sizes of D and D' are inherently identical and public.

User	Movie	Rating
A	Inception	5
B	Matrix	5
C	Joker	4
A	Batman	4

Table 2.6: Unbounded DP: neighboring dataset D' with records removed

User	Movie	Rating
A	Inception	5
A	Titanic	4
B	Avatar	3
B	Matrix	5
C	Joker	4
D	Nemo	5

Table 2.7: Bounded DP: neighboring dataset D' with record replaced

Under the unbounded model, this neighboring relationship is typically formalized using the L_1 norm. For histograms where x_i represents the number of elements in bin i , D and D' are adjacent if their L_1 distance is exactly 1:

$$\|D - D'\|_1 = \sum_i |D_i - D'_i| \leq 1$$

Mechanisms

Having established what it means for two datasets to be neighbors, we next define the objects that process them. As discussed in Appendix A.2.1, Dinur and Nissim [22] proved that any useful privacy-preserving mechanism must be *randomized*: a deterministic algorithm will inevitably leak information under repeated or combined queries. Randomness is therefore not a design choice but a mathematical necessity.

Let $\mathcal{M}$ be a randomized algorithm (called a *mechanism* in the DP literature) that takes a dataset $D \in \mathcal{D}$ as input and produces an output $y \in \mathcal{R}$, where $\mathcal{R}$ is the range of all possible outputs. Because $\mathcal{M}$ is randomized, running $\mathcal{M}(D)$ multiple times produces a *probability distribution* over $\mathcal{R}$, rather than a single fixed answer.

The Formal Definition of Differential Privacy

Having introduced datasets, neighboring relations, and randomized mechanisms, we can now state the formal guarantee provided by Differential Privacy as introduced by Dwork, McSherry, Nissim, and Smith [44].

Definition 2.3.1 A randomized algorithm $\mathcal{M}$ satisfies ε -differential privacy if, for all neighboring datasets $D, D' \in \mathcal{D}$ differing by at most one element, and for all subsets of possible outputs $S \subseteq \mathcal{R}$, the following inequality holds:

$$\Pr[\mathcal{M}(D) \in S] \leq \exp(\varepsilon) \cdot \Pr[\mathcal{M}(D') \in S]$$

At a high level, this inequality says that the mechanism behaves almost the same, up to a certain bound proportional to ε , whether or not a single *entity*'s data is included in the input dataset.

The set $S \subseteq \mathcal{R}$ is an arbitrary collection of possible outputs. The definition requires the inequality to hold for *every* measurable S , not just for individual outcomes. This ensures that an adversary cannot weaken the guarantee by focusing on very specific ranges of outputs or by post-processing and combining multiple outputs in a clever way.

The parameter $\varepsilon \geq 0$, is called the *privacy budget* or *privacy loss*, and controls how similar the two output distributions must be. Its use will be explained in more details in Subsection 2.3.4.

Definition 2.3.1 formalizes the idea of *plausible deniability* [24]. If an adversary observes an output $y \in S$, then under ε -DP that observation is almost as likely whether the mechanism was run on D (where a given individual is present) or on D' (where that individual is absent). In other words, the adversary gains at most a small multiplicative amount of additional confidence about any single entity's participation, no matter what external information they have. This is the core guarantee offered by Differential Privacy.

2.3.4 ε -Differential Privacy

The formulation in Definition 2.3.1 is the original framework, known as *pure* or ε -*differential privacy*. This section explores the practical meaning of the privacy budget ε , the inherent trade-off between privacy and utility, and the structural constraints this strict definition imposes on algorithm design.

The Privacy Loss Random Variable

For any neighboring pair $D \simeq D'$ and any output $y \in \mathcal{R}$, we can measure the *privacy leakage* associated with observing y via the log-likelihood ratio, known as the **privacy loss random variable** [25]:

$$\mathcal{L}_{\mathcal{M}(D \parallel D')}(y) = \ln \left(\frac{\Pr[\mathcal{M}(D) = y]}{\Pr[\mathcal{M}(D') = y]} \right)$$

This quantity measures how much more likely the output y is under D than under D' : a positive value means y is more consistent with D , while a negative value means it is more

consistent with D' . Under pure ε -DP, this ratio is bounded in absolute value for *every* possible output and *every* pair of neighbors:

$$\left| \mathcal{L}_{\mathcal{M}(D \parallel D')}(y) \right| \leq \varepsilon \quad \forall y \in \mathcal{R}, \forall D \simeq D'$$

Equivalently, this bound guarantees that the presence or absence of any single individual's data can shift the probability of any given outcome by at most a multiplicative factor of $\exp(\varepsilon)$.

The Privacy-Utility Trade-off

The privacy budget ε is the central dial that governs the balance between mathematical privacy protection and data utility, because it directly dictates the scale of noise added to the query outputs. To understand the real-world meaning of this parameter, it is highly intuitive to analyze it through the lens of an adversary attempting to infer a single individual's presence in a dataset, as popularized by Desfontaines [48].

Suppose an adversary wants to determine whether a target user's records are included in the dataset. Under the unbounded, user-level DP framework established in Section 2.3.3, this is mathematically equivalent to distinguishing whether the stateful mechanism $\mathcal{M}$ is operating on dataset D_{in} (target present) or D_{out} (target absent), where $D_{\text{in}} \simeq D_{\text{out}}$ are neighboring datasets differing by only one user's history.

Let the adversary have an initial suspicion (prior belief) that the target user is present in the dataset, represented as the probability $p_{\text{prior}} = \Pr[D = D_{\text{in}}]$. Correspondingly, their prior belief in the target's absence is $1 - p_{\text{prior}}$. Once the randomized mechanism releases an output $y \in \mathcal{R}$, the adversary observes this value and updates their certainty. This updated suspicion (posterior belief) is defined as $p_{\text{post}} = \Pr[D = D_{\text{in}} \mid \mathcal{M}(D) = y]$.

Because $\mathcal{M}$ satisfies ε -differential privacy, the probability of any given output occurring under D_{in} compared to D_{out} can vary by at most a multiplicative factor of e^ε . Applying this core probability ratio within a standard Bayesian update yields strict mathematical bounds on the adversary's updated suspicion:

$$\frac{p_{\text{prior}}}{e^\varepsilon + (1 - e^\varepsilon)p_{\text{prior}}} \leq p_{\text{post}} \leq \frac{e^\varepsilon p_{\text{prior}}}{1 + (e^\varepsilon - 1)p_{\text{prior}}} \quad (2.1)$$

Equation 2.1 and Figure 2.7 provide a clear and quantitative measure of the risk-utility trade-off associated with different ε values:

- **Under a strong privacy regime ($\varepsilon = 1$):** If an adversary begins with neutral uncertainty ($p_{\text{prior}} = 0.50$), their updated suspicion p_{post} is guaranteed to stay within the narrow band of $[0.27, 0.73]$. The individual's participation remains highly deniable.
- **Under a permissive privacy regime ($\varepsilon = 5$):** An adversary with the same initial suspicion ($p_{\text{prior}} = 0.50$), can have their updated suspicion rise to as high as 0.94 after observing a single query response. This illustrates how quickly theoretical protection degrades as ε increases.

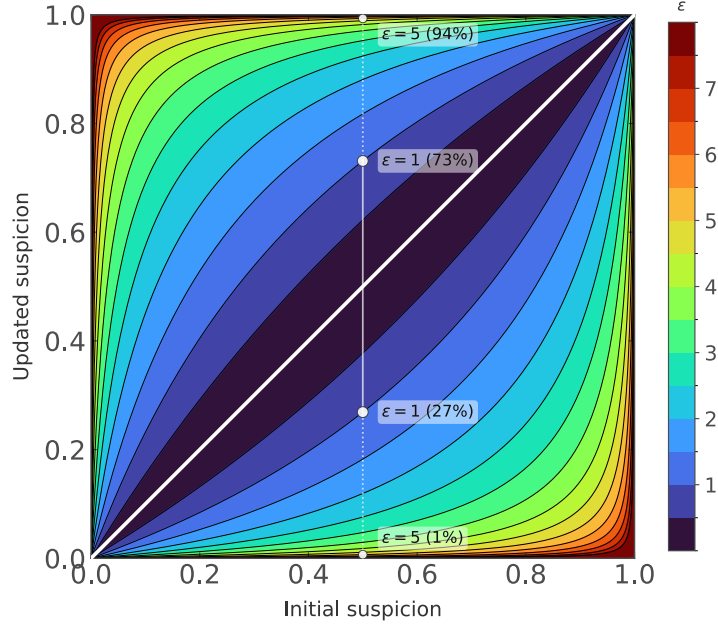


Figure 2.7: Lower and upper bounds on the updated suspicion from a prior suspicion

Crucially, these bounds represent a *theoretical* worst-case scenario where an adversary possesses perfect background knowledge of the other $n - 1$ records in the dataset. In practical deployments, adversaries rarely have access to such detailed auxiliary information.

To preserve critical data utility, prominent organizations frequently deploy privacy budgets that far exceed the theoretical $\varepsilon \leq 1$ ideal. For instance, Apple [49] collects user telemetry with daily budgets ranging from $\varepsilon = 4$ to 16. LinkedIn [50] utilizes $\varepsilon = 14.4$ to publish its Labor Market Insights, and the U.S. Census Bureau [51] allocated budgets between $\varepsilon = 19.46$ and 45.68 for its 2020 demographic data releases. These real-world deployments highlight a practical consensus: while higher budgets appear mathematically permissive under worst-case assumptions, they are often necessary to secure actionable utility against realistic adversaries.

Structural Constraints and the Failure Case

Pure ε -differential privacy enforces a strict structural constraint: an algorithm $\mathcal{M}$ must have the exact same set of possible outputs (support) for any two neighboring datasets. If an output is possible under dataset D but impossible under D' , the privacy loss diverges to infinity, resulting in a catastrophic privacy failure.

To illustrate this, consider a simple count query of patients with a disease, applied to two neighboring datasets:

Suppose we attempt to privatize the query by adding bounded discrete noise, randomly drawn from the set $\{-1, 0, +1\}$.

- For D , the possible noisy outputs are $\{1, 2, \mathbf{3}\}$.
- For D' , the possible noisy outputs are $\{\mathbf{0}, 1, 2\}$.

User	Has Disease	User	Has Disease
Alice	Yes		
Bob	Yes	Bob	Yes

Dataset D (True Count = 2) **Dataset D'** (True Count = 1)

Table 2.8: Neighboring datasets differing by Alice’s presence.

If an adversary observes the output $y = 3$, they know with absolute certainty that the underlying dataset must be D , definitively revealing Alice’s presence. The same logic applies if $y = 0$, which perfectly exposes her absence. This, in term, makes the privacy loss random variable 2.3.4 diverge to $\pm\infty$, allowing an adversary to bypass the privacy budget entirely which violates the pure ε -DP guarantee [25].

This forces pure ε -DP mechanisms to use noise distributions whose support is unbounded in both directions, of which the *Laplace mechanism* (Section 2.4.2) is the canonical example. Gaussian noise, while preferable for its compositional properties, cannot satisfy this strict requirement for any finite ε [45]. Relaxing the guarantee to permit a small failure probability $\delta > 0$ yields *approximate* (ε, δ) -differential privacy, which we discuss in Appendix A.3.1.

2.3.5 Why Differential Privacy is Considered the Gold Standard

Several mathematical properties collectively make differential privacy the dominant framework for privacy-preserving data analysis.

Post-processing Immunity

Any deterministic or randomized function applied to the output of a differentially private mechanism remains differentially private, without incurring any additional privacy cost [25]. This theorem ensures that the results can be freely transformed, combined with public information, or visualized without weakening the original privacy guarantee.

Composition

One of the most powerful features of DP is that it is *composable*. Multiple differentially private mechanisms applied to the same dataset can be combined with a quantifiable and bounded total privacy cost, a property explored in detail in Subsection 2.5.1. This allows researchers to build complex, multi-stage pipelines from simpler private components, each contributing a known share of the total privacy "budget".

Robustness to Auxiliary Information

Unlike syntactic models, the DP guarantee holds regardless of the external knowledge or "side-channel" information an adversary may possess. Because the bound on privacy loss is absolute and mathematical, it is not conditional on unstated assumptions about what the adversary does or does not know [24] [25].

Formal Provability and Adoption

Because the guarantee is a mathematical theorem, it enables rigorous reasoning and systematic communication of privacy properties to regulators and users. These properties have led to real-world deployments at an unprecedented scale, with implementations by the U.S. Census Bureau, Google, Apple, and Microsoft [26, 52].

2.4 Core Mechanisms of Differential Privacy

2.4.1 Global Sensitivity (Δf)

The core insight of the Laplace mechanism is that the amount of noise required to hide any individual's contribution is determined by the maximum influence they could possibly have on the query output. This is formalized through the concept of global sensitivity.

Definition and L_1 sensitivity

Global sensitivity measures the maximum change in the output of a function f across any two neighboring datasets.

Definition 2.4.1 (L_1 Global Sensitivity) *For a function $f : \mathcal{D} \rightarrow \mathbb{R}^k$, the L_1 global sensitivity $\Delta_1 f$ is defined as:*

$$\Delta_1 f = \max_{D \approx D'} \|f(D) - f(D')\|_1 \quad (2.2)$$

where the maximum is taken over all pairs of neighboring datasets $D, D' \in \mathcal{D}$ [44].

Sensitivity depends critically on the choice of privacy unit (user or event level) established in Subsection 2.3.3. Under event-level privacy, where D and D' differ by a single record, the sensitivity represents the impact of one event. Under user-level privacy, where D and D' differ by all records associated with a single user, the sensitivity must account for the user's total possible contribution, denoted by κ .

Calculation Sensitivity for Common Queries

The calculation of $\Delta_1 f$ varies based on the type of aggregation being performed:

- **Count Queries:** A count query $f(D)$ returns the number of records satisfying a predicate. Under event-level privacy, $\Delta_1 f = 1$, as adding or removing one record changes the count by exactly one. Under user-level privacy, if a user can contribute at most κ records, the sensitivity is $\Delta_1 f = \kappa$.
- **Sum Queries:** For a sum query over values in the range $[a, b]$ the event-level sensitivity is $\max(|a|, |b|)$. If the values are not bounded, the global sensitivity is technically infinite, which is why clipping (truncation) is a prerequisite for differentially private sums. At the user-level, the sensitivity becomes $\kappa \times \max(|a|, |b|)$.

- **Mean Queries:** The mean is a non-linear function, and its global sensitivity is generally high because a single outlier can significantly shift the average. In practice, means are usually computed as a *composite query*: a private sum divided by the private count, each with its own sensitivity and privacy budget.
- **Histograms:** A histogram consists of k non-overlapping bins. In the unbounded DP model (where datasets differ by the addition/removal of a record), a single record affects only one bin. Thus, the L_1 sensitivity of a histogram is the same as a count query: 1 for event-level and κ for user-level.

This scaling of sensitivity by κ is precisely why user-level privacy requires substantially more noise than event-level privacy.

2.4.2 Laplace Mechanism

The Laplace mechanism achieves pure ε -differential privacy by perturbing the output of a query function with noise drawn from a Laplace distribution. The magnitude of this noise is directly proportional to the L_1 sensitivity of the query and inversely proportional to the privacy budget ε .

Formal Definition

For a function $f : \mathcal{D} \rightarrow \mathbb{R}^k$ with L_1 sensitivity $\Delta_1 f$ and a privacy parameter $\varepsilon > 0$, the Laplace mechanism $\mathcal{M}_L$ is defined as:

$$\mathcal{M}_L(D) = f(D) + (\eta_1, \dots, \eta_k) \quad (2.3)$$

where each η_i is an independent and identically distributed (i.i.d.) random variable sampled from the Laplace distribution $\text{Lap}(b)$ with scale parameter $b = \Delta_1 f / \varepsilon$ [44].

The probability density function of the Laplace distribution centered at 0 with scale b is given by:

$$\text{pdf}(x|b) = \frac{1}{2b} \exp\left(-\frac{|x|}{b}\right), \quad \sigma^2 = 2b^2 = 2(\Delta_1 f / \varepsilon)^2 \quad (2.4)$$

By scaling the noise to $\Delta_1 f / \varepsilon$, the mechanism ensures that any change in the true output induced by a single individual (up to $\Delta_1 f$) is statistically masked by the noise to an extent controlled by the chosen privacy budget [25].

Theoretical Importance for User-Level DP

The Laplace mechanism is essential to the architecture of user-level streaming systems for two primary reasons. First, it achieves *pure ε -DP*, meaning the privacy bound holds for every possible output with no failure probability. This is a stricter guarantee than the approximate (ε, δ) -DP provided by mechanisms like the Gaussian mechanism.

Second, and most importantly for this work, pure ε -DP mechanisms interact cleanly with **group privacy** section 2.5.2. Because the group privacy amplification for pure DP is

exact and linear, a mechanism that provides event-level protection can be mathematically "lifted" to a user-level guarantee by scaling the noise according to the contribution bound κ .

2.5 Fundamental Properties of Differential Privacy

A critical advantage of differential privacy over syntactic models is its mathematical predictability. The following properties allow for the design of complex privacy systems by combining simpler building blocks while maintaining a quantifiable bound on the total privacy loss.

2.5.1 Composition Theorems

Composition is the primary tool for tracking accumulated privacy loss in systems that release multiple outputs. This is the defining challenge in stream processing, where new results are published continually and the budget must be managed over time.

Basic Composition

Basic composition is a fundamental property of differential privacy. If a sequence of mechanism $\mathcal{M}_1, \dots, \mathcal{M}_k$ are applied to the same dataset, where each $\mathcal{M}_i$ satisfies ε_i -DP, their joint release satisfies $(\sum \varepsilon_i)$ -DP [25].

In the streaming setting, this property presents a significant challenge: as the number of releases k approaches infinity, the total privacy cost $\sum \varepsilon_i$ diverges. To prevent a total breach of privacy over an infinite horizon, any practical streaming system must ensure that the sum of the privacy budget over time converges to a finite value. This requires a carefully designed budget allocation strategy, where the budget for each step or epoch decreases according to a convergent series.

Advanced Composition

To achieve better scaling for a large number of queries k , *advanced composition* provides a tighter bound. For k mechanisms each satisfying (ε, δ) -DP, the sequence satisfies $(\varepsilon', k\delta + \delta')$ -DP for any chosen $\delta' > 0$. If the base mechanisms satisfy pure (ε) -DP (meaning $\delta = 0$), the overall guarantee becomes (ε', δ') -DP, where the total privacy cost ε' grows as $O(\varepsilon\sqrt{k \log(1/\delta')})$ rather than $k\varepsilon$ [53].

This sublinear growth is significantly more favorable for high-frequency data releases, as it allows for a much higher number of queries for the same total privacy budget. However, advanced composition requires a relaxation into approximate (ε, δ) -DP. While it offers superior utility in many settings, the square-root scaling makes the relationship for group privacy (section 2.5.2) non-linear, which can complicate mechanisms that rely on exact budget transformations to move from event-level to user-level guarantees.

Parallel Composition

When mechanisms are applied to disjoint (non-overlapping) subsets of a dataset, the total privacy cost does not sum. Instead, the joint release satisfies $(\max \varepsilon_i)$ -DP [25]. This property is frequently exploited in distributed and sharded systems, if a stream is partitioned by keys such that each key contains a unique set of users, queries on those partitions can be executed in parallel without additive budget consumption.

2.5.2 Group Privacy

Differential privacy naturally extends to groups of individuals. A mechanism satisfying ε -DP, for a single individual automatically satisfies $k\varepsilon$ -DP for a group of k individuals [25]. Formally, if D and D' are neighboring datasets that differ by the data of at most k users:

$$\Pr[\mathcal{M}(D) \in S] \leq \exp(k\varepsilon) \cdot \Pr[\mathcal{M}(D') \in S] \quad (2.5)$$

This property is fundamental to user-level streaming. If a mechanism achieves event-level ε -DP, it automatically provides user-level $(\kappa\varepsilon)$ -DP for any user who contributes at most κ events. Removing all κ events of a single user is mathematically equivalent to κ consecutive single-event removals, each protected by the event-level guarantee.

Consequently, to achieve a fixed target user-level budget of ε_{user} through group privacy, the mechanism must allocate a per-event budget of $\varepsilon_{user}/\kappa$. This relationship demonstrates that the maximum user contribution κ is the dominant factor in the utility of user-level mechanisms. As κ increases, the amount of noise required to mask a user's total influence grows linearly, which represents a fundamental trade-off between the strength of the privacy unit and the resulting accuracy of the query.

2.6 Differential Privacy in Streaming Settings

While the foundations of differential privacy were originally developed for static datasets [24][44], the paradigm of continual observation, where query results are updated and released as new data arrives, introduces mathematical and systemic complexities.

2.6.1 Why Streaming Makes DP Hard

These challenges were first formally identified by Dwork *et al.* [30] and Chan *et al.* [31], who established that three interrelated issues distinguish the streaming setting from the static case.

Unbounded Composition

As established in section 2.5.1, every differentially private release consumes a portion of the privacy budget. In a streaming context, naive application of this property leads to an unbounded accumulation of privacy loss. For an infinite stream ($t \rightarrow \infty$), the

total budget $\sum \varepsilon_i$ diverges, meaning any finite budget will eventually be exhausted [30]. Consequently, mechanisms are designed to "spread" a fixed total budget across a potentially infinite number of future releases, a challenge that requires fundamentally different budget management strategies than the single-release static setting.

Continual Observation

In the static setting, an adversary observes only a single output. In the streaming setting, the adversary observes a sequence of outputs over time. Even if each individual output is private, the sequence of outputs can leak information. For example, an adversary observing consecutive counts at time t and $t + 1$ can compute the exact change between those steps, potentially revealing a single user's record. To prevent this, the mechanism must ensure that the global privacy guarantee holds across the entire output transcript rather than just individual time steps [31].

Correlated Outputs

Streaming aggregations, such as running sums or counts, are typically prefix sums. This means that the output at time t is a function of all events from time 1 to t . Because outputs are highly correlated over time, privacy accounting cannot treat releases as independent observations. The "Tree-Based Approach" was specifically developed to manage these correlations by decomposing the stream into segments, ensuring that any single event contributes to at most $O(\log T)$ published sums [30][31].

2.6.2 Event-Level vs User-Level Privacy in Streams

The distinction between privacy units, introduced in section 2.3.3, has significant consequences for utility in a streaming context. This choice determines whether the system protects an isolated action or the user's entire temporal footprint [25].

Event-Level DP for stream

This model was the focus of the earliest foundational research into continual observation [30][31]. It is computationally efficient because the sensitivity of most counting queries remains $\Delta f = 1$, allowing for high utility with minimal noise. However, it provides no protection for users who contribute multiples events over time. As an adversary observes the long-run sequence of outputs, they can often infer a user's behavioral patterns, peak activity times, or total contribution scale [54]. Consequently, while event-level DP is practical for large-scale processing, it represents a relatively weak guarantee for long-term participation in a stream.

User-Level Differential Privacy

The primary challenge of this model in a streaming context is that the global sensitivity becomes proportional to the maximum number of events any single user can contribute, denoted as $\kappa(D_t)$. Because $\kappa(D_t)$ can grow without bound in an infinite stream, traditional noise calibration becomes impossible without either enforcing a fixed contribution cap or estimating κ adaptively. While a fixed cap ensures a bounded sensitivity at the cost of

introducing systemic bias (through data clipping), adaptive estimation allows the noise to scale with the actual data hardness. This distinction represents a fundamental trade-off in the field: mechanisms must balance the risk of excessive noise against the bias introduced truncating highly active users.

2.6.3 Data Hardness

The *data hardness* of a stream prefix D_t is defined by its maximum user contribution [36]:

$$\kappa(D_t) := \max_{u \in \mathcal{U}} |\{(x_i, u_i) \in D_t : u_i = u, x_i \neq \perp\}| \quad (2.6)$$

where $\mathcal{U}$ is the set of all users, (x_i, u_i) represents an event consisting of an item x_i and its associated user u_i , and $\perp$ denotes a dummy (absent) record.

Under user-level DP, the global sensitivity of a count query is $\kappa(D_t)$, necessitating Laplace noise of magnitude $\Theta(\kappa(D_t)/\varepsilon)$ (Section 2.4.2). In additive streaming settings, $\kappa(D_t)$ is dynamic and monotonically non-decreasing. Relying on fixed bounds is risky: underestimation introduces *truncation bias*, while overestimation destroys utility via excessive *variance*.

2.7 Tools for Differentially Private Streams

2.7.1 Binary Tree Mechanism

The Binary Tree Mechanism (BTM) addresses a fundamental challenge in continual observation: answering a continuous stream of prefix queries (i.e., the running count of events up to time t) without accumulating $O(t)$ noise. If a mechanism were to add noise to each individual timestamp and sum them, the error would grow linearly. Instead, by structuring time into a binary hierarchy, we reduce the number of noisy components required to reconstruct any prefix sum to $O(\log t)$.

Infinite Forest

Dwork et al [30] used a fixed stream horizon T in order to be able to build their Binary Tree. To handle infinite streams where the maximum time horizon T is unknown, Dong *et al.* [36] cleverly implement the BTM as a dynamically growing *forest*. The time axis is partitioned into disjoint dyadic periods of exponentially increasing length:

$$[2^\ell, 2^{\ell+1} - 1], \quad \ell = 0, 1, 2, \dots$$

As shown in Figure 2.8, within each period ℓ , a finite binary tree of depth $\ell + 1$ is instantiated. Each node v covering the time interval $[a_v, b_v]$ maintains an exact sum of the valid events that occurred during that window:

$$c_v = \sum_{i=a_v}^{b_v} 1$$

To ensure differential privacy, the *released* value for each node is a noisy counter $\tilde{c}_v = c_v + \eta_v$, where η_v is drawn from a Laplace distribution.

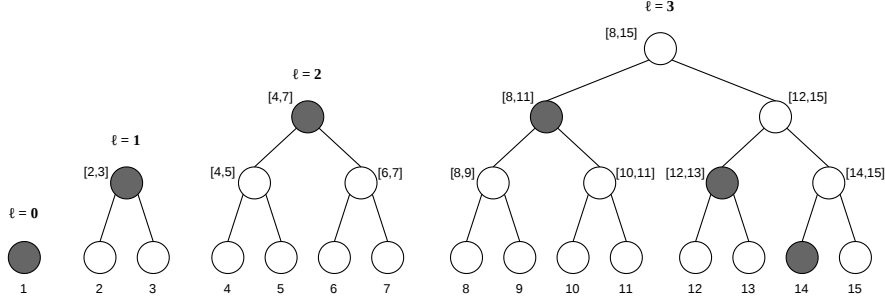


Figure 2.8: Binary Tree Mechanism as seen in the paper by Dong *et al.* [36]

Privacy Budget Allocation

the dyadic periods are disjoint, any single event falls into exactly one period. By the parallel composition theorem of differential privacy as seen in Section 2.5.1, the mechanism can safely allocate the *full* privacy budget ε_i available to each tree in the forest independently.

Each tree starting at a period ℓ consists of $\ell + 1$ levels. Since each level can participate in one node per level only, using the rules of basic composition, see Section 2.5.1, the budget can be split evenly into $\varepsilon' = \varepsilon/(\ell + 1)$ across all levels.

Consequently, the noise added to each node in the ℓ -th tree is scaled to the depth of that specific tree:

$$\eta_v \sim \text{Lap}\left(\frac{\ell + 1}{\varepsilon}\right)$$

Query Answering

To answer a prefix query at time t , the mechanism reconstructs the sum using the dyadic decomposition of the interval $[1, t]$. This is achieved by summing the single root counters of all fully completed past periods, alongside a minimal subset of internal nodes from the currently active tree.

Based on the Figure 2.8 and for $t = 14$, the exact count would be the sum of the root nodes for periods $\ell \in \{0, 1, 2\}$, and the nodes $[8, 11]$, $[12, 13]$ and $[14]$ for period $\ell = 3$.

From Event-Level to User-Level

To elevate the tree guarantees from *event-level* to *user-level* differential privacy, we simply feed it a dynamically truncated version of the stream, ensuring no single user exceeds the *Global Sensitivity* 2.4.1 contribution threshold.

2.7.2 The Sparse Vector Technique (SVT)

The Sparse Vector Technique (SVT) addresses a limitation of standard differential privacy composition: evaluating a potentially infinite sequence of queries without exhausting the privacy budget. SVT is cleverly designed to only output whether a query exceeds a

predefined threshold, consuming privacy budget *only* when the threshold is crossed.

Core Mechanism

Given a potentially infinite sequence of queries $Q = (q_1, q_2, \dots)$ and a public threshold η , to guarantee privacy, SVT adds Laplacian noise to *both* the incoming queries and the threshold.

To understand its mechanics, consider an example where a system tracks how many users have exceeded a contribution limit. The system wants to trigger an update when exactly $\eta = 1000$ users reach this limit. First, the mechanism generates a noisy threshold by adding a random noise value ρ :

$$\tilde{\eta} = \eta + \rho, \quad \rho \sim \text{Lap}\left(\frac{\Delta f}{\varepsilon_1}\right)$$

Suppose the drawn noise is $\rho = 15$, making our noisy threshold $\tilde{\eta} = 1015$.

Then, as each query q_j arrives (representing the current true count of users exceeding the limit), the mechanism perturbs it with its own noise ν_j :

$$\tilde{q}_j = q_j + \nu_j, \quad \nu_j \sim \text{Lap}\left(\frac{\Delta f}{\varepsilon_2}\right)$$

The algorithm compares the noisy query $\tilde{q}_j$ to the noisy threshold $\tilde{\eta}$, as illustrated in Figure 2.9:

- **Query 1:** The true count is $q_1 = 950$. The added noise is $\nu_1 = 10$, resulting in $\tilde{q}_1 = 960$. Since $960 < 1015$, the mechanism outputs $\perp$ (Bottom) and waits for the next query.
- **Query 2:** The true count rises to $q_2 = 1010$. The added noise is $\nu_2 = 12$, resulting in $\tilde{q}_2 = 1022$. Since $1022 \geq 1015$, the mechanism outputs $\top$ (Top), signaling that the threshold was successfully crossed.

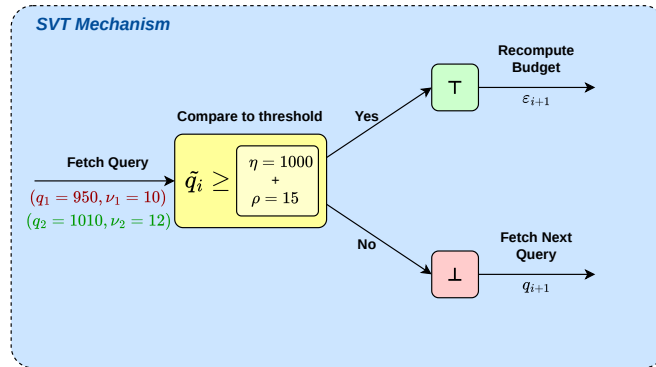


Figure 2.9: SVT Mechanism behavior tracking queries against a threshold of $\eta = 1000$.

Privacy Budget Allocation

The SVT relies on a chunk ε_i of the total budget ε_{SVT} at each epoch i . The available privacy budget ε_i is then split between the threshold perturbation and the query perturbation (i.e., $\varepsilon_1 + \varepsilon_2 = \varepsilon_i$). The fundamental breakthrough of SVT is its ability to answer an infinite number of $\perp$ results, while incurring *no additional privacy cost*. The SVT only pays the budget penalty upon a $\top$ release by recomputing a new chunk ε_{i+1} at epoch $i + 1$.

Dynamic Threshold Tracking

In the context of continual observation on infinite streams, Dong *et al.* [36] leverage SVT to solve the global sensitivity problem. Rather than fixing a poorly estimated upper bound on how many events a single user can contribute (Global Sensitivity), they use SVT to dynamically estimate the maximum user contribution $\kappa(D_t)$.

The SVT continuously monitors the number of users whose contributions exceed the current truncation threshold τ_i . When this count exceeds the noisy SVT threshold $\tilde{\eta}$, the mechanism outputs $\top$. This triggers a threshold doubling ($\tau_{i+1} \leftarrow 2\tau_i$) and initiates a new SVT epoch. This allows the system to scale its truncation boundary adaptively as the stream evolves, maintaining high utility without bleeding budget at every timestamp.

2.7.3 Telescoping Series for Infinite Epochs

Because each threshold doubling triggers a new epoch, assigning a constant privacy budget to each epoch would eventually exhaust the total system budget ε . To support a theoretically unbounded number of epochs on an infinite stream, Dong *et al.* [36] split the total budget exactly in half ($\varepsilon/2$ for the SVT to track user contributions, and $\varepsilon/2$ for the BTM to answer prefix queries) and distribute it across the epochs using a convergent telescoping series.

The budget allocated to the i -th epoch decays according to a rate $\theta > 0$:

$$\varepsilon_i = \frac{(\varepsilon/2) \cdot \theta}{(i + 1)^{1+\theta}}$$

Because the infinite sum of this series converges ($\sum_{i=1}^{\infty} \varepsilon_i \leq \varepsilon/2$), the system can theoretically undergo infinite threshold updates without breaching the global differential privacy guarantee. This satisfies the convergence requirement imposed by basic composition (Section 2.5.1).

However, this mathematical solution introduces a severe practical trade-off that alters the behavior of both mechanisms as the stream evolves.

Impact on the SVT

As the epoch index i grows, the shrinking budget ε_i causes the Laplace noise added to the SVT threshold $\tilde{\eta}$ to increase polynomially, making it harder to fire the SVT in practice. The effect of this will be studied in detail in Section 3.5.

Impact on the BTM

Every time a new epoch is triggered, a brand new BTM forest must be instantiated to recount the stream under the new truncation threshold τ_i . Because this new tree is also bound by the shrinking ε_i budget, the Laplace noise injected into the actual prefix query answers (η_v) grows polynomially with each epoch, degrading query accuracy over time.

2.8 State of the art

Having looked at the formal foundations of differential privacy and its particular challenges in streaming settings (Sections 2.3 and 2.6), we now review the existing research that has tried to address these challenges. Our goal is not to list every paper, but to show how the field has developed: from the first mechanisms for continual observation, through the successive limitations each generation uncovered, up to the mechanism we choose to implement in this thesis. Table 2.9 at the end of this section provides a consolidated comparison.

2.8.1 Continual Observation under Event-Level DP

The first work on continual observation comes from Dwork, Naor, Pitassi, and Rothblum [30]. They formally defined the problem (as described in section 2.6.1), and introduced the tree-based approach that decomposes the time horizon into binary segments of lengths 2^k . They also proved that for monotone functions, the additive error at any single time step is $O(\log^{1.5} T/\varepsilon)$. When considering the utility over all T rounds simultaneously, they showed that the additive error is at most $O(\log^{2.5} T/\varepsilon)$ via a union bound. Their paper also proposed a user-level mechanism, but it required both a bounded global sensitivity GS , which in practice means capping each user’s contribution to GS events, and a strong “unvarying” condition limiting the rate of change of the query value over time. For an infinite stream neither condition can be guaranteed in general, making their user-level result difficult to apply in real-world scenarios.

In an independent work, Chan, Shi, and Song [31] formulated the *Binary Mechanism*. While mathematically similar to the tree-based approach from Dwork et al., their approach was designed specifically for infinite streams by maintaining a dynamically growing forest of trees that stores only $O(\log t)$ active nodes. Their analysis confirmed an additive error of $O(\log^{1.5} t/\varepsilon)$ for the count function, effectively allowing the error to scale with the current time t rather than the fixed horizon T required by the Dwork *et al.* construction. These two foundational works form the basis of what is now commonly referred to as the *Binary Tree Mechanism* (BTM). This is the direct event-level building block used by Dong et al. and it is the version we implement in this thesis.

Following these foundations, research branched into optimizing for specific stream characteristics. For sparse streams, Dwork et al. [55] proposed a private partitioning scheme that avoids spending budget on time steps with no data. Chen et al. [56] introduced PEGASUS, a data-adaptive mechanism that smooths query results during stable periods to reduce effective noise. Again, both works are event-level and do not address the user

contribution problem.

Later work has also focused on improving the constant factors of the BTM rather than how the error scales over time. Denisov et al. [33] used the matrix mechanism with optimal linear operators under (ϵ, δ) -DP, and Fichtenberger *et al.* [57] provided tight fine-grained complexity bounds using completely bounded norms. Henzinger et al. [58] further tightened the error to nearly match the known lower bounds. All three works operate exclusively at event-level and on finite streams; they are nonetheless relevant as potential drop-in replacements for the BTM subroutine in any user-level mechanism.

Expanding the scope to more complex queries, Cardoso and Rogers [59] addressed continual histogram publication by generalizing the model to allow events to consist of an arbitrary subset of items from an unknown domain. While they improved constants by employing Gaussian noise under the $zCDP$ framework, their mechanism remains limited in two ways: first, it provides only event-level privacy. And second, its noise calibration still relies on a fixed time horizon T rather than the current time t . This reinforces the need for mechanisms that can handle both arbitrary user-level contributions and an infinite stream horizon simultaneously.

In summary, while event-level mechanisms have become highly computationally efficient and achieve near-optimal error rates, they inherently fail to protect users who contribute multiple events over the long-term evolution of a stream.

2.8.2 Practical DP Systems

In parallel with the foundational theoretical work of the early 2010s, a separate research thread focused on the engineering challenges of enforcing differential privacy within distributed compute frameworks. This systems-oriented lineage is directly relevant to this thesis, as it provides the historical precedent for integrating privacy primitives into execution engines like Apache Flink.

The first significant milestone was PINQ (Privacy Integrated Queries) [60], which demonstrated how DP could be enforced automatically for non-expert analysts. By embedding DP within a LINQ-like API, PINQ established a blueprint for privacy-aware query languages where budget tracking and noise addition are handled transparently by the system rather than the user. Shortly thereafter, Airavat [61] extended this vision to the MapReduce and Hadoop ecosystem. Airavat is particularly relevant to this work as it provided the first existence proof that DP could be enforced transparently in a distributed, parallel data processing environment, a direct conceptual precursor to modern stream processing engines. It combined mandatory access control (MAC) with modifications to the Java Virtual Machine (JVM) to ensure that untrusted mapper code could not leak information through side channels, while trusted reducers added calibrated noise to guarantee differential privacy on the output.

The most significant recent milestone in this systems thread is DP-SQLP (Differential Privacy SQL Pipelines) [32], a production-scale streaming DP system developed by Google. DP-SQLP is built on a micro-batch streaming framework and addresses five concrete

engineering challenges that any real-world deployment must solve: an unknown key space where keys are discovered only as data arrives, synchronous execution driven by the data stream, scalability to billions of distinct keys, user contribution bounding to achieve user-level privacy, and correlated noise addition to manage the privacy cost of repeated releases. Its deployment to Google Shopping and Google Trends demonstrates that differentially private stream processing is industrially viable. Crucially, the paper explicitly notes that none of the major open-source streaming frameworks, including Apache Flink, currently offer built-in differentially private queries [32], which directly motivates the implementation undertaken in this thesis.

Despite these contributions across fifteen years, these systems share a fundamental limitation. They all rely on a fixed, a priori contribution bound: PINQ and Airavat require the analyst to pre-declare a sensitivity limit before execution, while DP-SQLP requires setting a maximum contribution parameter C in advance, typically estimated from a data sample or proxy dataset. If this bound is set too small, heavy users have their excess contributions silently clipped, introducing systematic bias into the output. If it is set too large, the noise added scales proportionally with C , destroying utility through excessive variance. None of these systems can dynamically adapt the bound to the actual hardness (2.6.3) of the data instance being processed. Addressing this limitation, moving from fixed sensitivity declarations to automated, instance-adaptive user-level estimation, is the core theoretical goal of the mechanisms described in Sections 2.8.4 through 2.8.6, and the primary objective of the Flink implementation evaluated in Chapter 3.

2.8.3 Weaker Privacy Relaxations for Streams

Faced with the difficulty of user-level DP, several lines of work adopted weaker privacy models that trade protection strength for utility.

Kellaris et al. [62] introduced **w-event DP** Appendix A.3.3, which protects the privacy of any w consecutive events in the stream rather than the entire history. By allocating the budget only over a sliding window of size w , this model reduces the noise by a factor proportional to T/w compared to full continual observation, yielding error $O(w \log^{1.5} t/\epsilon)$. Several follow-up works [63] [64] extended this model to spatio-temporal and local settings. The fundamental limitation of w -event DP is that an adversary who observes events outside any particular window of size w is entirely unprotected. This model is unsuitable for the threat model of this thesis, which requires User-Level DP to protect a user’s entire participation history against an analyst who observes the full, long-term output sequence.

In parallel to these developments, Joseph et al. [65] and Bao et al. [66] studied event-level DP under the *local* model, where each data point is perturbed by the user before being sent to the curator. Local DP provides stronger trust assumptions but inherits a quadratic error penalty relative to the central model. In order to achieve the same utility as the BTM, it would require $O(n)$ times more users. Bounded per-user contributions are still assumed in both works. For our use case, a central curator running Apache Flink who already has access to the raw stream, the local model adds significant noise overhead without providing a meaningful increase in practical privacy.

2.8.4 User-Level DP on Static Datasets

Progress on user-level DP for streaming was preceded by a significant breakthrough in the static setting that reframed how utility should be measured.

Earlier approaches to user-level DP in the static setting tried to optimise worst-case error, which is necessarily $\Omega(\kappa(D))$ where D represents the dataset and $\kappa(D)$ is the maximum user contribution in the dataset. Without an a priori bound on $\kappa(D)$, worst-case analysis yields results that aren't very meaningful in practice.

Huang, Liang, and Yi [67] and Dong and Yi [68] independently proposed instance-specific mechanisms: instead of bounding the worst-case error, they proved that the error on any given instance D is $\tilde{O}(\kappa(D))$ without needing to know $\kappa(D)$ in advance. They also proved this is down-neighbourhood optimal, no ε -DP mechanism can do better on all instances simultaneously.

Fang, Dong, and Yi [69] then generalised this result to all monotonic functions using the Shifted Inverse mechanism, effectively closing the static user-level DP problem. The natural open question that remained was: how much of this instance-specific approach can be carried over to a streaming, infinite-horizon setting?

2.8.5 Toward User-Level DP for Infinite Streams

The transition toward user-level protection in streaming was characterized by attempts to manage the total privacy budget over time. Fan and Xiong [29] proposed the **FAST** (Filtering and Adaptive Sampling Trend) framework.

Unlike the event-level mechanisms discussed in Section 2.8.1, FAST protects the user's entire history by allocating a fixed privacy budget ϵ across the stream. To maintain utility over long horizons, it uses a Kalman filter to estimate the state of the stream and adaptively samples data only when the detected dynamics change significantly. However, the mechanism relies on a hard constraint: it assumes a maximum number of allowed samples M , and it assumes a fixed global sensitivity for each timestamp (i.e., each user contributes at most once per sampled time step). Once this budget is exhausted, the system ceases to collect new data and relies entirely on prediction to release future values. While influential, this approach creates a fundamental trade-off: for infinite streams, the prediction error eventually accumulates once the sampling budget is spent, making it difficult to support a truly infinite horizon without a-priori knowledge of the stream's length or characteristics.

The theoretical difficulty of user-level privacy identified in Section 2.6.2 leads to a precise gap in the literature prior to 2023. No mechanism existed for continual observation under user-level DP that satisfied simultaneously:

1. requiring no a priori bound on user contributions
2. providing a theoretically guaranteed utility that degrades gracefully with the actual data hardness

3. supported an infinite stream where noise scales with the current time t .

2.8.6 Closing the Gap: Dong, Luo, and Yi (2023)

Dong, Luo, and Yi [36] resolve all three requirements by applying the instance-specific approach of [67] [68] from the static to the dynamic setting. Their mechanism for the continual count query achieves error $\tilde{O}(\kappa(D_t)/\varepsilon)$ at every time t . Here, D_t denotes the prefix of the stream up to time t , and the $\tilde{O}$ notation hides polylogarithmic factors in t and $1/\delta$. $\kappa(D_t)$ grows with the stream and is never assumed to be known.

The key technical contribution is a private online estimator for $\kappa(D_t)$: using the Sparse Vector Technique (SVT) [70], the mechanism privately detects when the number of users exceeding the current truncation threshold τ crosses a discounted bound, then doubles τ and launches a new SVT epoch. To solve the problem identified in section 2.5.1, Dong *et al.* utilize a specific telescoping series $\varepsilon_i = \varepsilon\theta/(i+1)^{1+\theta}$. This ensures that the total privacy consumption stays bounded over infinitely many epochs. Once $\tilde{\tau}_t \approx \kappa(D_t)$ is established, the stream is truncated to $\tilde{\tau}_t$ items per user, and the event-level BTM of Chan et al. [31] is applied with budget $\varepsilon/\tilde{\tau}_t$ via group privacy.

For count and sum, the mechanism is proved to be $(1, \tilde{O}(1))$ -down-neighbourhood optimal, matching the static case up to polylogarithmic factors. A further contribution is a separation result: for distinct count, histogram, and k -selection, down-neighbourhood optimality is impossible in the dynamic setting, even though it is achievable statically. This is the first formal separation between the static and streaming cases under differential privacy.

The transition toward these instance-specific methods represents the most significant recent shift in the field. This paradigm was finally brought to infinite streams by Dong *et al.* (2023), effectively closing the theoretical gap for user-level streaming. However, the practical implementation of such a system remains highly complex. This thesis builds directly on their work by implementing this SVT-based estimator within an Apache Flink environment. A major goal of this work is to evaluate the practical robustness of the mechanism when dealing with the contribution skew and user cardinality issues encountered during the implementation, which are evaluated in Chapter 3.

2.8.7 Summary

To summarize the research discussed so far, Table 2.9 provides a consolidated comparison of the discussed mechanisms based on the requirements of a real-world streaming system. This comparison reveals a clear progression in the field, which can be categorized into three distinct eras:

1. **The Foundational Era (2010-2015):** Research focused on establishing the *Binary Tree Mechanism* to handle the composition challenges of continual observation. While these methods are now highly efficient, they operate almost exclusively at the event-level. As discussed, this provides insufficient protection for users contributing multiple events over a long-term stream.

2. **The Relaxation Era (2015-2022):** To address the utility-privacy trade-off, several works introduced weaker models like w -event DP or Local DP. These allowed for lower noise but required significant sacrifices in the strength of the privacy guarantee or assumed a fixed time horizon T .
3. **The Instance-Adaptive Era (2023-Present):** The most recent shift involves moving away from worst-case sensitivity bounds toward instance-specific methods. These mechanisms calibrate noise based on the actual data hardness (κ) without requiring *a priori* limits on user behavior.

As highlighted in Table 2.9, the mechanism by Dong *et al.* [36] is the only one to satisfy the requirements for infinite-horizon, user-level streaming without requiring pre-declared contribution bounds. Consequently, it serves as the theoretical baseline for the Apache Flink implementation and empirical evaluation presented in the remainder of this thesis.

Table 2.9: Comparison of differential privacy mechanisms for streaming data.

Paper	Year	Model	Stream	Bound	Inst.	Functions	Additive Error	Core Technique	Key Limitations (Streaming)
Dwork <i>et al.</i> [30]	2010	E, U	Fin.	✓	✗	Count, monotone	$O\left(\frac{\log^{1.5} T}{\epsilon}\right)$	Binary Tree Mechanism	Finite horizon (requires fixed T); requires hard cap; user-level bound no better than group privacy.
Chan <i>et al.</i> [31]	2011	E	Inf.	N/A	✗	Count, Sum	$O\left(\frac{\log^{1.5} t}{\epsilon}\right)$	Dynamic BTM forest;	Event-level only; user-level via group privacy multiplies noise by $\kappa(D_t)$.
Kellaris <i>et al.</i> [62]	2014	w-E	Inf.	N/A	✗	Sequences, Aggregates	$O\left(\frac{w \log^{1.5} t}{\epsilon}\right)$	Sliding-window budget allocation	Protects only w events; no user-level guarantee; w must be a priori.
Fan & Xiong (FAST)[29]	2013	U	Inf.	✓	✗	Count, Sum monitoring	No formal bound	Adaptive noise + Kalman filter	Limited by sampling budget M ; system goes “blind” after exhaustion; assumes fixed sensitivity.
Dwork <i>et al.</i> [55]	2015	E	Fin.	N/A	✗	Rectangle queries	$O\left(\frac{\text{polylog } T}{\epsilon}\right)$	Private stream partitioning	Event-level only; finite stream; no infinite-stream support.
Chen <i>et al.</i> [56]	2017	E	Inf.	N/A	~	Count, Sum, Histogram	Adapts to density	Data-adaptive smoothing	Event-level only; ignores user contribution skew.
Joseph <i>et al.</i> [65]	2018	LDP	Inf.	~	✗	Count (evolving)	$\Omega(\sqrt{n})$ penalty	Local randomisation	Local model: high error penalty vs. central DP.
Bao <i>et al.</i> [66]	2021	LDP	Inf.	✓	✗	Count, Histogram	$O\left(\frac{\sqrt{T}}{\epsilon}\right)$ under LDP	Enhanced local + correlation	Local DP noise floor; hard cap required; comms overhead.
Denisov <i>et al.</i> [33]	2022	E (δ)	Fin.	N/A	✗	Count, Sum	Improved BTM constants	Matrix mechanism	Relaxes to (ϵ, δ) -DP; finite stream; event-level only.
Fichtenberger [57]	2022	E (δ)	Inf.	N/A	✗	Count	$O\left(\frac{\log^{1.5} T}{\epsilon}\right)$	Operator-theoretic BTM	Highly theoretical; no streaming implementation considerations.
Cardoso & Rogers [59]	2022	E	Inf.	N/A	✗	Histogram, Heavy Hitters	$O\left(\frac{\text{polylog } T}{\epsilon}\right)$ per bin	Streaming selection into unknown	Event-level only; fixed horizon T required for noise scaling.
Huang <i>et al.</i> [67]	2021	U	Static	✗	✓	Count, Mean	$\tilde{O}(\kappa(D)/\epsilon)$	Instance-optimal estimator	No temporal composition; extending to streaming was open problem.
Dong & Yi [68]	2021	U	Static	✗	✓	General queries	$\tilde{O}(\kappa(D)/\epsilon)$	Universal estimators	Static only; no dynamic extension discussed.
Zhang <i>et al.</i> [32]	2024	U (δ)	Inf.	✓	✗	Count, Sum, Histogram	$O\left(\frac{C \log^{1.5} t}{\epsilon}\right)$	BTM + Key Selection	Fixed contribution bound C required; bias-variance trade-off; no instance-optimal guarantee.
Dong, Luo & Yi [36]	2023	U	Inf.	✗	✓	Count, Sum, Hist, k-sel, Distinct	$\tilde{O}(\kappa(D_t)/\epsilon)$	SVT κ-est + Dynamic Truncation	Practical limitations identified in this thesis (see Chapter 3).

Model: E: Event-level; U: User-level; w-E: w-Event; LDP: Local DP.

Stream: Fin: Finite horizon (fixed T); Inf: Infinite horizon (scales with t).

Bound: ✓: Requires an *a priori* fixed contribution bound (hard cap); ✗: No prior bound required.

Inst.: ✓: Instance-adaptive (error scales with actual data hardness κ); ✗: Worst-case noise.

† **Highlighted row** represents the baseline for this thesis.

Chapter 3

Testing the Limits

3.1 Overview

As discussed in Chapter 2, applying differential privacy to infinite data streams presents unique challenges. The recent work by Dong *et al.* [36] represents a fundamental theoretical breakthrough for user-level DP in this setting. By replacing the static global sensitivity Δ_f with a privately and dynamically estimated threshold via the Sparse Vector Technique (SVT) (Section 2.7.2), their mechanism provides rigorous asymptotic utility bounds with an error of $\tilde{O}(\kappa(D_t)/\varepsilon)$. This elegantly eliminates the need for an *a priori* bound on user contributions.

However, asymptotic theoretical optimality does not guarantee practical robustness. The $\tilde{O}$ notation suppresses polylogarithmic factors that can manifest as severe utility bottlenecks in distributed systems. Crucially, the mathematical guarantees of the Dong *et al.* algorithm rely on the assumption that the SVT mechanism will continually and accurately detect when to double the user contribution truncation threshold τ .

The primary motivation of this chapter is to answer the critical following question: *Under what real-world data distributions does this state-of-the-art algorithm succeed, and under what mathematical conditions is it guaranteed to fail?*

To answer this, we translated the algorithm’s abstract pseudo-code into a fully distributed, stateful, and fault-tolerant pipeline (concepts introduced in Section 2.1), and studied the breaking points of the algorithms. In doing so, this chapter makes the following core contributions for Differentially Private stream processing:

- **Production-ready Apache Flink module.** We provide an implementation of the Dong *et al.* algorithm integrated with Apache Kafka. By encapsulating theoretical DP components such as the dynamic Binary Tree Mechanism (BTM) 2.7.1 and the SVT 2.7.2 into native Flink stateful operators, we deliver an intuitive, plug-and-play module. The source code is publicly available at <https://github.com/loukini/continual-observation-userdp-flink>.
- **Comprehensive dataset profiling.** We curate and analyze a diverse corpus of real-world datasets representing a wide spectrum of data distributions. We profile

these streams across multiple dimensions, focusing on the variables that predict the algorithm’s success or failure, such as active user cardinality $|U_t|$ and maximum user contribution $\kappa(D_t)$.

- **Identification of algorithmic limitations.** We empirically isolate and formally characterize the boundary conditions under which the algorithm fails. We demonstrate how user participation distribution skew can starve the SVT, inducing a pseudo-deterministic¹ halting point to the threshold-doubling mechanism. We formally define this failure state as **Truncation Lock**.
- **A priori relative error estimation.** We derive a predictive mathematical model to estimate the relative error of a dataset over time based purely on its raw frequency counts. This provides analysts with a lightweight tool to forecast utility degradation and algorithmic failure without the computational overhead of deploying and running the entire DP pipeline.

3.2 System Architecture and Orchestration

While implementing the continual count mechanism of Dong *et al.* [36], what we had to keep in mind is that the theoretical mechanism is defined as a sequence of logical steps. An Apache Flink pipeline deployment, however, must realize these steps under event-time semantics, managed state, and failure recovery as detailed in 2.1.

We achieved this using a dataset preparation and replay layer, a Kafka ingestion layer, a Flink event-time processing layer, and a localized differential privacy execution layer centered around a single stateful coordinator.

Rather than distributing the differential privacy logic across several communicating operators, the full continual-count state machine is encapsulated inside one `KeyedProcessFunction` (2.1.5), which simplifies synchronization and maintains a single coherent view of the evolving threshold τ .

3.2.1 Data Ingestion and Event-Time Semantics

The Apache pipeline is decoupled into a Kafka replay stage, and a Flink stream-processing job. The framework accepts a prepared dataset from disk, derives a dataset identifier from the input path, and maps that dataset to a Kafka topic name.

The orchestration entry point is `DPCountJob`. Its role is to configure and launch the Flink continual counting job: it reads the Kafka connection parameters, constructs the `KafkaSource` consuming from the prepared topic, assembles the processing pipeline, and submits the job for both exact and differentially private runs.

¹By isolating the truncation bias from the Laplace noise, we can model this halting point using fixed mathematical bounds.

Each input record is read from the prepared dataset format and converted into a JSON Kafka message represented as illustrated in Listing 1:

Listing 1 StreamingEvent model

```

1 public class StreamingEvent implements java.io.Serializable {
2
3     private static final long serialVersionUID = 1L;
4     public String userId;
5     public String itemId;
6     public long timestamp;
7     public String payload;
8
9     // rest of the code ...
10 }

```

On the Flink side, the counting job consumes these messages through a `KafkaSource` configured to start from the latest available offsets.

Inside the Flink pipeline, each incoming JSON record is parsed into the `StreamingEvent` representation. The DP operator therefore receives a uniform event model regardless of which prepared dataset was replayed into Kafka.

A critical streaming concept used here is the distinction between *processing time* and *event time*, explained in Section 2.1.2. The continual-observation mechanism follows the chronological stream prefix, therefore correctness depends on processing records according to the timestamps carried by the events rather than according to their physical arrival order at the machine. To enforce this, the job assigns timestamps and watermarks using `WatermarkStrategy.forBoundedOutOfOrderness(...)`, allowing slightly delayed records to be reordered before they are processed by the stateful DP operator.

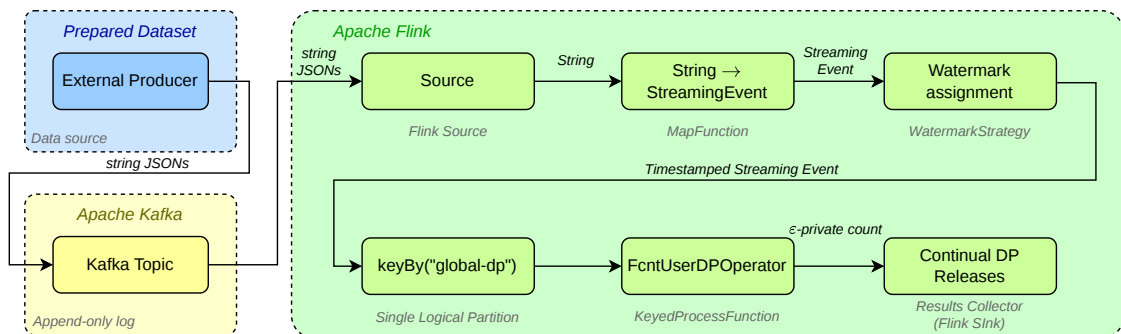


Figure 3.1: End-to-end orchestration of the continual DP count pipeline

3.2.2 The Unified DP Coordinator

To compute a global continual count query, all events are routed to a single logical partition. This is achieved in the DP pipeline via Flink’s `keyBy` operation, which channels the entire stream into one instance of a `KeyedProcessFunction`, as explained previously in Section 2.1.5. In our implementation, this function is the `FcntUserDPOperator`.

This design is deliberate, `FcntUserDPOperator` acts as a monolithic and stateful coordinator. This allows it to maintain a single coherent state machine for dynamic truncation, SVT-based threshold updates, and noisy continual counting.

Internally, the coordinator stores the global stream clock, the latest noisy release, the cumulative effective count, the cumulative truncated count, and the current hardness statistic $\kappa(D_t)$ (See 2.6.3) as Flink-managed `ValueState`. During initialization, it instantiates three sub-engines: `UserTrackingOperator`, `SVTUserContributionOperator`, and `EventDPBinaryTreeOperator`.

For each incoming event, the coordinator:

1. Updates the logical time step
2. Delegates user tracking
3. Evaluates whether the Sparse Vector Technique should trigger
4. Releases any newly admissible add-back events, if the SVT is triggered
5. Advances the noisy binary tree
6. Emits a `DPrelease` object containing the current privatized count and diagnostic metadata

A detailed illustration of the orchestration is detailed in Figure 3.2.

Sub-Engine 1 Dynamic Truncation (`UserTrackingOperator`)

The first internal component is `UserTrackingOperator`, whose role is to enforce the current contribution bound τ for each user. To do so, it maintains, for each user, both the total number of events observed so far and the number of events currently withheld by truncation, implemented through two `MapState<String, Long>` objects.

When a new event arrives, the operator first increments the total number of events associated with that user for the current rating. If the updated count is at most τ , the event is accepted immediately and contributes to the downstream continual counter. If the updated count exceeds τ , the event is marked as truncated, added to the user’s truncation buffer, and, when this is the first time the user crosses the threshold, the global count of users above τ is incremented.

A crucial feature of this operator is the add-back mechanism. As shown in the Figure 3.3, if the SVT later doubles the threshold to $\tau_{i+1} = 2\tau_i$, some events that were previously

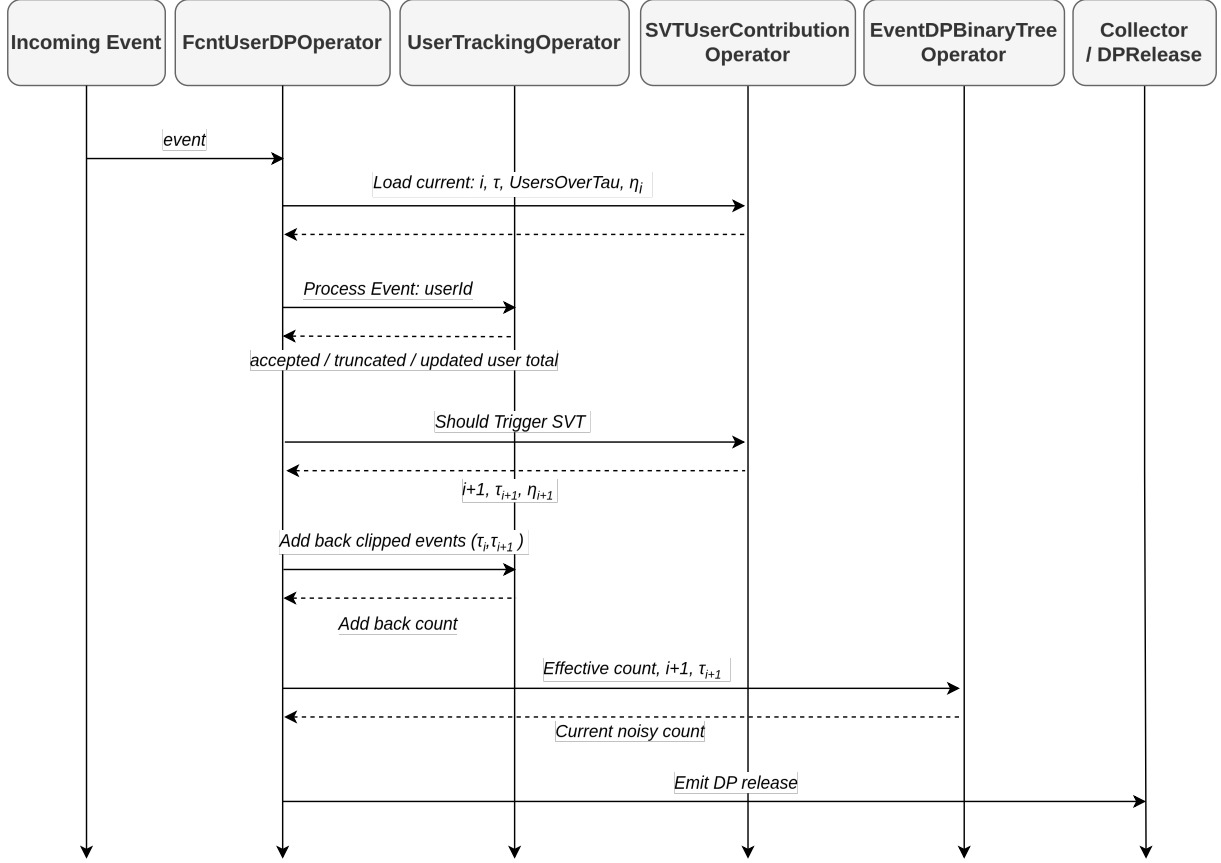


Figure 3.2: Per-event execution flow inside the pipeline

buffered become admissible again under the new contribution cap. Instead of replaying the stream from the beginning, the operator scans the truncation buffers and releases up to the newly available headroom for each user. This implementation strategy is described by Dong *et al.*: once the bound increases, earlier truncated contributions are incrementally restored without rewinding the historical stream.

Sub-Engine 2 Hardness Tracking (SVTUserContributionOperator)

The second internal component, **SVTUserContributionOperator**, is responsible for deciding when the current threshold τ_i should increase. It implements the Sparse Vector Technique used in the Dong *et al.* mechanism to privately track whether the observed data hardness warrants a doubling of the truncation level. Four **ValueState** quantites are tracked: the current SVT index i , the current threshold τ_i , the number of users currently exceeding τ_i , and the noisy stopping threshold $\tilde{\eta}_i$, variables explained in the Section 2.7.2.

At each event, the coordinator passes the updated number of users above τ to the SVT engine. If this count exceeds the SVT limit it fires, and then: the threshold doubles, the SVT epoch increments, and a fresh noisy threshold is generated for the next stage.

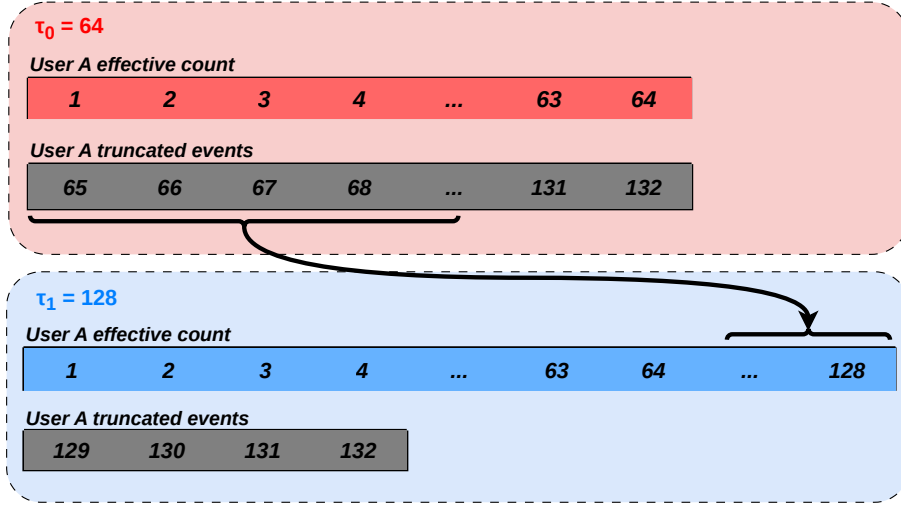


Figure 3.3: Add-back mechanism implemented by `UserTrackingOperator`

Sub-Engine 3 Noisy Prefix Summation (`EventDPBinaryTreeOperator`)

Once the effective count allowed by the truncation threshold at the current time step is determined, it is forwarded to `EventDPBinaryTreeOperator`. This component implements the continual user-level DP counting primitive based on the Binary Tree Mechanism explained in Section 2.7.1.

The implementation is intentionally sparse. Instead of materializing a full binary tree, it stores only the active exact nodes in `MapState<Integer, Long>` and their noisy counterparts in `MapState<Integer, Double>`. The current logical tree time, the active period depth, and the base accumulator on the other hand, are maintained as `ValueState`.

For each update, the operator determines which tree level must be modified by using bitwise arithmetic on the current time index. The operator then forms the exact partial sum, computes the node-level privacy budget, injects Laplace noise through a shared `LaplaceMechanism`, stores the new active node, and removes the lower-level nodes that have been absorbed into the merge, as seen in Figure 2.8.

The final noisy prefix estimate is reconstructed from the active nodes selected by the binary representation of the current logical time step. Because only the active frontier of the tree is retained, the state footprint grows only as $O(\log t)$. This memory-efficient design is what makes indefinite continual release feasible in a long-running Flink operator.

3.2.3 Fault Tolerance and Privacy Budget Integrity

In a streaming differential privacy system, fault tolerance is part of the privacy guarantee itself. If a task fails, the stream is replayed from Kafka, and the operator reprocesses the same logical events while drawing fresh noise, then the same query would silently consume privacy budget multiple times. This would invalidate the theoretical privacy accounting

assumed by the mechanism, discussed in Section 2.1.4.

For this reason, all privacy-relevant variables in our implementation are stored as Flink managed state. This includes the current threshold τ_i , the SVT epoch i , the noisy SVT threshold η_i , the binary-tree nodes, the global logical clock, the per-user counters, the truncation buffers, the latest noisy output, and budgeting parameters such as ε_i and β_i . A restart restores each keyed operator instance to exactly the same differential privacy state it held before failure.

This property is essential. Previously generated noisy values are recovered rather than regenerated, which prevents privacy budget leakage under replay and ensures that the implemented system remains faithful to the assumptions of the theoretical mechanism throughout the life of the stream.

3.3 Dataset Selection

To properly benchmark the algorithm across diverse scenarios, we define the following key metrics used to profile each stream at any given time step t :

- **Active User Cardinality** ($|U_t|$): The cumulative number of unique users observed in the stream up to time t .
- **Maximum User Contribution** ($\kappa(D_t)$): The maximum number of events contributed by any single user up to time t . This represents the heaviest hitter in the stream.
- **User Contribution Skewness**: The distribution of events among users. Highly skewed datasets are characterized by a small fraction of users contributing the vast majority of events.
- **Relative Error**: The absolute difference between the true query evaluation and the differentially private output, divided by the true value, serving as our primary utility metric $\frac{|c_t - \tilde{c}_t|}{c_t}$, with c_t the exact count and $\tilde{c}_t$ the Differentially Private count.

To evaluate the practical limits of the mechanism, we gathered an extensive corpus of publicly available datasets that represent real-world streaming scenarios. These datasets exhibit varying temporal lengths, cardinalities, and user contributions skewness. We passed these static logs into our Apache Kafka/Flink pipeline applying the Differentially Private algorithm, as explained in Section 3.2 and monitored the relative error of the output.

To provide context on the nature of the streaming events, our chosen datasets span several distinct domains:

- **E-commerce and Reviews**: *Amazon Fashion*, *Amazon Movies*, *Epinions*, *RetailRocket*, *Steam*, and *Yelp* capture user product interactions, clicks, and textual reviews.

- **Media and Entertainment:** *Netflix*, *MovieLens 1M*, *MovieTweatings*, *LastFM*, and *Goodreads* record user ratings, viewing history, and music listening events (scrobbles).
- **Location-based Social Networks:** *Foursquare (Global, NYC, TKY)*, *Brightkite*, and *Gowalla* log sequential geographical check-ins made by mobile users.
- **Search and Short Video:** *AOL* contains historical search engine queries, while *KuaiRec Big* tracks continuous user interactions with a short-video recommendation app.

The dataset metrics are summarized in Table 3.1. This diversity allowed us to benchmark the algorithm and get a primary intuition on which data splits yield good outputs, and which ones output unusable results with relative errors sometimes approaching 100%. We used the following privacy parameters for all our tests: $\varepsilon = 1$, $\theta = 1$ and $\beta = 0.1$

Dataset	Total Events	Cardinality $ U $	Hardness $\kappa(D)$	Avg/User	Avg RE%
Amazon Fashion [71]	883,636	749,233	40	1.18	3
Amazon Movies [71]	8,765,568	3,826,085	4,254	2.29	4
AOL [72]	19,442,629	521,693	150,936	37.27	3
Brightkite [73]	4,747,281	51,406	2,100	92.35	27
Epinions [74]	922,267	22,164	5,357	41.61	14
Foursquare Global [75]	33,263,631	266,909	5,430	124.63	3
Foursquare NYC [75]	227,428	1,083	2,697	210.00	56
Foursquare TKY [75]	573,703	2,293	2,991	250.20	37
Goodreads [76] [77]	15,739,967	465,323	21,811	33.83	10
Gowalla [73]	6,442,892	107,092	2,175	60.16	16
KuaiRec Big [78]	12,530,806	7,176	16,015	1,746.21	24
LastFM [79]	19,098,862	992	183,103	19,252.89	99
MovieLens 1M [80]	1,000,209	6,040	2,314	165.60	43
MovieTweatings [81]	921,398	71,707	2,875	12.85	15
Netflix [82]	100,480,507	480,189	17,653	209.25	4
RetailRocket [83]	2,756,101	1,407,580	7,757	1.96	6
Steam [84]	7,793,069	2,567,520	2,045	3.04	2
Yelp [85]	6,990,280	1,987,929	3,048	3.52	5

Table 3.1: Statistical profile and baseline performance of the benchmarking corpus. The metrics illustrate the relationship between dataset cardinality, user contribution skew, and the resulting average relative error (Avg RE). Highlighted rows indicate datasets with high skew and low cardinality that encounter structural utility limits.

3.4 Intuition: The Impact of User Cardinality and Distribution

To understand the algorithm’s vulnerability, we must recall its truncation mechanism (Section 3.2.2). To avoid spending privacy budget ε on continuous threshold evaluations, the algorithm employs a Sparse Vector Technique (SVT) as an internal monitor. The SVT

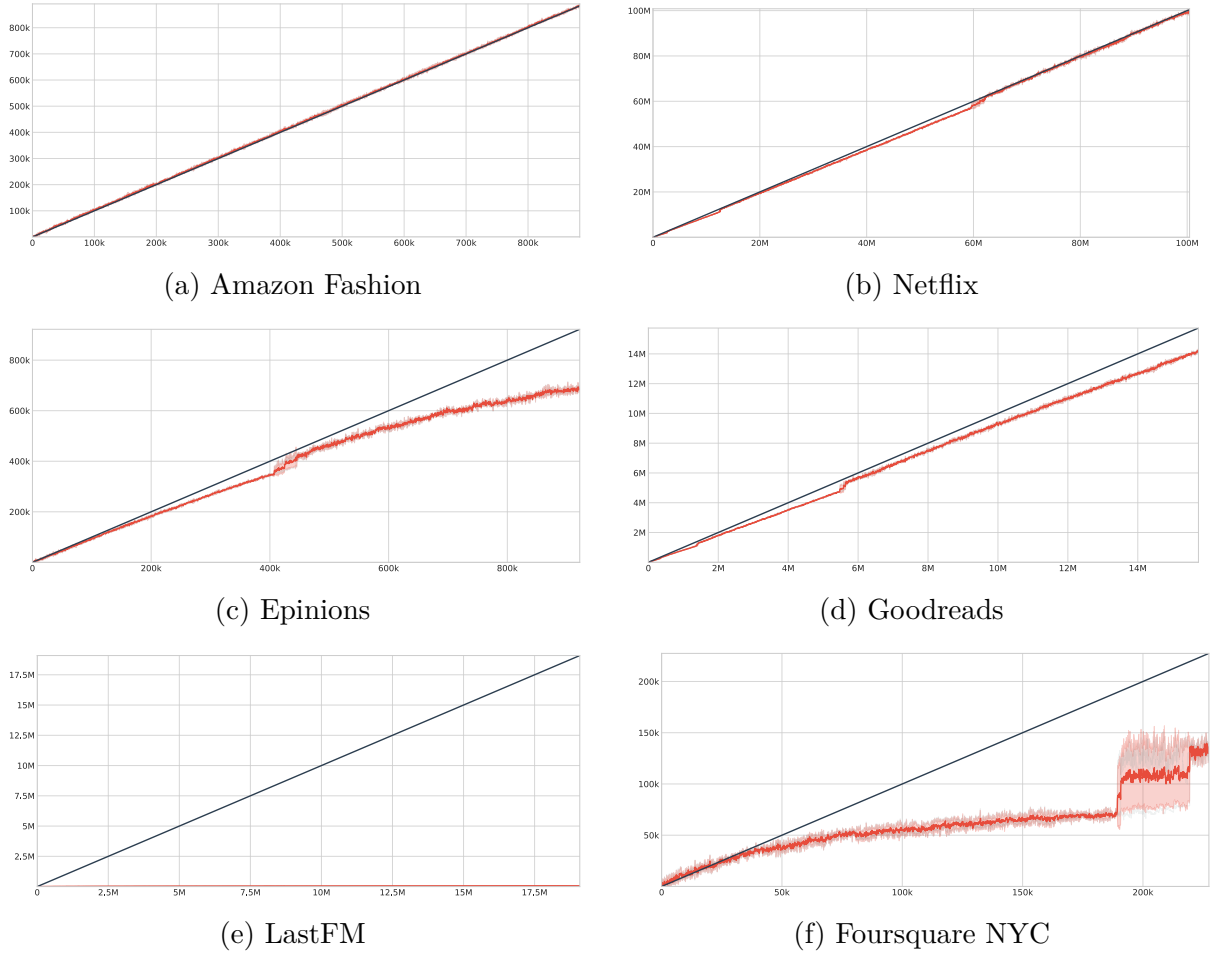


Figure 3.4: Real count (black) vs DP count (red) across time with $\varepsilon = 1$, $\theta = 1$ and $\beta = 0.1$

continually tracks the number of “voters” which are users whose event count exceeds the current threshold τ . When this voter pool surpasses a noisy algorithmic requirement, τ doubles and the mechanism enters a new epoch.

As showcased by the dataset profiles in Table 3.1 and selected benchmark plots (with the comprehensive suite of dataset figures available in Appendix A.4, A.6), our tests reveal a fundamental limitation in this SVT voting system. While it scales effectively in balanced datasets, it fails pseudo-deterministically under conditions of low user cardinality combined with high contribution skew.

Optimal conditions: High cardinality streams. Balanced streams with massive user populations, such as the Netflix, Amazon Movies, and Yelp datasets, represent the ideal environment for the Dong *et al.* algorithm. These datasets possess an exceptionally high active user cardinality $|U_t|$ relative to their maximum user contribution $\kappa(D_t)$. Because the stream possesses a vast user base, the number of distinct users crossing the current threshold τ easily overcomes the SVT’s noisy threshold requirements. Consequently, the SVT triggers reliably, allowing τ to dynamically scale alongside $\kappa(D_t)$. This successful scal-

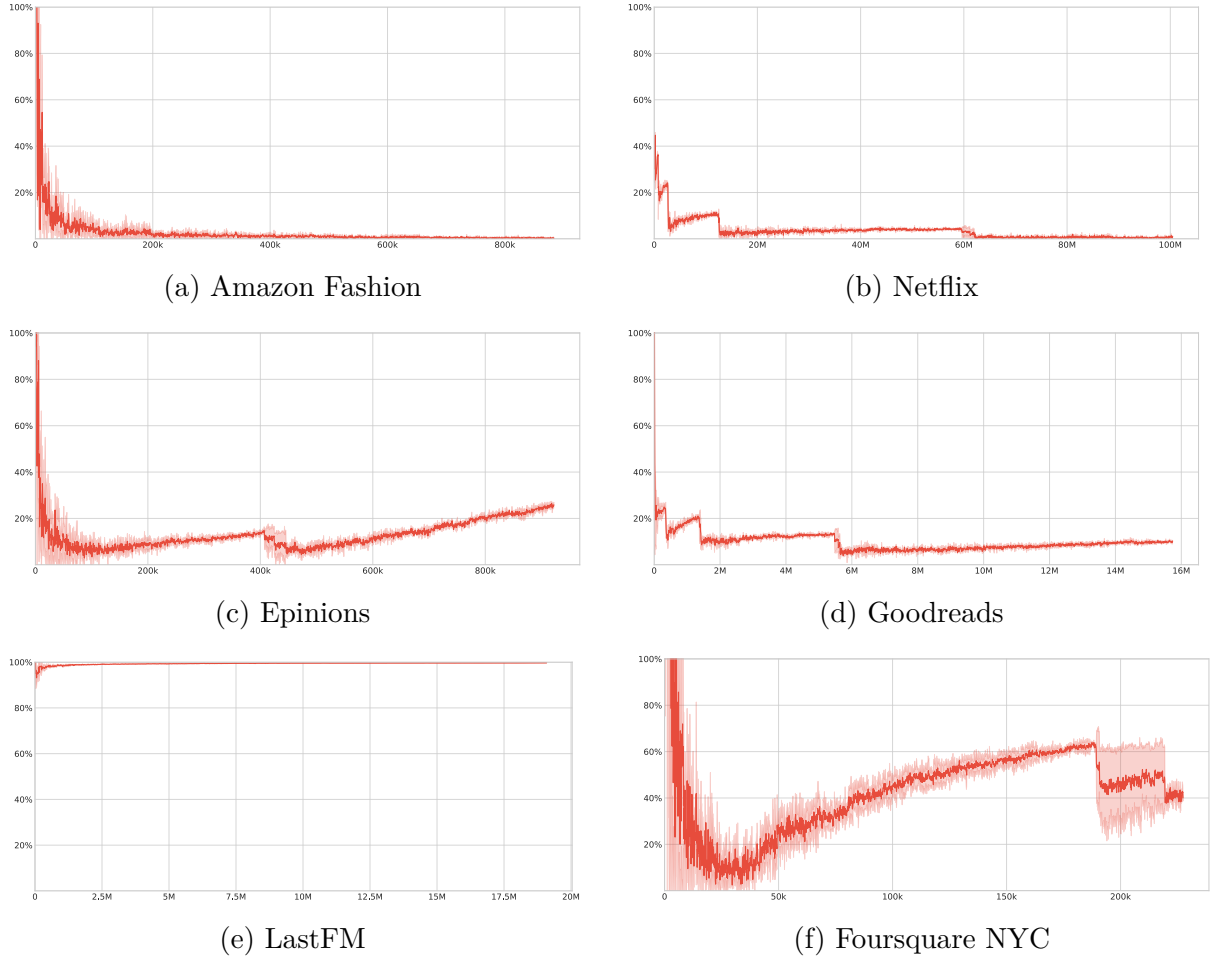


Figure 3.5: Relative error across time with $\varepsilon = 1$, $\theta = 1$ and $\beta = 0.1$

ing minimizes truncation bias, allowing the relative error to converge toward zero over time.

Adverse conditions: High skew and low cardinality. Conversely, datasets characterized by low cardinality and extreme skew, such as LastFM and KuaiRec, expose the algorithm’s practical blind spot. These streams are driven by a few dominant heavy hitters generating extreme maximum contributions $\kappa(D_t)$, paired with a comparatively small unique user base $|U_t|$. In these scenarios, the theoretical maximum number of voters is strictly bounded by the total population $|U_t|$. When this global population is smaller than the mathematical threshold required to overcome the SVT’s injected noise, the update mechanism starves.

Truncation Lock. Because the skewed stream fails to generate sufficient voters, the SVT is never triggered. The truncation threshold τ remains permanently frozen at a fraction of $\kappa(D_t)$. We formally define this state of permanent algorithmic stalling as **Truncation Lock**.

Under Truncation Lock, the mechanism systematically discards the vast majority of events generated by heavy hitters. For instance, in the LastFM dataset, a user with 183,103

valid events is severely truncated to just 64. While balanced streams dynamically manage their error, streams trapped in Truncation Lock suffer from a relative error approaching 100%, entirely destroying query utility.

Failure point. To illustrate this stalling, suppose the current global threshold is $\tau_0 = 64$ and the SVT requires exactly 1,000 distinct voters to double the threshold to $\tau_1 = 128$. Consider a state where 999 users have already crossed τ_0 . The system is waiting for one final voter. We observe two specific users:

- **User A** has participated 300 times, but due to the truncation mechanism, only $\tau_0 = 64$ of these events are currently counted.
- **User B** has participated 60 times so far.

Table 3.2 illustrates the intended, functional behavior. As soon as User B contributes a few more events and reaches the threshold, the SVT triggers. Truncated events from the initial 999 users are immediately added back into the stream and counted in the output, up to the new threshold $\tau_1 = 128$. User A suddenly has 128 of their 300 contributions counted.

t	User	Threshold	Users over τ	Contribution	Effective count
$t_{10,000}$	A	$\tau_0 = 64$	999	300	64
$t_{10,001}$	B	$\tau_0 = 64$	999	60	60
$\vdots$	B	$\tau_0 = 64$	999	$\vdots$	$\vdots$
t_k	B	$\tau_0 = 64$	1,000	65	64
t_{k+1}	A	$\tau_1 = 128$	105	301	128
t_{k+2}	B	$\tau_1 = 128$	105	66	65

Table 3.2: Illustration of a successful threshold doubling from $\tau_0 = 64$ to $\tau_1 = 128$.

Table 3.3 illustrates the Truncation Lock failure mode. If User B stops participating at 61 events, and no other new user ever reaches the threshold $\tau_0 = 64$ events, the $1k$ -voter requirement is never met. In the worst-case scenario, the algorithm perpetually clips the contributions of the highly active users. As User A continues to generate up to $180k$ events, the system continues to count only 64.

This perfectly mirrors the issue observed in the LastFM stream: an exceptionally high user participation average and severe skew, but a strictly limited absolute cardinality ($|U| = 992$). The truncation threshold is forever locked at 64, while individual heavy hitters contribute hundreds of thousands of uncounted events.

3.5 Mathematical Formulation of the Limitation

The reference paper by Dong *et al.* [36] proposes an instance-specific approach, claiming that the utility guarantees degrade gracefully as the hardness 2.6.3 of the data increases.

t	User	Threshold	Users over τ	Contribution	Effective count
$t_{10,000}$	A	$\tau_0 = 64$	999	300	64
$t_{10,001}$	B	$\tau_0 = 64$	999	60	60
$t_{10,002}$	B	$\tau_0 = 64$	999	61	61
$\vdots$	$\vdots$	$\tau_0 = 64$	999	$\vdots$	$\vdots$
t_T	A	$\tau_0 = 64$	999	180,000	64

Table 3.3: Illustration of Truncation Lock permanently stalling at $\tau_0 = 64$.

While mathematically true within the bounds of the asymptotic notation, our analysis demonstrates that in practice, the true maximum contribution $\kappa(D_t)$ is shielded behind the SVT tracking mechanism see Section 2.7.2, which is in reality the limiting factor and possesses a mathematical ceiling.

An analysis of the SVT pseudo-code reveals that the algorithm’s ability to track $\kappa(D_t)$ is dictated by a user voting system.

According to the mechanism’s design, at any epoch i , the SVT triggers the doubling of τ_i when the discounted query $\text{Count}(D_t, \tau_i, \beta_i)$ of users above τ_i exceeds a noisy threshold:

$$\text{trigger if } \left(\text{Count}(D_t, \tau_i, \beta_i) = \text{Count}(D_t, \tau_i) - d_1(i) - d_2(i, t) + \text{Lap}(4/\varepsilon_i) \right) > \eta_i, \quad (3.1)$$

where $\eta_i \sim \text{Lap}(2/\varepsilon_i)$ represents the random Laplace noise that the SVT adds to the baseline threshold, as explained previously in Section 2.7.2. This noisy threshold ensures differential privacy by hiding the exact query count required to trigger an update. The discount factors are defined as:

$$d_1(i) = \frac{6}{\varepsilon_i} \log \left(\frac{2}{\beta_i} \right), \quad d_2(i, t) = \frac{8}{\varepsilon_i} \log(t + 1) \quad (3.2)$$

The first term, $d_1(i)$, guards against the Laplace noise added to the threshold η_i . In the SVT mechanism, this threshold noise is drawn only once at the initialization of an epoch and could potentially take a large negative value. Without correction, this would make the trigger condition overly sensitive and cause an immediate false positive regardless of the actual query value. Thus, $d_1(i)$ provides a fixed, one-time safety buffer that absorbs this worst-case threshold fluctuation.

The second term, $d_2(i, t)$, guards against the noise added to each individual query at every time step t . Because the stream is infinite and a fresh noisy query is issued at each step, the cumulative probability of at least one random noise spike causing a spurious trigger naturally grows over time. To counteract this, $d_2(i, t)$ grows logarithmically with t at a

rate calibrated to outpace the query noise magnitude.

Together, these two terms guarantee that when no user has contributed more than τ_i items, the SVT will trigger incorrectly with a probability of at most β over the entire infinite stream, while keeping the theoretical cost of missed true triggers within the target error budget. Ignoring the random Laplace variables for a baseline analysis, the fundamental mathematical requirement for the SVT to trigger at epoch i becomes:

$$\text{Count}(D_t, \tau_i) \gtrsim d_1(i) + d_2(i, t) \quad (3.3)$$

Lemma 3.1 (The Temporal Cardinality Race): *For the Sparse Vector Technique to trigger a threshold update at epoch i and time t , the number of unique users exceeding the threshold, strictly bounded by the active user cardinality $|U_t|$, must satisfy:*

$$|U_t| \geq \text{Count}(D_t, \tau_i) \gtrsim d_1(i) + d_2(i, t) \quad (3.4)$$

This inequality dictates a temporal race condition. If the active user cardinality $|U_t|$ cannot outpace the required trigger threshold $d_1 + d_2$, the algorithm mathematically cannot update τ , regardless of how massive the individual user contributions $\kappa(D_t)$ become.

To prove why this inequality guarantees the failure of the algorithm under skewed conditions, we must deconstruct the factors d_1 and d_2 . By expanding the definitions of the telescoping privacy budget $\varepsilon_i = \frac{\varepsilon\theta}{(i+1)^{1+\theta}}$ 2.7.3 and the failure probability $\beta_i = \frac{\beta}{(i+1)^2}$ utilized by Dong *et al.* [36], we expose the true magnitude of the constants hidden within the $\tilde{O}$ bound.

3.5.1 Deconstructing the Baseline Noise Penalty (d_1)

To support an infinite stream, the mechanism allocates privacy using a telescoping series, ensuring that the total privacy consumption over infinitely many epochs never exceeds $\sum_{i=1}^{\infty} \varepsilon_i = \varepsilon/2$, see Section 2.7.3.

As stated by Dong *et al.* [36], the d_1 factor acts as the baseline noise penalty because it mathematically cancels the Laplace noise that the SVT adds to the threshold η . By substituting the definitions of ε_i and β_i into Equation 3.2 at epoch i , d_1 becomes:

$$d_1(i) = \frac{6(i+1)^{1+\theta}}{\varepsilon\theta} \log \left(\frac{2(i+1)^2}{\beta} \right) \quad (3.5)$$

Equation 3.5 reveals a severe structural limitation. Because the total privacy budget ε must be fractured into exponentially shrinking fractions, the inverse term $1/\varepsilon_i$ forces the d_1 threshold to inflate at a polynomial-logarithmic rate of $O(i^{1+\theta} \log i)$ through the τ

doubling epochs.

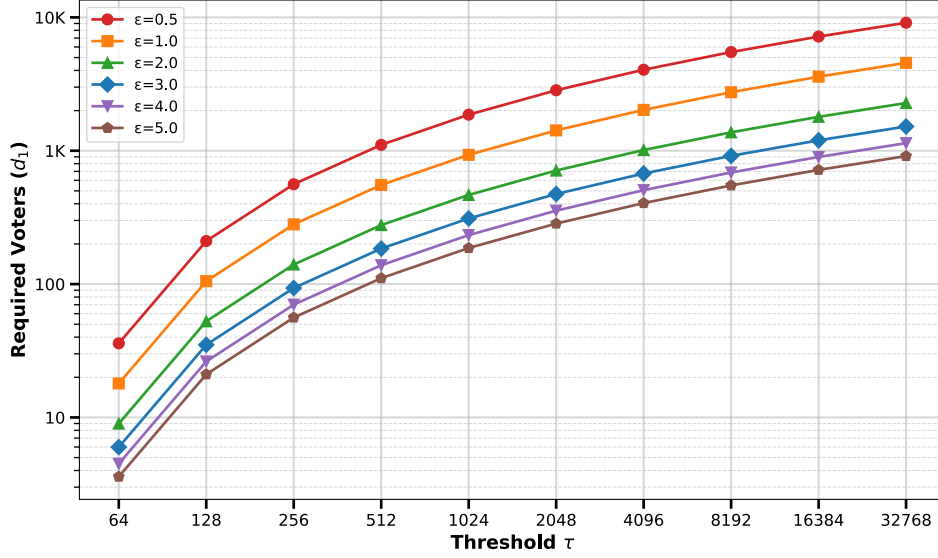


Figure 3.6: Required voters for d_1 across τ epochs, for different values of ϵ .

As illustrated in Figure 3.6, even at the beginning of the stream ($t = 0, d_2 = 0$), each successive doubling of τ is exponentially harder to achieve than the last. Within a few epochs, the algorithm strictly demands tens of thousands of highly active users simply to pass the internal noise check. This requirement can rapidly become an obstacle in real-world scenarios.

3.5.2 Deconstructing the Temporal Penalty (d_2)

While d_1 penalizes the algorithm across epochs, the d_2 factor manages the noise accumulation continuously across time. Dong *et al.* state that this second discount term attempts to remove the Laplace noise the SVT adds to each query over time [36]. Expanding d_2 yields:

$$d_2(i, t) = \frac{8(i+1)^{1+\theta}}{\epsilon\theta} \log(t+1) \quad (3.6)$$

Without this penalty, the continuous addition of noise at every time step t would guarantee a false SVT trigger. However, our analysis reveals that this precise defensive term guarantees asymptotic failure for streams with stabilizing user populations. Because d_2 scales proportionally to $\log(t+1)$, the threshold requirement continuously drifts upward as the stream advances regardless of the threshold epoch i .

As demonstrated in Figure 3.7, for the algorithm to remain functional, the arrival rate of *new, distinct heavy hitters* must perpetually outpace the growth of d_2 . This exposes the challenge of infinite streaming under this mechanism: as $t \rightarrow \infty, d_2 \rightarrow \infty$. Therefore,

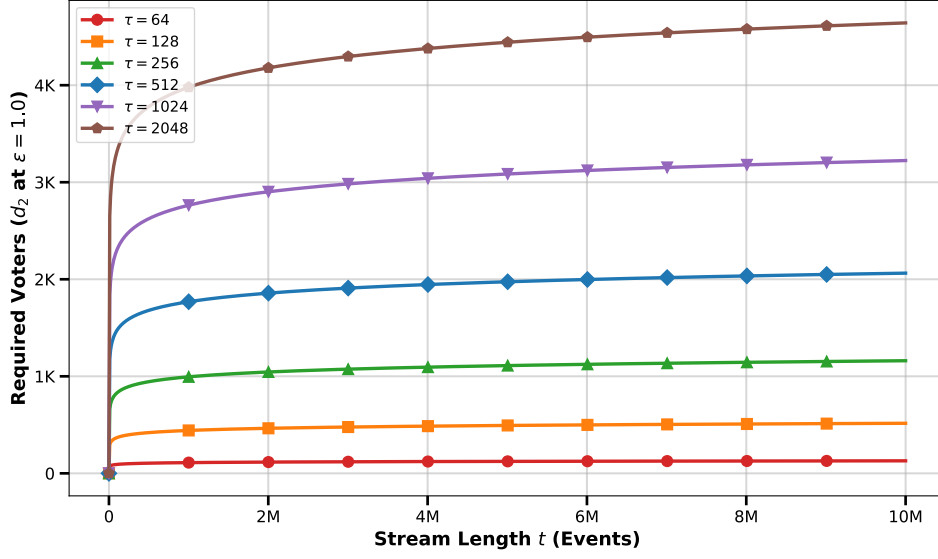


Figure 3.7: Required voters for d_2 across t events, for $\tau \in \{64, 128, 256, 512, 1024, 2048\}$ and $\varepsilon = 1$.

if a stream’s user cardinality stabilizes ($|U_t| \rightarrow |U|$), the required threshold will pseudo-deterministically surpass $|U_t|$. At that exact timestamp, the algorithm enters permanent Truncation Lock 3.4.

3.5.3 Combining d_1 and d_2

The true severity of the algorithm’s limitation is fully realized when we evaluate the combined threshold requirement ($d_1 + d_2$) against the reality of statistical distributions.

The left side of the SVT trigger inequality (Equation 3.3) dictates that a user only qualifies as a “voter” if their event count exceeds τ . We define the combined required threshold $d_1 + d_2$ as the *Algorithmic Demand*.

In reality, it is quite common for datasets to be highly skewed, typically following power-law or Zipfian distributions [86]. In such distributions, the maximum user contribution $\kappa(D_t)$ consumes a massive percentage of the total stream. Consequently, the vast majority of the user base acts as “ghost users” who contribute very few events and never cross the τ threshold. As τ doubles with each epoch, the eligible voters monotonically and drastically decreases.

Figure 3.8 visualizes this phenomenon by mapping the algorithmic demand at $t = 0$ against standard statistical distributions. This creates a mathematically guaranteed *halting point* for the algorithm. It operates on the assumption that a stream will organically supply enough heavy hitters to overcome the SVT noise thresholds. In reality, the failure is dictated by:

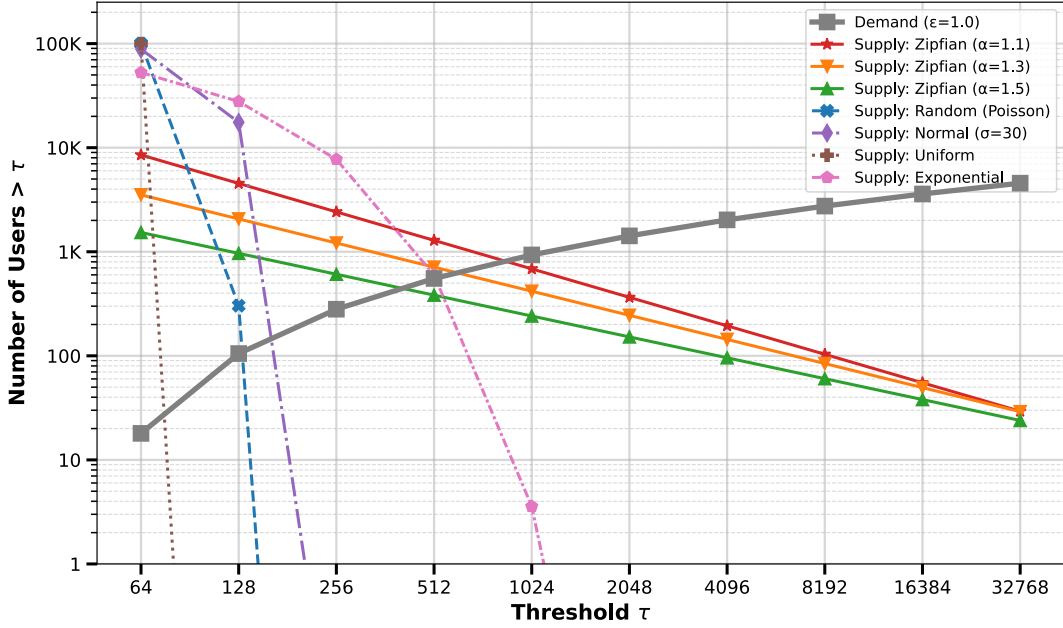


Figure 3.8: Demand (threshold) vs. available voters

- The **Algorithmic Demand** ($d_1 + d_2$) scales *upwards* across both epochs and time.
- The **Available Voters** scale *downwards* across any valid mathematical probability distribution as τ increases.

Because these two curves are guaranteed to intersect, the algorithm will inevitably reach an epoch where Demand exceeds Supply. For instance, as seen in Figure 3.8, even under a relaxed privacy budget of $\epsilon = 5.0$, the Zipfian supply curve goes under the demand curve at epoch 5, and the algorithm permanently stalls.

3.5.4 The Consequence: Unbounded Truncation Bias

When the algorithm reaches this pseudo-deterministic intersection point and succumbs to Truncation Lock, the threshold freezes at an estimated value $\tilde{\tau}_t \ll \kappa(D_t)$.

At this stage, the theoretical utility bound claimed in the reference paper breaks down. The true error is no longer governed by the injected Laplace noise, but by an unbounded truncation bias. Because the algorithm systematically discards every event above the frozen $\tilde{\tau}_t$ threshold, the resulting bias scales linearly with both the magnitude of the heavy hitters and the number of truncated users:

$$\text{Bias}_U \approx - \sum_{u \in U} \max \left(\text{Count}(u, D_t) - \tilde{\tau}_t, 0 \right) \quad (3.7)$$

Because the algorithm drops any events exceeding $\tilde{\tau}_t$, the true count is severely under-reported. Specifically, the deficit is the exact difference between the true user contributions and the frozen threshold, aggregated across all users whose events are discarded. For highly skewed datasets like LastFM, this equates to silently ignoring over $183k$ valid records from the heaviest user alone. Ultimately, this destroys query utility and yields relative errors that approach 100% which validate our empirical observations of catastrophic algorithmic failure.

3.6 Estimating the relative error

In this section, we attempt to estimate the relative error of a given dataset. Our motivation came from our need to understand why certain datasets performed better than others as demonstrated in previous sections, but also to give a preliminary estimation prior to running the algorithm. This would avoid us having to run the algorithm and help us predict the output on datasets.

3.6.1 The Halting Epoch k

While the previous sections establish the static conditions for Truncation Lock 3.4, the SVT mechanism is inherently dynamic. The algorithm does not simply fail at the initial threshold τ_0 ; rather, it successfully doubles τ through successive epochs $i = 0, 1, 2, \dots$ until the available supply of voters can no longer satisfy the inflating threshold.

Let $D(i, T)$ represent the **Algorithmic Demand** evaluated at the end of the stream length T . This is the absolute minimum number of voters required to successfully transition from epoch i to epoch $i + 1$:

$$D(i, T) = d_1(i) + d_2(i, T) \quad (3.8)$$

Let c_u represent the total number of events contributed by user u , such that $c_u = \text{Count}(u, D_T)$. The **Voter Supply** at epoch i , $V(\tau_i)$, is the exact number of unique users whose contributions exceed the threshold $\tau_i = \tau_0 \cdot 2^i$:

$$V(\tau_i) = |\{u \in U \mid c_u \geq \tau_i\}| \quad (3.9)$$

Because the Algorithmic Demand scales upwards across time and epochs, while the Voter Supply scales downwards as τ_i increases as seen in Section 3.5.3, the curves will intersect at some point when there is not enough supply voters for the algorithm demand. We formalize this intersection as the **Halting Epoch** (k):

$$k_{halt} = \min\{i \in \mathbb{N} \mid V(\tau_i) < D(i, T)\} \quad (3.10)$$

Once the algorithm reaches epoch k_{halt} , the SVT mechanism is permanently starved of updates. The truncation threshold locks pseudo-deterministically at $\tau_{lock} = \tau_0 \cdot 2^{k_{halt}}$.

3.6.2 The Concentration Ratio

When the algorithm enters *Truncation Lock* at τ_{lock} , the contribution threshold stops evolving while the stream continues to grow. In this regime, the Laplace noise becomes fractionally negligible relative to the total stream mass, i.e. $\lim_{T \rightarrow \infty} \frac{\text{Lap}(\cdot)}{T} = 0$, and the Relative Error is therefore dominated by the truncation bias.

To capture this, we define the **Concentration Ratio** as the proportion of the stream's total mass carried by users whose total contribution exceeds the threshold τ :

$$\rho(\tau) = \frac{1}{T} \sum_{u: c_u \geq \tau} c_u \quad (3.11)$$

where c_u is the total number of events contributed by user u , and $T = \sum_{u \in U} c_u$ denotes the total stream mass. The quantity $\rho(\tau)$ therefore measures how much of the entire stream is concentrated in the heavy-hitter users (those that are above τ).

Figure 3.9 provides an intuitive visualization. Users are sorted in decreasing order of contribution, and the horizontal dashed line marks the locked threshold τ_{lock} . The blue bars correspond to users with $c_u \geq \tau_{\text{lock}}$, whose contribution mass is the sum in Equation 3.11, while the gray bars correspond to users below the threshold.

A high value of $\rho(\tau_{\text{lock}})$ means that a large fraction of the entire stream is generated by users who lie above the frozen truncation threshold. In that case, once the mechanism is locked, most of the stream mass is exposed to clipping, and the resulting truncation bias dominates the long-run error.

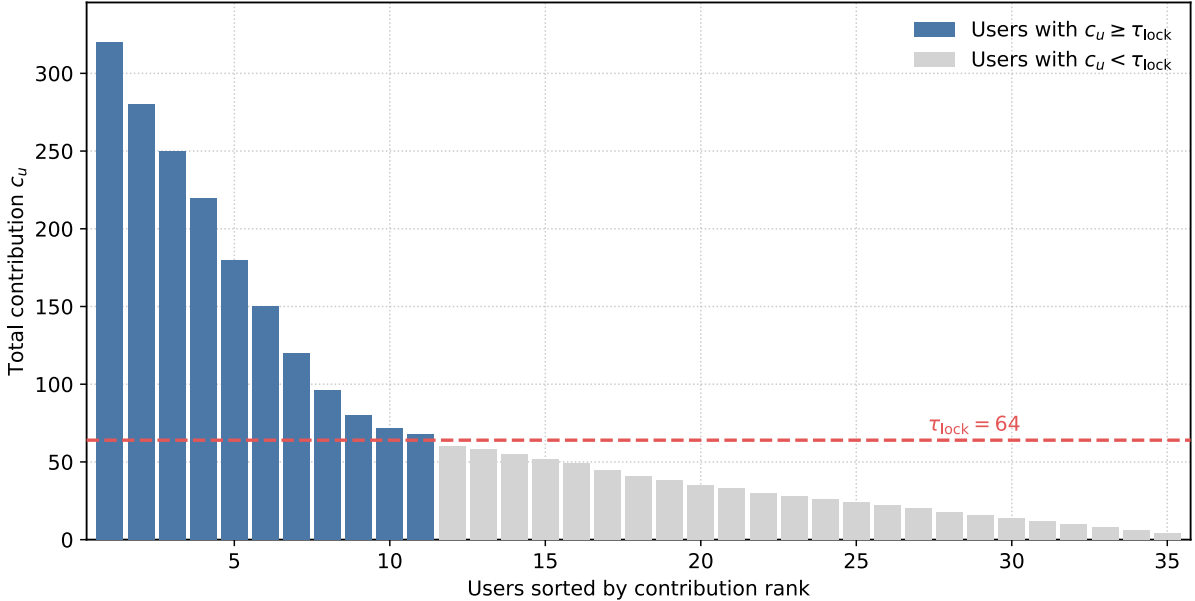


Figure 3.9: Illustration of the concentration ratio at the locked threshold τ_{lock} . Users are sorted by decreasing contribution

3.6.3 Relative error

Once the algorithm has entered Truncation Lock at τ_{lock} , the dominant source of utility loss is no longer the injected noise but the exact mass discarded by truncation. For every user u such that $c_u \geq \tau_{\text{lock}}$, the mechanism preserves at most τ_{lock} events and discards the remaining $c_u - \tau_{\text{lock}}$. Therefore, the predicted Relative Error is the total discarded mass divided by the total stream mass T :

$$RE_{\text{pred}} = \frac{1}{T} \sum_{u: c_u \geq \tau_{\text{lock}}} (c_u - \tau_{\text{lock}}) \quad (3.12)$$

We can decompose the numerator into two interpretable terms:

$$\sum_{u: c_u \geq \tau} (c_u - \tau) = \sum_{u: c_u \geq \tau} c_u - \tau \sum_{u: c_u \geq \tau} 1 = T \cdot \rho(\tau) - \tau \cdot V(\tau) \quad (3.13)$$

where $\rho(\tau)$ is the Concentration Ratio defined in Equation 3.11, and $V(\tau)$ is the Voter Supply, i.e. the number of users whose contribution reaches or exceeds τ .

Substituting Equation 3.13 into Equation 3.12, we obtain:

$$RE_{\text{pred}} = \rho(\tau_{\text{lock}}) - \frac{V(\tau_{\text{lock}}) \cdot \tau_{\text{lock}}}{T} \quad (3.14)$$

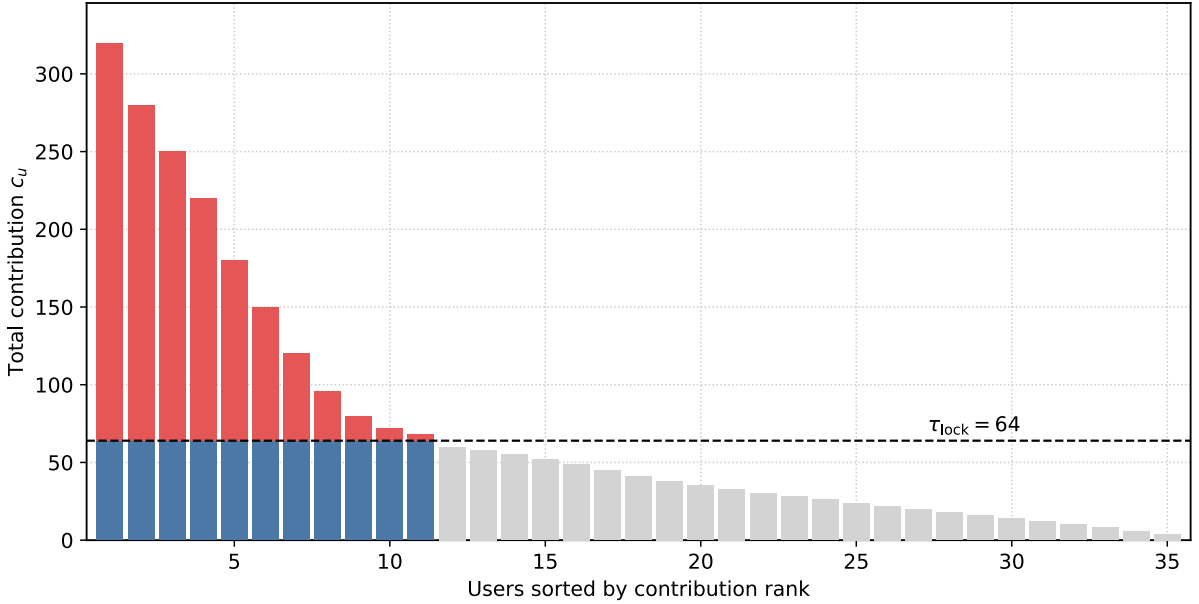


Figure 3.10: Illustration of the predicted Relative Error at the locked threshold τ_{lock} . Users are sorted by decreasing contribution.

Figure 3.10 illustrates this decomposition visually. For each user above τ_{lock} , the lower segment of the bar (blue) represents the contribution mass legally preserved by the mechanism, while the upper segment (red) represents the discarded mass. The predicted Relative Error is exactly the red discarded area normalized by the total mass T .

Equation 3.14 therefore shows that the predicted error is dominated by two competing quantities: the total fraction of user stream mass concentrated above the locked threshold (indicated by the red and blue in the figure), and the exact fraction of that mass that the mechanism is still allowed to retain (indicated by the blue color on the figure). In highly skewed streams, the first term dominates and the predicted Relative Error approaches 1.

3.6.4 The Pre-Execution Prediction Algorithm

From equation 3.14, we can construct a deterministic prediction algorithm. Instead of deploying the full algorithm and resource-intensive Flink application to discover if the algorithm will fail, using this prediction we can compute the exact failure state and utility loss *a priori*, using only the raw frequency counts of the dataset.

Given a sorted array of user contributions $c_1 \geq c_2 \geq \dots \geq c_{|U|}$, the algorithm evaluates the threshold conditions iteratively:

Algorithm 1 Predict Relative Error

Input: User contributions $C = \{c_u\}_{u \in U}$, privacy budget ε , failure probability β , decay rate θ , initial threshold τ_0

Output: Halting epoch k_{halt} , locked threshold τ_{lock} , predicted relative error RE_{pred}

```

1:  $T \leftarrow \sum_{u \in U} c_u$  ▷ Compute total stream length
2:  $i \leftarrow 0$ 
3: while true do
4:    $\tau_i \leftarrow \tau_0 \cdot 2^i$ 
5:    $V_i \leftarrow |\{u \in U \mid c_u \geq \tau_i\}|$  ▷ Count available voters
6:    $\varepsilon_i \leftarrow \frac{\varepsilon \theta}{(i+1)^{1+\theta}}$  ▷ Update telescoping budget
7:    $\beta_i \leftarrow \frac{\beta}{(i+1)^2}$  ▷ Update failure probability
8:    $D_i \leftarrow \frac{6}{\varepsilon_i} \log\left(\frac{2}{\beta_i}\right) + \frac{8}{\varepsilon_i} \log(T+1)$  ▷ Compute algorithmic demand
9:   if  $V_i < D_i$  then ▷ Mechanism starves
10:     $k_{halt} \leftarrow i$ 
11:     $\tau_{lock} \leftarrow \tau_i$ 
12:    break
13:  end if
14:   $i \leftarrow i + 1$ 
15: end while
16:  $\rho(\tau_{lock}) \leftarrow \frac{1}{T} \sum_{u: c_u \geq \tau_{lock}} c_u$  ▷ Compute Concentration Ratio
17:  $RE_{pred} \leftarrow \rho(\tau_{lock}) - \frac{V_{k_{halt}} \cdot \tau_{lock}}{T}$  ▷ Apply Equation 3.14
18: return  $(k_{halt}, \tau_{lock}, RE_{exact})$ 
```

Because the input array can be sorted, determining V_i via binary search ensures the entire prediction process operates in $O(|U| \log |U|)$ time.

Chapter 4

Evaluation

4.1 Experimental Setup

4.1.1 Hardware and Software Setup

To ensure reproducibility and consistent performance measurement, all experiments are executed on a single dedicated machine. We rely on containerized deployments of the stream processing frameworks. The hardware specifications and software versions are summarized in Table 4.1.

Component	Specification
<i>Hardware</i>	
Machine	Lenovo ThinkPad T16
Processor	AMD Ryzen AI 7 PRO 350 (8 cores, 16 threads)
Memory	28 GB LPDDR5
Storage	High-speed PCIe NVMe SSD
<i>Software & Frameworks</i>	
Operating System	Arch Linux (Kernel 7.0.3)
Containerization	Docker 29.4.2
Stream Engine	Apache Flink 1.17.0
Message Broker	Apache Kafka 7.4.0 (Confluent Platform)

Table 4.1: Summary of the hardware and software environment used for all evaluations.

The Flink JobManager and TaskManager are co-located on the same host. Although the TaskManager is configured with two available slots, we strictly limit the execution parallelism to 1 during the experiments, utilizing only a single slot. This matches the single-partition design of the `FcntUserDPOperator` described in Section 3.2.2, ensuring that all events are routed to and processed by a single, unified state machine.

4.1.2 Metrics

- **Relative Error (RE%):** The primary utility metric, defined at every release step t as $RE(t) = 100 \cdot |\hat{c}_t - c_t| / c_t$, where c_t is the exact continual count and $\hat{c}_t$ is the noisy DP count.
- **Continual Count:** The raw value of the DP release $\hat{c}_t$ plotted alongside the exact count c_t , allowing visual inspection of systematic under or over-estimation.
- **Process CPU% Load:** The CPU utilisation of the JVM process hosting the Flink task manager, sampled at one-second intervals throughout the run.
- **Processing Throughput:** The number of stream events processed per second by the Flink operator, computed from operator-level metrics exposed by Flink.
- **Pipeline Latency:** The time it takes for a single event to go through our pipeline, with DP applied to it vs. without DP applied to it.
- **Operator Latency:** The time it takes for an event to be counted, by the DP operator vs. a normal counting operator.

4.1.3 Justification of Privacy Hyperparameters

The theoretical mechanism requires specifying four hyperparameters: the total privacy budget ε , the failure probability β , the telescoping decay rate θ , and the initial threshold τ_0 . Our chosen values ($\varepsilon \in \{1, 10, 100\}$, $\beta = 0.1$, $\theta = 1.0$, $\tau_0 = 64$) were deliberately selected to balance theoretical rigor with practical stream operability.

The relationship between ε and θ . As established in Section 3.5, the Algorithmic Demand ($d_1 + d_2$) required to trigger an SVT threshold update is inversely proportional to the privacy budget ε , while $\varepsilon = 1$ and 10 are acceptable in practice 2.3.4, the value $\varepsilon = 100$ has the role of identifying inherently bad datasets. A higher ε lowers the voter threshold, and therefore the risk of *truncation lock*. The Algorithmic Demand is also influenced by θ , which dictates how the total budget ε is split across the epochs with the help of the telescoping series 2.7.3.

To understand how this budget split affects long-term utility, we evaluated the interaction between ε and θ on our worst performing dataset, LastFM. We performed a parameter sweep to observe the resulting average relative error (the full heatmap of this evaluation is available in Appendix A.4). Our results confirmed that $\theta = 1.0$ consistently provided a robust, balanced performance across all tested ε regimes. Consequently, we decided to retain $\theta = 1.0$, remaining faithful to the original theoretical baseline proposed by Dong *et al.* [36].

The failure probability β . The failure probability was kept strictly at $\beta = 0.1$. In the SVT 2.7.2, the mechanism allocates failure probabilities across epochs such that their infinite sum never exceeds β . In practice, setting $\beta = 0.1$ caps the probability of the SVT firing a false positive, caused by noise addition, to at most 10% over the *entire* lifetime of

the stream. This provides a high level of statistical confidence for the threshold-tracking mechanism without excessively inflating the baseline noise penalty d_1 3.5.

The *initial threshold* τ_0 . Finally, we kept the initial truncation threshold at $\tau_0 = 64$, matching the starting parameter used in the original paper. While tuning this parameter might temporarily mask utility issues, it does not resolve the underlying vulnerability.

For instance, setting a *lower* τ_0 might initially seem beneficial because it allows the SVT to fire easily early on, providing acceptable initial utility. However, because the required number of voters $d_1(i)$ 3.6 strictly increases with each epoch i , inherently skewed datasets will inevitably fail to sustain these updates later in the stream, causing the error to quickly diverge. Conversely, setting a *higher* τ_0 to reach a higher final hardness does not change the fact that the algorithm still stalls for hard datasets.

The fundamental blocking point is thus primarily dictated by the Algorithmic Demand ($d_1 + d_2$) 3.5.3, which is mathematically independent from τ . Therefore, our focus is not on tuning τ_0 to delay utility collapse, but rather on exposing the structural mechanisms that make the algorithm inherently fail on datasets with specific distributions.

4.2 Utility Evaluation

4.2.1 Behavior Under Favorable and Adverse Regimes

In this subsection, we evaluate utility on the two datasets that bracket the full behavioral range of the mechanism: the Netflix stream (favorable conditions) and the LastFM stream (adverse conditions leading to Truncation Lock, see Section 3.4). For each dataset we examine three privacy budgets so as to isolate the effect of ϵ from the structural effect of the data distribution. The results of the other datasets from Table 3.1 can be found in Appendix A.8 and A.10.

Experimental Parameters

- **Executions:** 1 Warmup and 3 Runs
- **Privacy Parameters:** $\epsilon \in \{1, 10, 100\}$, $\theta = 1.0$ and $\beta = 0.1$ (default parametrization proposed by Dong et al. [36])
- **Release Step:** From 100 for smaller datasets up to $50k$ for larger ones (resulting in $2k$ evaluation points for each dataset)

Netflix Stream: High Cardinality, Moderate Skew

The utility results for the Netflix dataset, characterized by its high user cardinality ($|U| \approx 480,189$) and moderate contribution skew, are presented in Figures 4.1 and 4.2. This dataset represents a regime where the voter supply is sufficient to overcome the

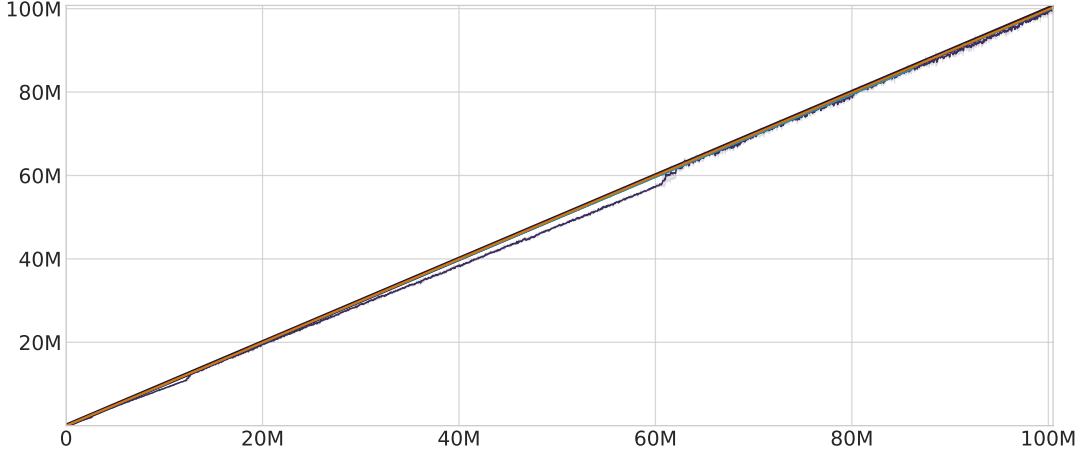


Figure 4.1: Continual DP count vs. exact count on the Netflix stream ($T \approx 100\text{M}$ events) for $\varepsilon \in \{1, 10, 100\}$ (dark blue, teal, orange). The thick black line represents the exact reference.

algorithmic demand of the SVT.

Continual count. As shown in Figure 4.1, all three privacy budget configurations maintain a DP count that closely tracks the exact reference throughout the entire 100M event stream. While the curves for $\varepsilon = 10$ and $\varepsilon = 100$ are nearly indistinguishable from the exact baseline at this scale, the curve for $\varepsilon = 1$ exhibits a slightly larger spread due to the increased Laplace noise, as well as the larger discount terms d_1 and d_2 from Equation 3.2.

Crucially, no systematic downward bias is observed in any of the configurations. This indicates that the mechanism is not silently discarding significant volumes of data, the SVT successfully doubles the truncation threshold τ with enough frequency to keep pace with the evolving maximum user contribution $\kappa(D_t)$.

Relative Error. Figure 4.2 provides a more granular view of utility loss. Three observations are prominent.

First, for all budgets, the relative error is highest at the beginning of the stream (approaching 100%) and rapidly decays. This is a predictable artifact of the counting query: at time $t \approx 0$, the true count is small, making even modest absolute noise appear catastrophic. As the total mass c_t grows, the noise becomes fractionally negligible.

Second, the long-run error is strictly ordered by ε . Consistent with the Laplace mechanism’s variance properties, a tighter budget ($\varepsilon = 1$, dark blue) results in a persistently higher relative error than a permissive one ($\varepsilon = 100$, orange). After approximately 20M events, the relative error for $\varepsilon = 100$ stabilizes near zero, while $\varepsilon = 1$ stabilizes at a slightly higher plateau.

Third, and most importantly for the validation of our implementation, none of the curves

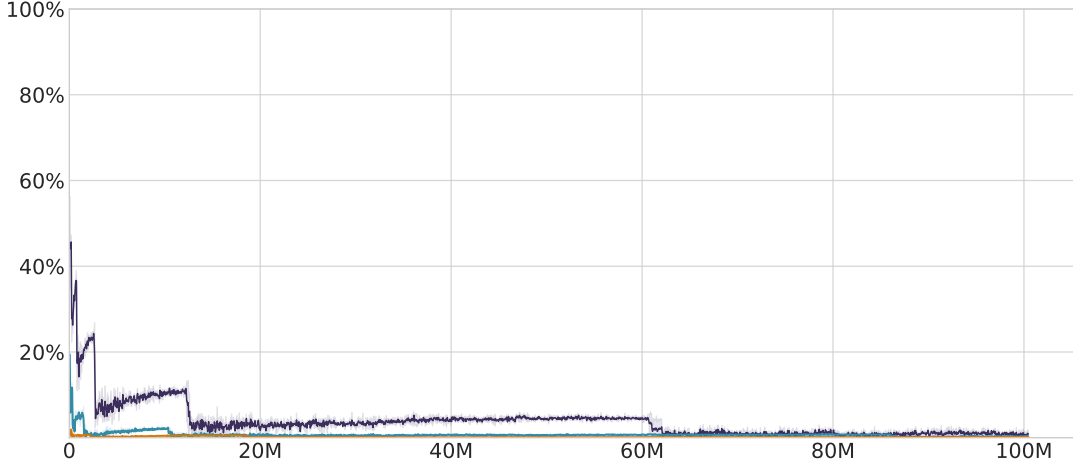


Figure 4.2: Relative error over time for the Netflix stream for $\varepsilon \in \{1, 10, 100\}$ (dark blue, teal, orange). The initial spike represents the cold-start phase where low counts make absolute noise appear large in relative terms.

exhibit the monotone upward drift characteristic of Truncation Lock (as detailed in Section 3.4). The error either stabilizes or continues to decrease slightly as the stream progresses. This confirms that on the Netflix dataset, the mechanism operates as intended by Dong et al. [36], successfully balancing the privacy-utility trade-off across a long-term horizon.

LastFM Stream: Low Cardinality, Extreme Skew

The results for the LastFM stream, presented in Figures 4.3 and 4.4, serve as the primary empirical validation of the *Truncation Lock* phenomenon 3.4. With a limited user base ($|U| = 992$) and a maximum user contribution ($\kappa(D) = 183,103$) that is nearly 200 times larger than the total population, this dataset violates the fundamental assumptions required for the SVT to trigger threshold updates.

Continual Count. Figure 4.3 exhibits a dramatically different pattern from the favorable Netflix regime. While the exact count grows linearly to approximately 19M events, the DP curves significantly undershoot the reference. Notably, the curve for the standard budget ($\varepsilon = 1$, purple) is effectively invisible at this scale, remaining near the x-axis for the duration of the stream due to its inability to trigger a single doubling.

As established in the theory of *Algorithmic Demand* (Section 3.5), a higher ε reduces the voter threshold required for an update. This allows the configurations for $\varepsilon = 10$ (teal) and $\varepsilon = 100$ (orange) to perform threshold doublings, visible as the "steps" in the count. However, these doublings do not occur frequently enough to keep pace with the massive contributions of heavy hitters. In the intervals between updates, the gap between the exact and DP counts grows as events are discarded by a threshold that is effectively "stalling". This illustrates the core mechanics of Truncation Lock (Section 3.4): the SVT is chronically starved of voters due to the low user cardinality, causing the error to remain

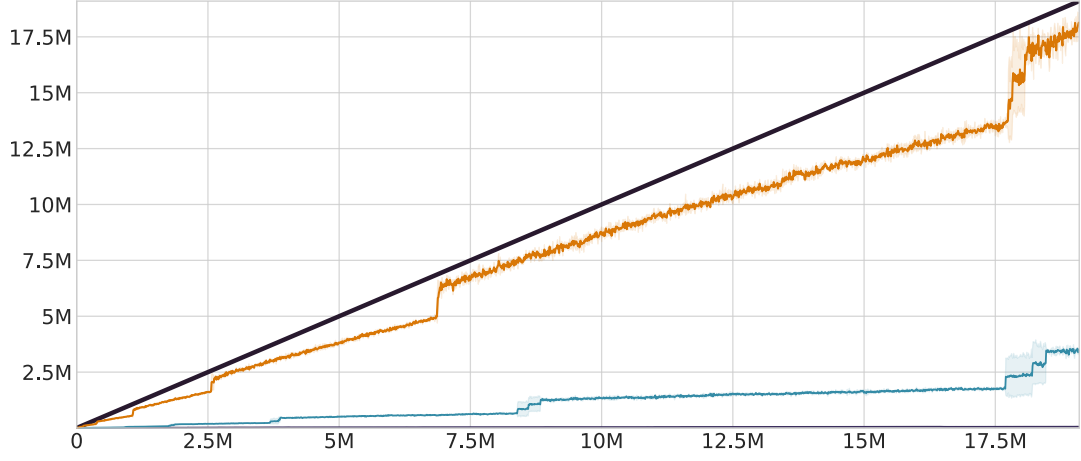


Figure 4.3: Continual DP count vs. exact count on the LastFM stream ($T \approx 19\text{M}$ events) for $\varepsilon \in \{1, 10, 100\}$ (dark blue, teal, orange). The thick black line represents the exact reference.

bias-dominated as modeled in Equation (3.7).

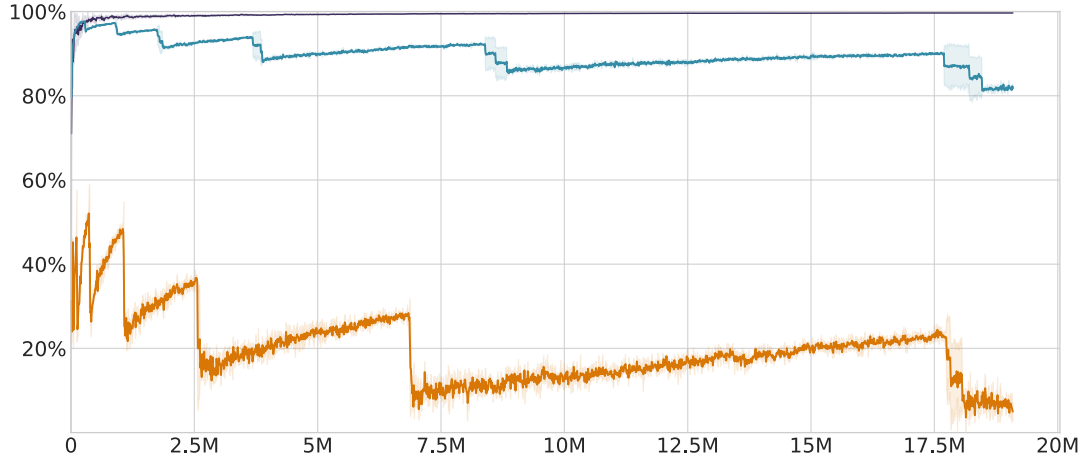


Figure 4.4: Relative error over time for the LastFM stream across three privacy budgets for $\varepsilon \in \{1, 10, 100\}$ (dark blue, teal, orange).

Relative error. The relative error shown in Figure 4.4 makes this failure quantitative. While $\varepsilon = 1$ sustains nearly 100% relative error, the higher budgets exhibit a "sawtooth" pattern. Each sharp drop in error corresponds to a successful threshold update, immediately followed by a monotonic increase. This drift occurs because the truncated mass grows without bound while the number of events from users below τ remains insufficient to provide a counterweight.

While utility appears better as the budget increases, it is important to note that $\varepsilon = 100$ represents a mathematically vacuous level of protection provides essentially "no privacy" as explained in section 2.3.4. Yet, even with the privacy constraints effectively removed, the mechanism fails to maintain a low relative error and continues to suffer from periodic

upward drift.

This behavior matches the theoretical prediction of the Concentration Ratio $\rho(\tau)$ derived in Section 3.6.2. Because the top few users generate the overwhelming majority of the stream mass, the retained fraction $(V(\tau_{\text{lock}}) \cdot \tau_{\text{lock}})/T$ is negligible, leaving the error to be dominated by the discarded mass.

This demonstrates that in the Truncation Lock regime, **utility is limited by the structural properties of the data distribution** rather than the statistical noise of the mechanism. If the user cardinality is too low to satisfy the SVT, no amount of privacy budget can recover the true count.

4.2.2 Predicting the Relative Error

In this subsection, we validate the relative error prediction models derived in Section 3.6. Using targeted synthetic datasets, we isolate the Concentration Ratio and temporal arrival dynamics to observe their direct impact on the algorithmic failure state.

Experimental Parameters

- **Executions:** 1 Warmup and 3 Runs
- **Privacy Parameters:** $\varepsilon = 1$, $\theta = 1.0$ and $\beta = 0.1$ (default parametrization proposed by Dong et al. [36])
- **Release Step:** From 100 for smaller datasets up to $50k$ for larger ones (resulting in $2k$ evaluation points for each dataset)

Concentration Ratio Evaluation

Dataset generation. We constructed synthetic streams designed to force the Dong *et al.* mechanism into an immediate, deterministic Truncation Lock (Section 3.4) at the initial epoch ($\tau_{\text{lock}} = \tau_0 = 64$). Each dataset contains $T \approx 10\text{M}$ randomly shuffled events, divided strictly into two populations:

- **Baseline group:** Every user contributes exactly $B = 64 = \tau_0$ events $((1 - \rho) \cdot T$ of the stream mass).
- **Heavy group:** $N_{\text{over}} \leq 100$ users contributing $H \gg \tau_0$ events $(\rho \cdot T$ of the stream mass).

Because the 100 heavy users fall short of the 147 voters required to trigger the SVT at initialization, the threshold never updates. Under these conditions, the negative term in Equation 3.14 is strictly bounded:

$$\frac{V(\tau_{\text{lock}}) \cdot \tau_{\text{lock}}}{T} \leq \frac{100 \cdot 64}{10\text{M}} = 0.00064$$

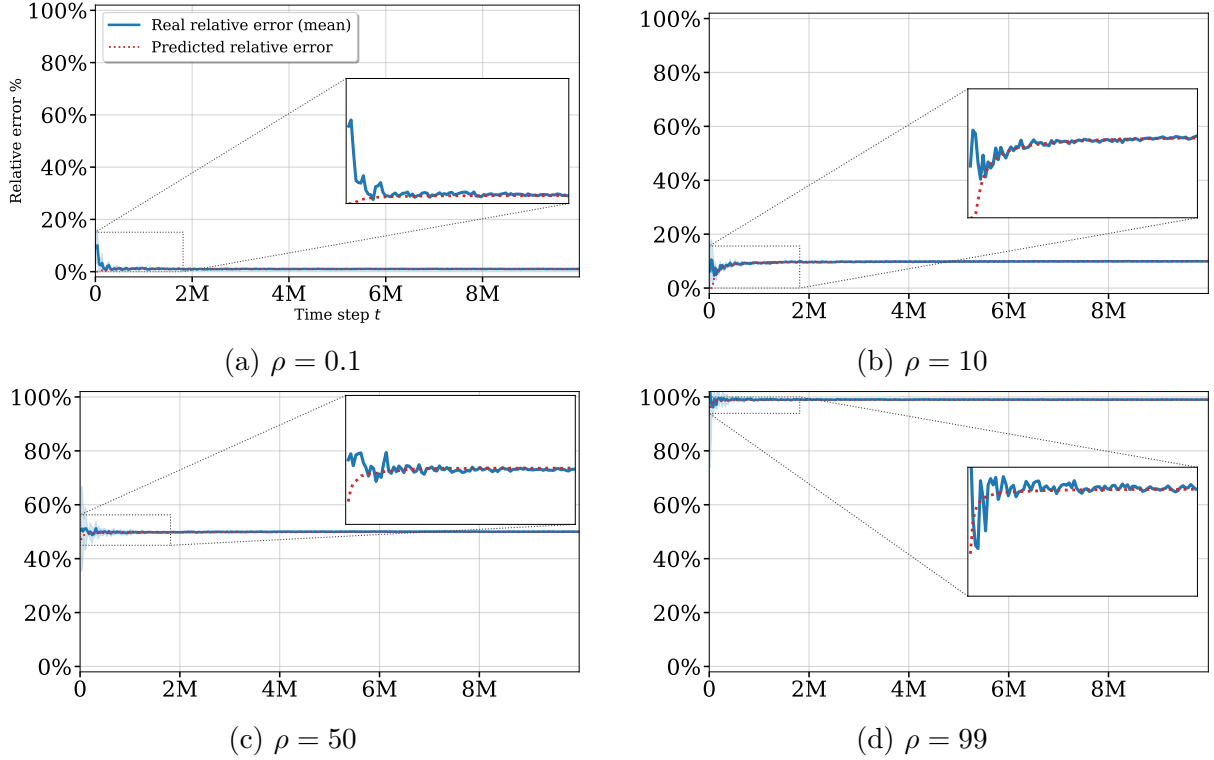


Figure 4.5: Real relative error vs Predicted relative error across time for different Concentration Ratios ρ

This simplifies our theoretical prediction to $RE_{\text{pred}} \approx \rho(\tau_{\text{lock}})$. We synthesized four distinct datasets with concentration ratios $\rho \in \{0.1\%, 10\%, 50\%, 99\%\}$.

Relative error. Figure 4.5 presents the continuous relative error across the four synthetic streams. The empirical results conclusively validate our mathematical model: the measured relative error (blue line) converges precisely to the theoretical prediction (red line), rendering them nearly indistinguishable.

In the early stages of the stream ($t \approx 0$), slight instability is visible. This behavior is expected because the total true count is initially small, making the logarithmic Laplace noise injected by the Binary Tree Mechanism 2.7.1 a visible fraction of the output. However, as the stream volume grows linearly, the noise magnitude grows only logarithmically ($O(\log t)$). Consequently, the noise fraction rapidly approaches zero ($\lim_{t \rightarrow \infty} \frac{\text{Noise}}{t} = 0$). Once the DP noise dissipates, the truncation bias becomes the absolute and sole driver of utility loss.

Because the SVT is locked at $\tau_{\text{lock}} = 64$, the mechanism systematically discards the vast majority of heavy hitter events. As demonstrated in Figure 4.5d, when $\rho = 99\%$, the algorithm effectively ignores 99% of the valid stream data, yielding a flat, near-100% relative error. These findings confirm the fundamental limitation identified in Chapter 3: when user cardinality cannot overcome the SVT’s Algorithmic Demand, accuracy is deterministically bounded by the stream’s concentration ratio.

Temporal Distribution and User Arrival Dynamics

While the Concentration Ratio (ρ) dictates the asymptotic failure state, global statistics alone cannot predict continuous utility over a stream’s lifetime. Because the SVT evaluates the stream sequentially, the *order* and *timing* of user participation fundamentally alter the mechanism’s behavior.

Dataset generation. To evaluate the impact of temporal arrival patterns, we contrasted two synthetic streams: a “Random” arrival dataset and a “Burst” arrival dataset. As detailed in Table 4.2, both share nearly identical global statistics. We manipulated only the temporal event ordering.

Metric	Random Arrival Dataset	Burst Arrival Dataset
Total Events (T)	10M	10M
Unique Users ($ U $)	3.26M	3.26M
Max Contribution (κ)	5k	5k
Event Ordering	Uniformly shuffled	Strictly phased
Heavy Hitter Arrival	Spread over time	Front-loaded at start
Voter Arrival	Continual	Massive bursts

Table 4.2: Comparison of the synthetic datasets used to evaluate temporal dynamics.

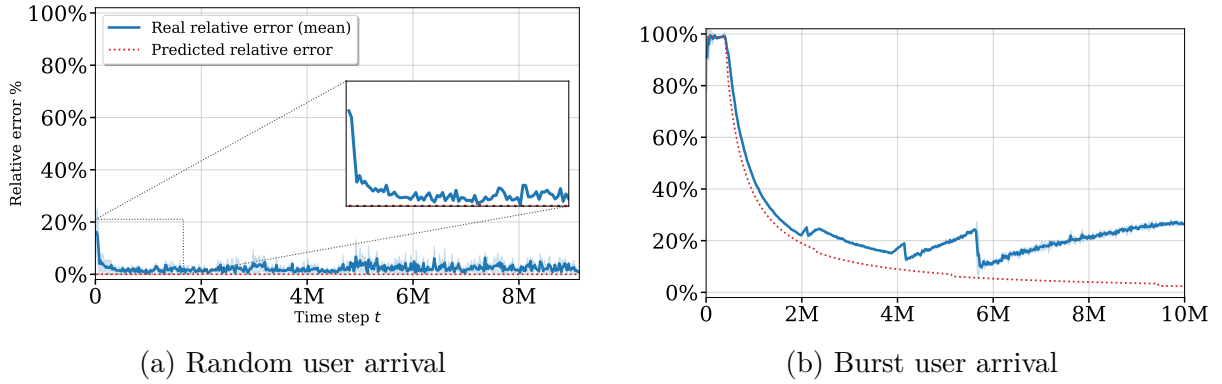


Figure 4.6: Real relative error vs. Predicted relative error across time for different temporal arrival patterns.

Random arrival. When users are perfectly interleaved (Figure 4.6a), the stream steadily supplies enough distinct voters to satisfy the Algorithmic Demand at each epoch. The SVT reliably triggers, scaling τ alongside the maximum user contribution $\kappa(D_t)$. Truncation Lock is completely avoided, and the relative error remains near zero, empirically confirming the optimal theoretical bounds claimed by Dong *et al.* [36].

Burst arrival. Real-world streams rarely exhibit perfect uniformity; users often generate massive, localized bursts of activity. By forcing the stream through distinct temporal phases, we induce the prominent “sawtooth” error pattern visible in Figure 4.6b. This unfolds in three distinct stages:

1. **Early heavy burst:** A small number of heavy hitters contribute thousands of events back-to-back at the start of the stream. Lacking sufficient distinct voters to satisfy the SVT demand, the threshold remains locked at $\tau_0 = 64$. Excess events are immediately truncated, causing the relative error to spike drastically.
2. **Voter bursts:** Later in the stream, a massive wave of moderately active users arrives, suddenly pushing the supply $V(\tau_i)$ past the algorithmic demand of an epoch i . The SVT finally triggers, doubling the threshold.
3. **The add-back drop:** As described in Section 3.2.2, our implementation features an add-back mechanism. When the threshold doubles, the system instantly restores previously truncated events up to the new cap. This causes the sharp, instantaneous drops in relative error.

Between these bursts, inside every τ epoch, the error slowly drifts upwards again as new events from heavy hitters arrive but cannot yet be counted. This reveals a critical deployment vulnerability: even if a dataset possesses favorable global statistics, bursty temporal distributions subject the query to severe, unpredictable utility degradation until the SVT is triggered.

4.3 Performance Evaluation

The utility results establish *when* the mechanism produces valid outputs. The performance results establish *at what cost* those outputs are produced. Three dimensions are examined: CPU overhead (Section 4.3.1), processing throughput (Section 4.3.2), and end-to-end release latency (Section 4.3.3). All performance experiments use the Netflix stream since it provides the longest and most stable steady-state phase, making it the most representative workload for production benchmarking.

4.3.1 CPU Overhead

In this section, we quantify the computational cost of the differential privacy logic by comparing the process-level CPU utilization of the DP pipeline against the exact baseline.

Experimental Parameters

- **Executions:** 1 Warmup and 3 Runs.
- **Privacy Parameters:** $\epsilon = 1$, $\beta = 0.1$, and $\theta = 1.0$.
- **Kafka Producer Rate:** 500,000 events/s, ensuring the consumer is consistently saturated without introducing artificial back-pressure into the Flink pipeline.

Methodology. To ensure a clean measurement of the Flink operator’s cost, we execute the exact baseline and the DP mechanism using dedicated runners, an exact count only and a DP count only. These runners utilize a dedicated background thread to sample the `ProcessCpuLoad` via the `OperatingSystemMXBean` interface every 500 ms. This metric

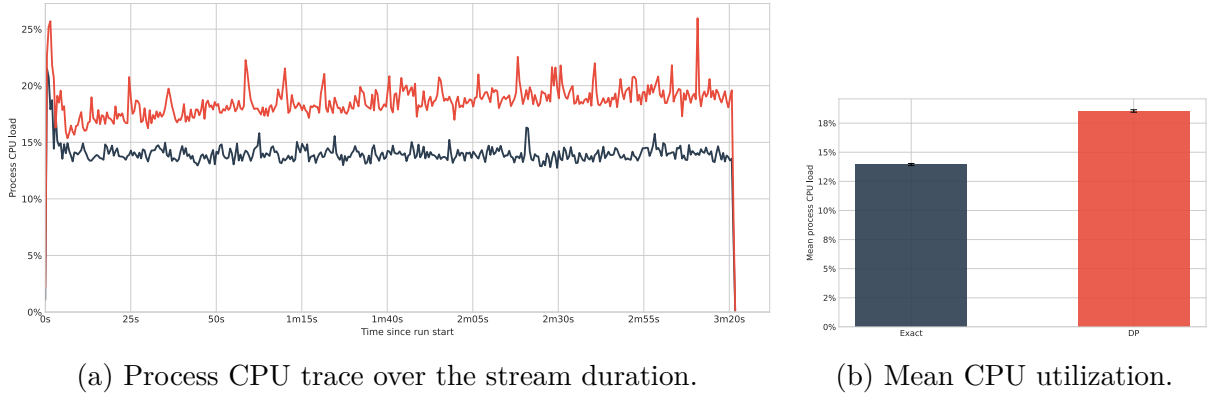


Figure 4.7: CPU utilization of the Flink TaskManager JVM on the Netflix stream. The relative computational overhead of the DP (red) mechanism is approximately 30%.

specifically captures the CPU cycles consumed by the Flink JVM process, effectively isolating the operator’s cost from the Kafka broker and the data producer.

All experiments were performed with a throttled production rate of 500,000 events/s. This rate was chosen to provide a steady-state workload that avoids the non-linear CPU spikes caused by network back-pressure or Kafka ingestion bottlenecks observed at higher frequencies (cf. Section 4.3.2). For each configuration, we performed four consecutive runs, the first run is treated as a warmup to allow the JVM’s Just-In-Time (JIT) compiler to optimize the bytecode, and only the subsequent three runs are used for the final analysis.

Results. As shown in Figure 4.7a, both pipelines exhibit a stable CPU profile throughout the stream duration. The exact baseline maintains a consistent load of approximately 14.0%. In contrast, the DP pipeline operates at a higher baseline of approximately 18.3%, with periodic spikes reaching up to 25%. These spikes correspond to threshold-doubling events where the system must perform an "Add-back" scan to re-process buffered events (see Section 3.2.2 and figure 3.3 for more information about the 'add-back').

Aggregating the results (Figure 4.7b) reveals that the DP mechanism introduces an absolute increase of ≈ 4.3 percentage points in CPU load. This translates to a relative computational overhead of approximately **30%**. The narrow error bars (standard deviation across runs) across runs confirm that this "privacy tax" is highly reproducible and stable.

Sources of overhead. The 30% overhead is driven by the sophisticated state-management and mathematical requirements of the `FcntUserDPOperator`. While the exact counter only requires a single accumulation step, the DP implementation must perform:

1. **Keyed State Access (`UserTrackingOperator`):** Unlike the exact pipeline, the DP pipeline performs per-event `MapState` reads and writes to maintain per-user contribution counters, an operation required to hide a user’s entire temporal footprint (cf. Section 2.6.2). Managing these complex state objects is significantly more expensive than simple numeric accumulation.
2. **Mathematical Sampling (`SVTUserContributionOperator`):** The mechanism

must evaluate the SVT (cf. Subsection 2.7.2) trigger condition at every event. This involves frequent random number generation to sample Laplace-distributed noise, an operation that adds a constant CPU tax to every processed record.

3. **Tree Arithmetic and Buffer Management (`EventDPBinaryTreeOperator`):** The pipeline performs bitwise tree-level arithmetic for the dyadic intervals (cf. Subsection 2.7.1). Furthermore, it must handle the CPU-intensive iteration over truncation buffers during threshold updates, which causes the observed "Add-back" spikes in the trace.

Practical implications. A 30% CPU overhead is a favorable result for a stateful privacy mechanism. It indicates that production Flink clusters can be made differentially private without requiring a massive horizontal scale-out. Furthermore, the $O(\log t)$ complexity of the Binary Tree Mechanism ensures that this overhead remains stable regardless of the stream's duration, making the system suitable for long-term real-time analytics.

4.3.2 Processing Throughput

In this section, we evaluate the system's ability to handle high-volume data by identifying the maximum sustainable throughput for both the exact and DP pipelines.

Experimental Parameters

- **Executions:** 1 Warmup and 3 Runs.
- **Privacy Parameters:** $\varepsilon = 1$, $\beta = 0.1$, and $\theta = 1.0$.
- **Kafka Producer Rate:** {500k, 750k, 1M, 1.5M, 2M, 3M, 4M} events/s

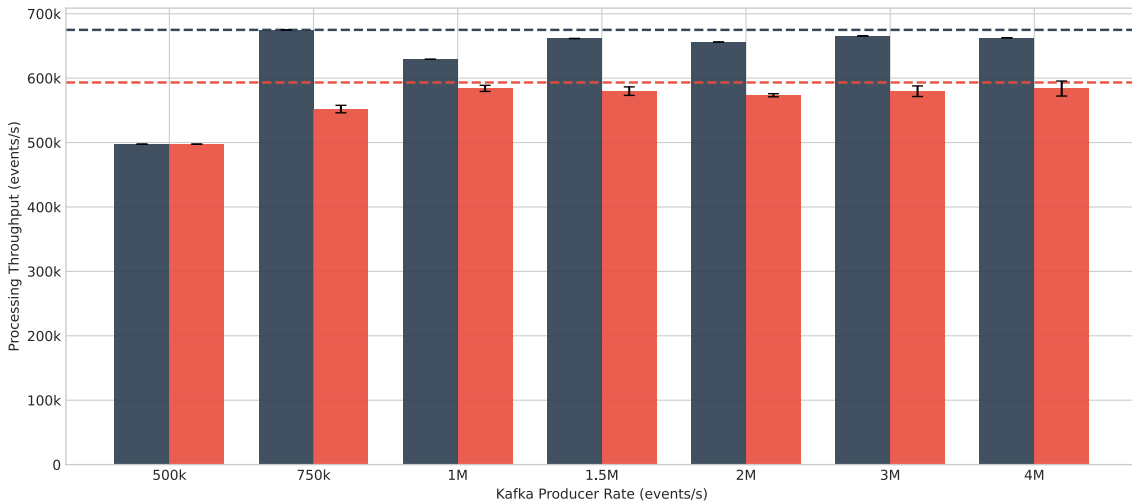


Figure 4.8: Mean processing throughput of the exact (dark blue) and DP (red) pipelines on the Netflix stream ($T \approx 100M$ events) as a function of Kafka producer rate. Dashed lines indicate the global maximum observed for each configuration.

Methodology. To determine the capacity ceiling of our implementation, we performed a parameter sweep of the Kafka production rate, ranging from 500,000 to 4,000,000 events/s. This methodology allows us to observe the transition from a *stable* regime (where Flink keeps pace with the producer) to a *saturated* regime (where the Flink operator becomes the bottleneck).

Results and Saturation Analysis. The results in Figure 4.8 reveal a clear performance ceiling for both configurations.

At a rate of 500,000 events/s, both the exact and DP pipelines process data at the full injection speed with zero lag. However, as the producer rate exceeds 750,000 events/s, the pipelines reach their respective saturation points. The **exact pipeline** (dark navy) plateaus at a maximum throughput of approximately **675,000** events/s.

The **DP pipeline** (red) plateaus at a lower ceiling. As highlighted by the dashed reference line in Figure 4.8, the maximum sustainable throughput for the DP implementation is approximately **595,000** events/s. The throughput remains remarkably stable across the higher production rates, with very narrow error bars indicating consistent performance even under heavy saturation.

The Privacy Throughput Tax. The absolute throughput penalty introduced by the DP mechanism is approximately 80,000 events/s, representing a relative decrease of **12%** in processing capacity. It is noteworthy that this 12% throughput loss is significantly lower than the 30% CPU overhead reported in Section 4.3.1. This indicates that while the DP logic, specifically the `MapState` management for user-level sensitivity (Section 3.2.2), is computationally more expensive, the system is efficient enough to absorb much of that cost without a proportional collapse in event-processing speed.

Implications for Deployment. The measured ceiling of $\approx 595,000$ events/s demonstrates that the DP implementation is highly performant. At this speed, the entire Netflix dataset of 100M records is processed in under 3 minutes.

4.3.3 Infrastructure Processing Latency

While throughput establishes the system’s capacity, latency defines its responsiveness. In this section, we evaluate the temporal cost of privacy by examining the distribution of event processing times. We adopt a top-down analysis: first observing the *Pipe Latency*, which represents the total journey of a record through the infrastructure, and then "zooming in" on the *Operator Latency* to isolate the specific computational tax of the DP logic.

Experimental Parameters.

- **Executions:** 1 Warmup and 3 Runs.
- **Privacy Parameters:** $\varepsilon = 1$, $\beta = 0.1$, and $\theta = 1.0$.
- **Kafka Producer Rate:** {1k, 2k, 5M, 10M, 50M} events/s

Instrumentation and Measurement Logic. To obtain high-precision measurements without distorting system behavior, we instrumented the pipeline using a side-output mechanism. Every event is tracked as it moves through the infrastructure using two primary endpoints:

1. **Broker Arrival** (T_{broker}): Extracted from the Kafka `ConsumerRecord` metadata. The broker is configured with `LogAppendTime`, ensuring this timestamp represents the exact moment the record was durably stored in the queue.
2. **Processing Completion** ($T_{completion}$): Recorded at the exit of the `processElement` method, representing the moment a privacy-protected result is ready.

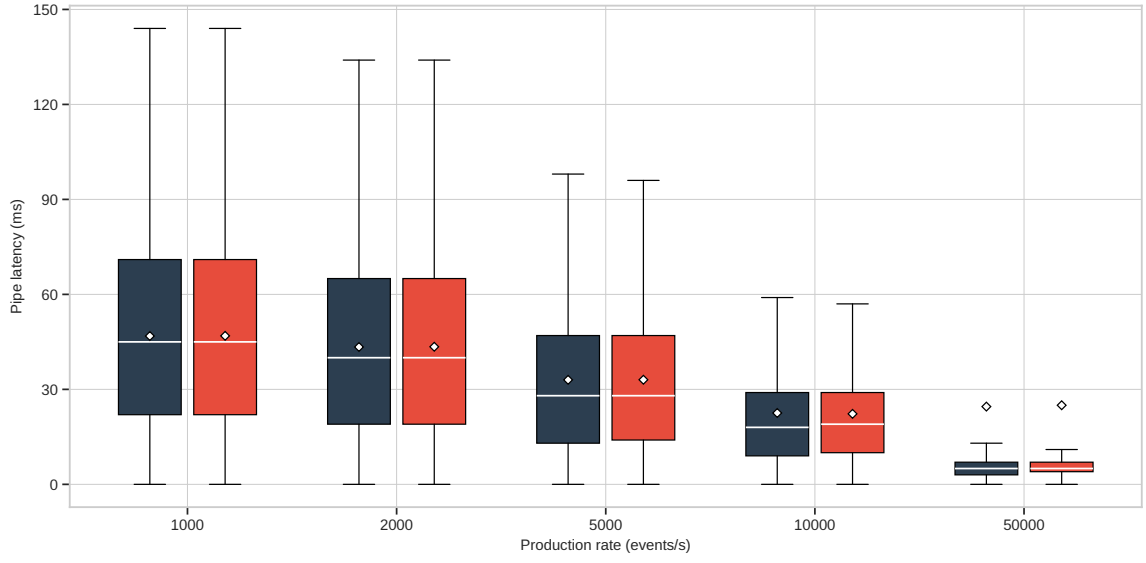
Based on these reference points, *Pipe Latency* is calculated as the duration $T_{completion} - T_{broker}$, representing the total time a record spends within the system’s boundaries. In contrast, *Operator Latency* is computed by taking the difference between two `System.nanoTime()` readings recorded at the immediate entry and exit of the `processElement` method. This measurement isolates the strictly computational time required for state lookups, SVT checks, and Binary Tree updates. These metrics are emitted for every event via a Flink side-output to a dedicated sink for post-hoc statistical analysis.

Pipe Latency: The Macro View. Figure 4.9a reports the pipe latency in milliseconds, representing the delay a data consumer would experience in a production environment. At this scale, the exact and DP pipelines appear virtually identical. Both configurations exhibit a decreasing trend in latency as the production rate increases, dropping from a median of ≈ 45 ms at 1,000 events/s to under 10 ms at 50,000 events/s. This decrease is a result of higher event volumes reducing the relative impact of "batching" timeouts and buffer-filling delays in the Kafka and Flink network stacks. Crucially, the "Privacy Tax" is invisible at the millisecond scale, suggesting that the DP logic does not impact the perceived freshness of the stream.

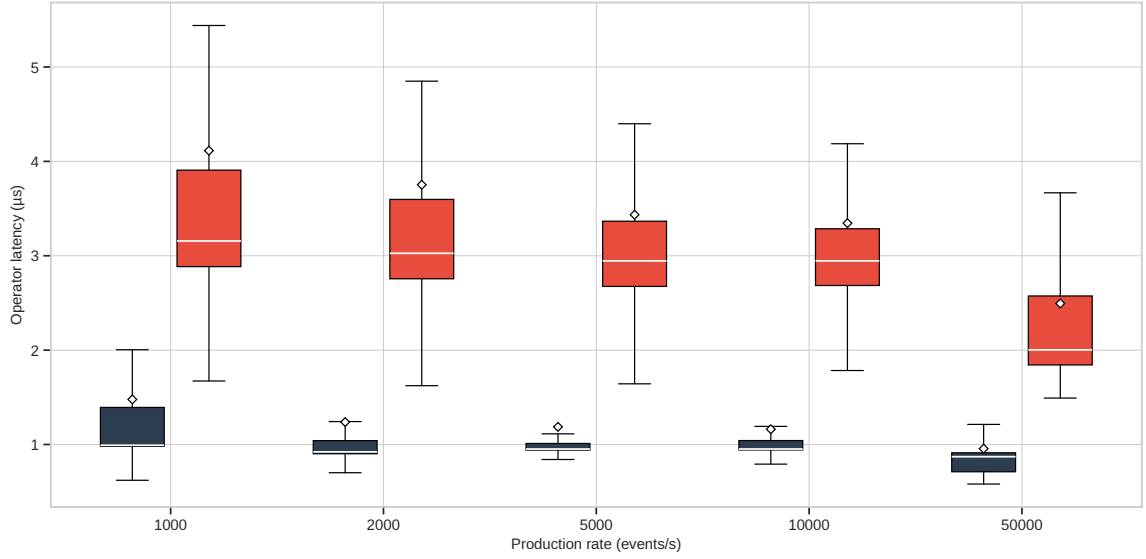
Operator Latency: The Micro Zoom. To understand why the overhead is invisible in the macro view, we examine the Operator Latency in Figure 4.9b. By measuring strictly the time spent inside the `processElement` method, we observe a clear but minute difference. The exact baseline (navy) is extremely efficient, with a median processing time of ≈ 1 μ s. The DP mechanism (red) requires ≈ 3 μ s per event.

This 200% relative increase is the direct cost of managing the complex state required for user-level DP, such as the `MapState` user-counters and the Binary Tree node updates (Section 3.2.2). Beyond the mean increase, the DP pipeline also exhibits higher variance (larger boxes and whiskers), reflecting the non-uniform cost of SVT evaluations and periodic "Add-back" buffer scans. However, because this tax is measured in *microseconds*, it is effectively "lost in the noise" of the millisecond-scale latencies caused by I/O and network brokerage shown in Figure 4.9a. We also observe that at 50,000 events/s, the DP operator latency decreases slightly, which we attribute to the JVM’s JIT compiler optimizing the hot code paths of the Laplace sampling logic under sustained load.

Summary of Findings. The combination of these results demonstrates that the implementation is highly optimized and production-ready. While the DP logic is computationally



(a) Pipe Latency (ms).



(b) Micro-level Operator Latency (μs) isolation.

Figure 4.9: Latency distribution for the **exact baseline (dark navy)** and the **DP mechanism (red)** across varying production rates. Diamonds indicate the mean value. Note the difference in scale between milliseconds (a) and microseconds (b).

more expensive than a simple counter, the absolute cost per event is so small ($< 5 \mu s$) that it does not affect the end-to-end performance of the streaming application. This confirms that user-level differential privacy can be integrated into high-speed Flink pipelines with virtually no impact on system responsiveness.

4.4 Discussion

The experimental results presented in Sections 4.2 and 4.3 collectively reveal a coherent and nuanced picture of the Dong *et al.* mechanism's behaviour in practice. In this section, we synthesize these findings, reflect on their implications for practitioners, and situate them within the broader landscape of differentially private stream processing.

The Paradox of Instance Optimality. A central finding of this evaluation is that the theoretical "instance-optimality" of the Dong *et al.* mechanism does not guarantee practical utility in adverse regimes. While the mechanism tracks well-conditioned streams like Netflix with high fidelity, it fails when the unique user base is small relative to the activity of heavy hitters. This reveals a significant gap in the current state of the art: the algorithm is most robust where privacy is easiest to maintain, but it fails precisely where protection is most needed to mask dominant individuals. Our discovery of *Truncation Lock* (Section 3.5) suggests that practitioners cannot rely on the privacy budget ϵ to "buy" accuracy. As formulated in our mathematical analysis (Section 3.6), if the voter supply is insufficient to satisfy the SVT's algorithmic demand, utility collapse is inevitable regardless of the budget's permissiveness.

The Dominance of Truncation Bias. In favourable regimes, the error is dominated by the statistical noise of the Laplace mechanism. However, as the stream moves toward Truncation Lock, the error becomes *bias-dominated*. This is clearly demonstrated by the LastFM results, where the relative error drifts toward 100%. This behaviour is explained by the *Concentration Ratio* (Section 3.6.2), which captures how much of the stream's total mass is generated by a few users who exceed the stalled threshold. This result proves that for highly skewed datasets, the primary source of error is not the "privacy noise" itself, but the physical removal of records required to bound sensitivity to a level that the SVT can support.

Temporal Dynamics as a Hidden Risk Factor. The synthetic experiments performed in Section 4.2.2 (Temporal Distribution and User Arrival Dynamics) reveal a risk factor that is invisible in global dataset statistics: the temporal ordering of user activity. We have proven that identical global statistics can yield completely different utility outcomes based on user arrival patterns. When heavy-hitter events arrive before a sufficient population of "voters" exists to trigger the SVT (Lemma 3.1: The Temporal Cardinality Race), the mechanism is forced to discard valuable data. Because our pre-execution prediction algorithm (Section 3.6.4) is based on global frequency counts, it provides an optimistic bound that may not hold in "bursty" real-world deployments. Developing an arrival-sensitive prediction model remains a vital direction for future research.

The Systemic Viability of Stateful DP. From a systems engineering perspective, the performance evaluation confirms that user-level DP is viable for high-speed production environments. The overhead is primarily characterized as a "CPU tax" rather than a "latency penalty." The microsecond-scale cost of the processing logic is effectively eclipsed by the millisecond-scale delays of the infrastructure transport layer. This confirms that the complex state management required for the coordinator (Section 3.2.2), including the use

of Flink’s `MapState` to track user contributions, is efficient enough to be integrated into existing pipelines without degrading responsiveness. Furthermore, the $O(\log t)$ complexity of the dyadic forest structure (Section 2.7.1) ensures that this performance remains stable over infinite horizons.

Significance for the Field. Taken together, our results demonstrate that user-level differential privacy in Apache Flink is practically viable for well-conditioned streams at an acceptable performance cost. Simultaneously, they expose a mathematically inevitable failure mode, *Truncation Lock*, that the existing theoretical literature does not explicitly warn against. By providing both a production-ready implementation and a formal characterization of its limits, this thesis advances the state of the art on two fronts: existence proof and failure proof. Both are essential for the responsible deployment of privacy-preserving systems in production.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The digitalization of our society has placed real-time analytics and individual privacy in a fundamental tension. While stream processing systems provide extraordinary insights at scale, they simultaneously enable the inference of sensitive personal data. This thesis has resolved this tension by integrating mathematically rigorous privacy guarantees into the Apache Flink ecosystem, demonstrating that robust privacy is compatible with high-performance stream processing.

System Implementation. We successfully implemented the state-of-the-art continual observation mechanism by Dong et al.[36] within Apache Flink[11] and Kafka[3]. By translating abstract theory into a stateful, fault-tolerant pipeline, we delivered a module capable of processing $\approx 595,000$ events/s with a modest CPU overhead of 30%. Crucially, the latency impact remains at the microsecond scale ($< 5\mu s$ per event), making user-level Differential Privacy (DP) viable for high-speed production environments without degrading system responsiveness.

The Truncation Lock Phenomenon. A primary theoretical contribution of this work is the identification of *Truncation Lock* (see Section 3.4). We have shown that the very "discount terms" 3.2 designed to protect the privacy budget in infinite streams can inadvertently starve the mechanism of the "voter supply" needed to update its accuracy. In environments with low user cardinality or high contribution skew, the system enters a deterministic stalemate where the truncation threshold stops evolving. This causes the algorithm to "go blind" to new data from heavy hitters, regardless of how much privacy budget is available.

Empirical Impact. We demonstrated empirically (Section 4.2.1) that Truncation Lock is not merely a theoretical edge case; it manifests on real-world datasets from diverse domains such as music streaming and location check-ins. When this lock occurs, the mechanism's utility collapses, with relative errors approaching 100% regardless of the privacy budget's permissiveness. Because this failure is "silent"—emitting plausible-looking but mathematically inaccurate numbers—it represents a significant hidden risk for the responsible deployment of privacy-preserving systems.

Predictive Modeling. To make these insights actionable for practitioners, we derived a lightweight pre-execution prediction algorithm (Section 3.6.4). By analyzing the user contribution distribution of a dataset, the algorithm computes the halting epoch k_{halt} and the predicted relative error in $O(|U| \log |U|)$ time. Empirical validation on synthetic datasets (Section 4.2.2) confirms that this mathematical model accurately forecasts algorithmic failure, providing analysts with a quantitative "go/no-go" tool to assess utility before committing to a resource-intensive deployment.

Bridging Theory and Practice. The broader lesson of this work is that mathematical "instance-optimality" does not guarantee practical utility in real-world distributed systems. Theoretical bounds often suppress hidden costs, such as polylogarithmic factors, that act as severe bottlenecks in production. To illustrate this quantitatively, while the mechanism theoretically adapts to any user behavior, our evaluation revealed that on skewed datasets, the dynamic threshold can permanently stall at a mere 64 events. Consequently, for a heavy-hitter contributing over 183,000 events, the system silently discards $> 99.9\%$ of their data, driving the relative error to 100% despite the formal utility proofs. This stark contrast highlights a critical requirement for the field: for DP to be truly robust, mechanisms must explicitly account for the bursty, heavily skewed nature of real-world streams. By delivering both a high-throughput implementation and a quantitative proof of its limits, this thesis bridges the gap between abstract privacy guarantees and the realities of responsible data analytics.

5.2 Future Work

The work presented in this thesis opens several directions for future research, spanning the theoretical foundations of differentially private streaming, the engineering of privacy-aware distributed systems, and the practical adoption of formal privacy guarantees in production infrastructure.

5.2.1 Extending the Implementation to the Full Query Portfolio

Mechanism Expansion. This thesis focuses exclusively on the continual count query F_{cnt} , corresponding to Algorithm 5 in Dong et al. [36]. The original paper provides mechanisms for a broader class of functions, including the sum query F_{sum} , the histogram query F_{ht} , the k -selection query F_{k-sel} , and the distinct count query F_0 . A natural extension of this work would be to implement and evaluate these families within the same Flink module, delivering a comprehensive DP library for stream analytics usable by practitioners without deep expertise in privacy theory.

Empirical Optimality. Dong et al. prove a formal separation result: for distinct count, histogram, and k -selection, down-neighborhood optimality is provably unachievable in the dynamic setting, even though it is achievable statically. Implementing these functions to empirically characterize how close to optimal they remain in practice, and under what data distributions they degrade the most, would be a valuable complement to existing theoretical impossibility results.

5.2.2 Mechanisms Robust to Truncation Lock

Alternative Estimation. The *Truncation Lock* phenomenon identified in Chapter 3 represents the primary practical limitation of the current mechanism. One direction for future work is to replace the voting-based SVT estimator for $\kappa(D_t)$ with a mechanism that does not require a minimum "voter supply." For example, a private quantile estimator could track the top- p percentile of the user contribution distribution. Such an estimator would naturally adapt to populations where the contribution skew is extreme but the total number of users is small.

Adaptive Initialization. A second direction involves dataset-adaptive initialization. Rather than starting the truncation threshold at a fixed $\tau_0 = 64$ regardless of stream characteristics, a brief private survey of the initial user distribution could set τ_0 at a level matched to the expected voter supply, reducing the epochs required to reach true hardness and extending the mechanism's operational horizon.

5.2.3 Distributed and Parallelized Architecture

Horizontal Scaling. The current implementation routes all events through a single logical partition, creating a throughput ceiling. For queries that decompose naturally, such as per-key aggregations or histogram buckets, a distributed implementation using parallel composition (See 2.5.1) is theoretically sound. Future work should explore how the privacy accounting model can be extended to multi-partition Flink deployments, including the design of a distributed SVT that tracks the global $\kappa(D_t)$ estimate without centralizing all state through a single coordinator.

5.2.4 Windowed Differentially Private Aggregations

Temporal Boundaries. Integrating the Dong et al. mechanism with Flink's native tumbling, sliding, and session window operators (See 2.1.3 and figure A.1) requires careful handling of the privacy budget across window boundaries. While the w -event DP model [62] provides a foundation for windowed privacy, it protects only a sliding window of events. Developing a variant that maintains full user-level protection while exploiting the finite window horizon to tighten noise calibration is an open research problem of significant practical importance.

5.2.5 Privacy Budget Management Across Concurrent Queries

Aggregate Accounting. In a real deployment, an operator may run multiple concurrent DP queries on the same stream. Managing the aggregate privacy cost requires a principled budget accounting layer. Future work should design a Flink-native budget manager that tracks the cumulative ϵ consumed by all active DP operators, enforces a configurable total ceiling, and allocates budgets based on query priority. This would be a major step toward making DP stream processing accessible for industrial use-cases.

5.2.6 The Fully-Dynamic Setting: Handling Deletions

Retraction Support. The current implementation handles only event insertions. In a fully-dynamic setting, events can be retracted (e.g., fraudulent transaction reversals). Dong et al. sketch a mechanism for this case by separating insertions and deletions into independent streams. Implementing this extension within Flink and studying whether *Truncation Lock* has an analogue in the deletion setting, where the population changes retroactively, is a vital direction for future research.

Bibliography

- [1] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, A. H. Byers, M. Chui, B. Brown, J. Bughin *et al.*, “Big data: The next frontier for innovation, competition, and productivity,” 2011.
- [2] A. Gandomi and M. Haider, “Beyond the hype: Big data concepts, methods, and analytics,” *International journal of information management*, vol. 35, no. 2, pp. 137–144, 2015.
- [3] J. Kreps, N. Narkhede, and J. Rao, “Kafka: A Distributed Messaging System for Log Processing,” vol. 11, no. 2011, pp. 1–7, 2011.
- [4] M. Stonebraker, U. Çetintemel, and S. Zdonik, “The 8 requirements of real-time stream processing,” *ACM Sigmod Record*, vol. 34, no. 4, pp. 42–47, 2005.
- [5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, “The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.
- [6] O. Kwon, N. Lee, and B. Shin, “Data quality management, data usage experience and acquisition intention of big data analytics,” *International journal of information management*, vol. 34, no. 3, pp. 387–394, 2014.
- [7] N. Kshetri, “Big data’s impact on privacy, security and consumer welfare,” *Telecommunications Policy*, vol. 38, no. 11, pp. 1134–1145, 2014.
- [8] D. Boyd and K. Crawford, “Critical questions for big data: Provocations for a cultural, technological, and scholarly phenomenon,” *Information, communication & society*, vol. 15, no. 5, pp. 662–679, 2012.
- [9] F. Provost and T. Fawcett, “Data science and its relationship to big data and data-driven decision making,” *Big data*, vol. 1, no. 1, pp. 51–59, 2013.
- [10] A. Bifet, R. Gavalda, G. Holmes, and B. Pfahringer, *Machine learning for data streams: with practical examples in MOA*. MIT press, 2023.
- [11] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink: Stream and batch processing in a single engine,” *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

- [12] Apache Software Foundation, “Apache flink: Stateful computations over data streams,” <https://flink.apache.org/>, accessed: 2026-03-01.
- [13] —, “Apache spark: Unified engine for large-scale data analytics,” <https://spark.apache.org/>, accessed: 2026-03-01.
- [14] A. Acquisti, L. Brandimarte, and G. Loewenstein, “Privacy and human behavior in the age of information,” *Science*, vol. 347, no. 6221, pp. 509–514, 2015.
- [15] S. Zuboff, “Big other: Surveillance Capitalism and the Prospects of an Information Civilization,” *Journal of Information Technology*, vol. 30, no. 1, pp. 75–89, Mar. 2015.
- [16] O. Tene and J. Polonetsky, “Big data for all: Privacy and user control in the age of analytics,” *Nw. J. Tech. & Intell. Prop.*, vol. 11, p. 239, 2012.
- [17] J. Isaak and M. J. Hanna, “User data privacy: Facebook, cambridge analytica, and privacy protection,” *Computer*, vol. 51, no. 8, pp. 56–59, 2018.
- [18] N. Kshetri, “Big data s impact on privacy, security and consumer welfare,” *Telecommunications Policy*, vol. 38, no. 11, pp. 1134–1145, 2014.
- [19] L. Sweeney, “k-anonymity: A model for protecting privacy,” *International journal of uncertainty, fuzziness and knowledge-based systems*, vol. 10, no. 05, pp. 557–570, 2002.
- [20] —, “Simple demographics often identify people uniquely,” *Health (San Francisco)*, vol. 671, no. 2000, pp. 1–34, 2000.
- [21] A. Narayanan and V. Shmatikov, “Robust de-anonymization of large sparse datasets,” in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 111–125.
- [22] I. Dinur and K. Nissim, “Revealing information while preserving privacy,” in *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2003, pp. 202–210.
- [23] S. R. Ganta, S. P. Kasiviswanathan, and A. Smith, “Composition attacks and auxiliary information in data privacy,” in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2008, pp. 265–273.
- [24] C. Dwork, “Differential privacy,” in *International Colloquium on Automata, Languages, and Programming (ICALP)*. Springer, 2006, pp. 1–12.
- [25] C. Dwork and A. Roth, “The algorithmic foundations of differential privacy,” *Foundations and trends® in theoretical computer science*, vol. 9, no. 3-4, pp. 211–487, 2014.
- [26] R. Cummings, D. Desfontaines, D. Evans, R. Geambasu, Y. Huang, M. Jagielski, P. Kairouz, G. Kamath, S. Oh, O. Ohrimenko *et al.*, “Advancing differential privacy: Where we are now and future directions for real-world deployment,” *arXiv preprint arXiv:2304.06929*, 2023.

- [27] Ú. Erlingsson, V. Pihur, and A. Korolova, “Rappor: Randomized aggregatable privacy-preserving ordinal response,” in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 1054–1067.
- [28] J. M. Abowd, “The us census bureau adopts differential privacy,” in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 2867–2867.
- [29] L. Fan and L. Xiong, “An adaptive approach to real-time aggregate monitoring with differential privacy,” *IEEE Transactions on knowledge and data engineering*, vol. 26, no. 9, pp. 2094–2106, 2013.
- [30] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum, “Differential privacy under continual observation,” in *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, ser. STOC ’10. Association for Computing Machinery, pp. 715–724. [Online]. Available: <https://dl.acm.org/doi/10.1145/1806689.1806787>
- [31] T.-H. H. Chan, E. Shi, and D. Song, “Private and Continual Release of Statistics,” vol. 14, no. 3, pp. 1–24. [Online]. Available: <https://dl.acm.org/doi/10.1145/2043621.2043626>
- [32] B. Zhang, V. Doroshenko, P. Kairouz, T. Steinke, A. Thakurta, Z. Ma, E. Cohen, H. Apte, and J. Spacek, “Differentially Private Stream Processing at Scale,” vol. 17, no. 12, pp. 4145–4158. [Online]. Available: <https://dl.acm.org/doi/10.14778/3685800.3685833>
- [33] S. Denisov, H. B. McMahan, J. Rush, A. Smith, and A. Guha Thakurta, “Improved differential privacy for sgd via optimal private linear operators on adaptive streams,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 5910–5924, 2022.
- [34] P. Covington, J. Adams, and E. Sargin, “Deep Neural Networks for YouTube Recommendations,” in *Proceedings of the 10th ACM Conference on Recommender Systems*, ser. RecSys ’16. New York, NY, USA: Association for Computing Machinery, Sep. 2016, pp. 191–198.
- [35] C. A. Gomez-Urbe and N. Hunt, “The netflix recommender system: Algorithms, business value, and innovation,” *ACM Trans. Manage. Inf. Syst.*, vol. 6, no. 4, Dec. 2016. [Online]. Available: <https://doi.org/10.1145/2843948>
- [36] W. Dong, Q. Luo, and K. Yi, “Continual Observation under User-level Differential Privacy,” in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2190–2207. [Online]. Available: <https://ieeexplore.ieee.org/document/10179466/>
- [37] T. Akidau, E. Begoli, S. Chernyak, F. Hueske, K. Knight, K. Knowles, D. Mills, and D. Sotolongo, “Watermarks in stream processing systems: Semantics and comparative analysis of Apache Flink and Google cloud dataflow,” vol. 14, no. 12, pp. 3135–3147. [Online]. Available: <https://dl.acm.org/doi/10.14778/3476311.3476389>
- [38] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, “State management in apache flink®: consistent stateful distributed stream processing,” *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017.

- [39] K. M. Chandy and L. Lamport, “Distributed snapshots: Determining global states of distributed systems,” *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, Feb. 1985.
- [40] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, “Optimizing space amplification in rocksdb,” in *CIDR*, vol. 3, 2017, p. 3.
- [41] The RocksDB Team, “Rocksdb: A persistent key-value store for fast storage,” <https://rocksdb.org/>.
- [42] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkatasubramanian, “l-diversity: Privacy beyond k-anonymity,” *Acm transactions on knowledge discovery from data (tkdd)*, vol. 1, no. 1, pp. 3–es, 2007.
- [43] N. Li, T. Li, and S. Venkatasubramanian, “t-closeness: Privacy beyond k-anonymity and l-diversity,” in *2007 IEEE 23rd international conference on data engineering*. IEEE, 2006, pp. 106–115.
- [44] C. Dwork, F. McSherry, K. Nissim, and A. Smith, “Calibrating noise to sensitivity in private data analysis,” in *Theory of Cryptography Conference (TCC)*. Springer, 2006, pp. 265–284.
- [45] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor, “Our data, ourselves: Privacy via distributed noise generation,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2006, pp. 486–503.
- [46] D. Kifer and A. Machanavajjhala, “No free lunch in data privacy,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. ACM, 2011, pp. 193–204.
- [47] J. P. Near, D. Darais, N. Lefkovitz, G. S. Howarth *et al.*, *Guidelines for evaluating differential privacy guarantees*. US Department of Commerce, National Institute of Standards and Technology, 2025.
- [48] D. Desfontaines, “Differential privacy in (a bit) more detail,” <https://desfontain.es/blog/differential-privacy-in-more-detail.html>, 08 2018, ted is writing things (personal blog).
- [49] “Learning with Privacy at Scale,” <https://machinelearning.apple.com/research/learning-with-privacy-at-scale>.
- [50] R. Rogers, A. R. Cardoso, K. Mancuhan, A. Kaura, N. Gahlawat, N. Jain, P. Ko, and P. Ahammad, “A members first approach to enabling linkedin’s labor market insights at scale,” *arXiv preprint arXiv:2010.13981*, 2020.
- [51] J. M. Abowd, R. Ashmead, R. Cumings-Menon, S. Garfinkel, M. Heineck, C. Heiss, R. Johns, D. Kifer, P. Leclerc, A. Machanavajjhala *et al.*, “The 2020 census disclosure avoidance system topdown algorithm,” *Harvard Data Science Review*, vol. 2, pp. 1–72, 2022.

- [52] D. Desfontaines, “A list of real-world uses of differential privacy,” <https://desfontaines.blog/real-world-differential-privacy.html>, 10 2021, ted is writing things (personal blog).
- [53] C. Dwork, G. N. Rothblum, and S. Vadhan, “Boosting and Differential Privacy,” in *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pp. 51–60. [Online]. Available: <https://ieeexplore.ieee.org/document/5670947/>
- [54] J. A. Calandrino, A. Kilzer, A. Narayanan, E. W. Felten, and V. Shmatikov, ““You Might Also Like:” Privacy Risks of Collaborative Filtering,” in *2011 IEEE Symposium on Security and Privacy*, pp. 231–246. [Online]. Available: <https://ieeexplore.ieee.org/document/5958032/>
- [55] C. Dwork, M. Naor, O. Reingold, and G. N. Rothblum, “Pure Differential Privacy for Rectangle Queries via Private Partitions,” in *Advances in Cryptology – ASIACRYPT 2015*, T. Iwata and J. H. Cheon, Eds. Springer Berlin Heidelberg, vol. 9453, pp. 735–751. [Online]. Available: http://link.springer.com/10.1007/978-3-662-48800-3_30
- [56] Y. Chen, A. Machanavajjhala, M. Hay, and G. Miklau, “PeGaSus: Data-Adaptive Differentially Private Stream Processing,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, pp. 1375–1388. [Online]. Available: <https://dl.acm.org/doi/10.1145/3133956.3134102>
- [57] H. Fichtenberger, M. Henzinger, and J. Upadhyay. Constant matters: Fine-grained Complexity of Differentially Private Continual Observation. [Online]. Available: <http://arxiv.org/abs/2202.11205>
- [58] M. Henzinger, J. Upadhyay, and S. Upadhyay. Almost Tight Error Bounds on Differentially Private Continual Counting. [Online]. Available: <http://arxiv.org/abs/2211.05006>
- [59] A. Rivera Cardoso and R. Rogers, “Differentially private histograms under continual observation: Streaming selection into the unknown,” in *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Camps-Valls, F. J. R. Ruiz, and I. Valera, Eds., vol. 151. PMLR, 28–30 Mar 2022, pp. 2397–2419. [Online]. Available: <https://proceedings.mlr.press/v151/rivera-cardoso22a.html>
- [60] F. D. McSherry, “Privacy integrated queries: An extensible platform for privacy-preserving data analysis.”
- [61] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, “Airavat: Security and Privacy for MapReduce.”
- [62] G. Kellaris, S. Papadopoulos, X. Xiao, and D. Papadias, “Differentially private event sequences over infinite streams,” vol. 7, no. 12, pp. 1155–1166. [Online]. Available: <https://dl.acm.org/doi/10.14778/2732977.2732989>
- [63] A. Xiong, T. Wang, N. Li, and S. Jha, “Towards Effective Differential Privacy Communication for Users’ Data Sharing Decision and Comprehension,” in *2020*

- IEEE Symposium on Security and Privacy (SP)*, pp. 392–410. [Online]. Available: <https://ieeexplore.ieee.org/document/9152658/>
- [64] X. Ren, L. Shi, W. Yu, S. Yang, C. Zhao, and Z. Xu, “LDP-IDS: Local Differential Privacy for Infinite Data Streams,” in *Proceedings of the 2022 International Conference on Management of Data*. ACM, pp. 1064–1077. [Online]. Available: <https://dl.acm.org/doi/10.1145/3514221.3526190>
 - [65] M. Joseph, A. Roth, J. Ullman, and B. Waggoner, “Local Differential Privacy for Evolving Data,” vol. 10, no. 1. [Online]. Available: <https://journalprivacyconfidentiality.org/index.php/jpc/article/view/718>
 - [66] E. Bao, Y. Yang, X. Xiao, and B. Ding, “CGM: An enhanced mechanism for streaming data collection with local differential privacy,” vol. 14, no. 11, pp. 2258–2270. [Online]. Available: <https://dl.acm.org/doi/10.14778/3476249.3476277>
 - [67] Z. Huang, Y. Liang, and K. Yi, “Instance-optimal mean estimation under differential privacy,” *Advances in Neural Information Processing Systems*, vol. 34, pp. 25 993–26 004, 2021.
 - [68] W. Dong and K. Yi, “Universal Private Estimators,” in *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, pp. 195–206. [Online]. Available: <https://dl.acm.org/doi/10.1145/3584372.3588669>
 - [69] J. Fang, W. Dong, and K. Yi, “Shifted Inverse: A General Mechanism for Monotonic Functions under User Differential Privacy,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, pp. 1009–1022. [Online]. Available: <https://dl.acm.org/doi/10.1145/3548606.3560567>
 - [70] C. Dwork, M. Naor, O. Reingold, G. N. Rothblum, and S. Vadhan, “On the complexity of differentially private data release: Efficient algorithms and hardness results,” in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. ACM, pp. 381–390. [Online]. Available: <https://dl.acm.org/doi/10.1145/1536414.1536467>
 - [71] J. Ni, J. Li, and J. McAuley, “Justifying recommendations using distantly-labeled reviews and fine-grained aspects,” in *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, 2019, pp. 188–197.
 - [72] AOL Inc., “AOL search query log dataset,” https://archive.org/download/AOL_search_data_leak_2006, 2006, released August 2006.
 - [73] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” <http://snap.stanford.edu/data>, Jun. 2014.
 - [74] J. Tang, X. Hu, H. Gao, and H. Liu, “Exploiting local and global social context for recommendation,” in *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, vol. 13, 2013, pp. 2712–2718. [Online]. Available: <https://www.cse.msu.edu/~tangjili/datasetcode/truststudy.htm>

- [75] D. Yang, D. Zhang, V. W. Zheng, and Z. Yu, “Modeling user activity preference by leveraging user spatial temporal characteristics in LBSNs,” in *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 45, no. 1, 2015, pp. 129–142. [Online]. Available: <https://sites.google.com/site/yangdingqi/home/foursquare-dataset>
- [76] M. Wan and J. McAuley, “Item recommendation on monotonic behavior chains,” in *Proceedings of the 12th ACM Conference on Recommender Systems (RecSys)*, 2018. [Online]. Available: <https://cseweb.ucsd.edu/~jmcauley/datasets/goodreads.html>
- [77] M. Wan, R. Misra, N. Nakashole, and J. McAuley, “Fine-grained spoiler detection from large-scale review corpora,” in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019, pp. 2605–2610.
- [78] C. Gao, S. Li, W. Lei, J. Chen, B. Li, P. Jiang, X. He, J. Mao, and T.-S. Chua, “Kuairec: A fully-observed dataset and insights for evaluating recommender systems,” in *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 2022, pp. 540–550.
- [79] I. Cantador, P. Brusilovsky, and T. Kuflik, “Second workshop on information heterogeneity and fusion in recommender systems (HetRec),” in *Proceedings of the 5th ACM Conference on Recommender Systems (RecSys)*, 2011, pp. 387–388. [Online]. Available: <https://grouplens.org/datasets/hetrec-2011/>
- [80] F. M. Harper and J. A. Konstan, “The MovieLens datasets: History and context,” *ACM Transactions on Interactive Intelligent Systems*, vol. 5, no. 4, pp. 1–19, 2015. [Online]. Available: <https://grouplens.org/datasets/movielens/1m/>
- [81] S. Dooms, T. De Pessemier, and L. Martens, “MovieTweatings: A movie rating dataset collected from twitter,” in *Workshop on Crowdsourcing and Human Computation for Recommender Systems (CrowdRec) at ACM RecSys*, 2013, p. 43. [Online]. Available: <https://github.com/sidooms/MovieTweatings>
- [82] Netflix Inc., “Netflix prize dataset,” <https://www.kaggle.com/datasets/netflix-inc/netflix-prize-data>, 2009, originally released for the Netflix Prize competition (2006–2009).
- [83] Retail Rocket, “Retail rocket e-commerce dataset,” <https://www.kaggle.com/datasets/retailrocket/ecommerce-dataset>, 2016, accessed via Kaggle.
- [84] J. McAuley, C. Targett, Q. Shi, and A. van den Hengel, “Image-based recommendations on styles and substitutes,” in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2015, pp. 43–52. [Online]. Available: https://cseweb.ucsd.edu/~jmcauley/datasets.html#steam_data
- [85] Yelp Inc., “Yelp open dataset,” <https://business.yelp.com/data/resources/open-dataset/>, 2024, accessed: 2024.
- [86] G. K. Zipf, *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio books, 2016.

- [87] M. Bun and T. Steinke, “Concentrated differential privacy: Simplifications, extensions, and lower bounds,” in *Theory of Cryptography*, M. Hirt and A. Smith, Eds. Springer Berlin Heidelberg, pp. 635–658.

Appendix A

Appendix

A.1 Windowing

Windowing Models To compute finite aggregates over unbounded data, streams are often divided into manageable temporal chunks called *windows*[5]. These models provide a mechanism to bound the scope of stateful operations in a continuous stream [11]. Three principal types are standard, as shown in Figure A.1:

- **Tumbling windows** partition the stream into non-overlapping, fixed-size time intervals. Each event belongs to exactly one window.
- **Sliding windows** are defined by a fixed duration and a slide interval shorter than the window. Events may belong to several overlapping windows simultaneously.
- **Session windows** are data-driven. A new window opens with the first event and closes when a predefined period of inactivity (a gap) is detected. They have variable lengths and are defined entirely by user behavior.

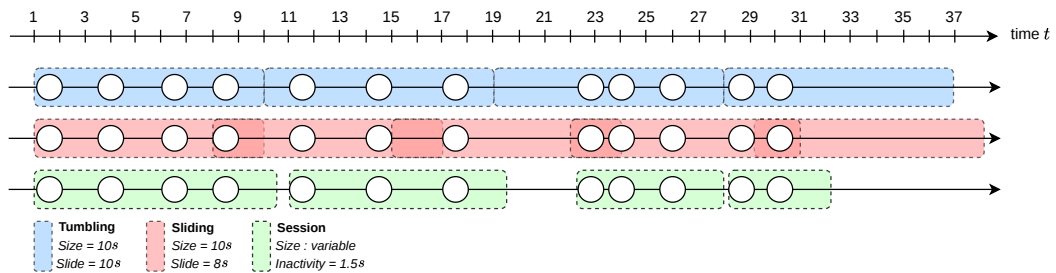


Figure A.1: Tumbling vs. Sliding vs. Session windows

The choice of windowing model dictates the analytical purpose and influences the final output. Consider a stream of user clicks on a website: A 1-hour *tumbling window* computes the total clicks strictly within fixed boundaries (e.g., 13:00 to 14:00), which is useful for rigid hourly reporting. In contrast, a 1-hour *sliding window* updated every 10 minutes yields a rolling sum of the past hour's activity, which is essential for real-time alerting and detecting sudden traffic spikes. Finally, a *session window* with a 15-minute inactivity gap

ignores fixed time boundaries altogether, instead grouping clicks into continuous bursts of engagement to output the total interactions per distinct user visit.

A.2 Privacy Before Differential Privacy

A.2.1 Classical Privacy Models

Before the emergence of differential privacy, privacy protection relied primarily on *syntactic anonymization*: transforming a dataset so that individual records are harder to link to specific people. However, as the following models demonstrate, these methods often failed to provide robust guarantees against determined adversaries.

De-identification and Re-identification Risks

De-identification removes or replaces explicit identifiers such as names or social security numbers. Sweeney [20] showed that this is insufficient. By using only three *quasi-identifiers*: ZIP code, birth date, and sex (none of which are explicit identifiers), Sweeney demonstrated that 87% of the U.S. population can be uniquely re-identified and linked back to supposedly anonymous medical records.

Name	Age	ZIP Code	Sex	Disease
****	28	13053	F	Flu
****	29	13068	F	Flu
****	35	13053	M	Cancer
****	36	13068	M	Flu
****	48	14853	F	Diabetes
****	50	14850	F	Cancer

Table A.1: Anonymized dataset with a sensitive attribute (red).

Table A.1, gives away sensitive information on anonymized individuals.

k -Anonymity

To address simple linkage attack, Sweeney [19] proposed k -anonymity. A dataset is considered k -anonymous if the quasi-identifier attributes of every record are indistinguishable from at least $k - 1$ other records. This state is typically achieved through two primary techniques, as illustrated in Table A.2.

Name	Age	ZIP Code	Sex	Disease
****	20–39	130**	F	Flu
****	20–39	130**	F	Flu
****	20–39	130**	M	Cancer
****	20–39	130**	M	Flu
****	40–59	148**	F	Diabetes
****	40–59	148**	F	Cancer

Table A.2: Dataset after applying 2-anonymity.

- **Generalization:** Replacing specific values with broader categories (e.g., changing an exact age of 26 to the range 20-30, or removing the last digits of a zip code).
- **Suppression:** Removing a certain attribute values or entire records from the dataset altogether to prevent them from forming a unique combination.

While these transformations prevent the identification of a specific individual within an "equivalence class", the model remains vulnerable to two primary structural attacks:

- **The Homogeneity Attack:** If all members of a k -anonymous group share the same sensitive attribute value (in Table A.2 line 1 and 2 have Flu and form a group), the privacy of that group is compromised despite being k -anonymous.
- **The Background Knowledge Attack:** An adversary can use external information to eliminate implausible candidates within a group, narrowing the possibilities to far fewer than k .

ℓ -Diversity

ℓ -Diversity [42] strengthens k -anonymity by requiring that each equivalence class contains at least ℓ “well-represented” values for a sensitive attribute. This model was designed to explicitly defend against both the homogeneity attack and the background knowledge attack explained above. By ensuring sufficient variety within a group, as illustrated in Table A.3, it prevents an adversary from reaching a certain conclusion even if they possess specific auxiliary information to rule out some candidates.

adversary from reaching a certain conclusion even if they possess specific auxiliary information to rule out some candidates.

Name	Age	ZIP Code	Sex	Disease
****	20–39	13***	*	Flu
****	20–39	13***	*	Flu
****	20–39	13***	*	Cancer
****	20–39	13***	*	Flu
****	40–59	14***	*	Diabetes
****	40–59	14***	*	Cancer

Table A.3: Dataset after enforcing 2-diversity.

However, ℓ -diversity remains vulnerable when the sensitive values in a group are semantically similar or when the distribution is highly skewed, as the mere presence of diversity does not prevent an observer from learning that a user likely falls into a specific category of sensitive traits.

t -Closeness

Li *et al.* [43] further strengthened these requirements with t -closeness. This model dictates that the distance between the distribution of a sensitive attribute within any equivalence class and its distribution in the overall table is no more than a threshold t .

In the extreme example of Table A.4, with t close to 0, we get a similar disease distribution as the original Table A.1, at the cost of having one single group, and hence very low utility.

Theoretically, in the most robust models, t -closeness often distorts data to such a degree that it destroys the analytical utility of the dataset, illustrating the fundamental trade-off between privacy and utility.

Name	Age	ZIP Code	Sex	Disease
****	20–59	1****	*	Flu
****	20–59	1****	*	Flu
****	20–59	1****	*	Cancer
****	20–59	1****	*	Flu
****	20–59	1****	*	Diabetes
****	20–59	1****	*	Cancer

Table A.4: Dataset after enforcing class-level Disease distribution matching the global distribution.

A.3 Differential Privacy Mechanisms

A.3.1 Gaussian Mechanism

The Gaussian mechanism is the primary alternative to the Laplace mechanism for protecting numeric queries. It achieves (ϵ, δ) -differential privacy (approximate DP) by adding noise sampled from a normal distribution [45]. Unlike the Laplace mechanism, which relies on the L_1 sensitivity, the Gaussian mechanism is calibrated to the L_2 sensitivity ($\Delta_2 f$) of the query.

The introduction of the δ parameter allows for a "failure probability" where the privacy guarantee might not hold, but in exchange, the Gaussian mechanism often provides better utility when composing a very large number of queries. However, as established in the previous section, the implementation in this thesis relies on the Dong *et al.* [36] framework. This framework requires the *pure ϵ -DP* property of the Laplace mechanism to maintain linear and exact group privacy bounds, a property that becomes mathematically much more complex and "wasteful" under the (ϵ, δ) guarantees of the Gaussian mechanism.

A.3.2 Zero-Concentrated Differential Privacy (zCDP) Mechanism

Several recent advancements in streaming DP, such as the work by Cardoso and Rogers [59], utilize **Zero-Concentrated Differential Privacy (zCDP)**. Introduced by Bun and Steinke [87], zCDP is a relaxation based on Rényi divergence that provides a tighter mathematical framework for analyzing Gaussian noise and query composition than standard (ϵ, δ) -DP.

While zCDP allows for a more refined error analysis in complex streaming sequences, its guarantees are typically converted back to (ϵ, δ) -DP for reporting. In the context of recent literature, zCDP has become the preferred tool for researchers aiming to optimize the constant factors or additive error in event-level mechanisms.

A.3.3 w -event Differential Privacy

The *w-event privacy* model [62] serves as a practical middle ground between the weak guarantees of event-level privacy and the strict demands of user-level privacy. Instead of protecting just a single isolated record, or attempting to hide a user's entire infinite history, it protects any sequence of events that occurs within a specific time frame. Intuitively, it guarantees that an adversary cannot determine whether a particular burst of user activity occurred within any window of w consecutive time steps. Formally, two data streams S and S' are considered *w-neighboring* if they are identical everywhere except for a span of at most w consecutive timestamps, where they are allowed to differ by any number of records.

This model defines a spectrum of protection based on the window size: setting $w = 1$ recovers standard *event-level privacy*, while setting w to the total stream length T is equivalent to *user-level privacy* for a finite horizon. The primary advantage of this model is utility, by allowing the privacy budget to be re-allocated across non-overlapping windows

of size w , the required noise is significantly reduced compared to full user-level mechanisms. However, w -event DP only provides “episodic” protection. It is mathematically insufficient for masking a user’s entire temporal footprint if their participation spans multiple windows or exceeds w consecutive steps.

A.4 Hyperparameter Sweep: ε vs. θ

As discussed in Section 4.1.3, the telescoping series decay rate θ dictates how the SVT privacy budget is allocated across infinite epochs. Figure A.2 illustrates a parameter sweep conducted on the highly skewed LastFM dataset, tracking the average relative error across varying combinations of ε and θ .

The results demonstrate that while the choice of θ can marginally impact utility at higher privacy budgets, $\theta = 1.0$ remains a highly robust default choice across the entire spectrum.

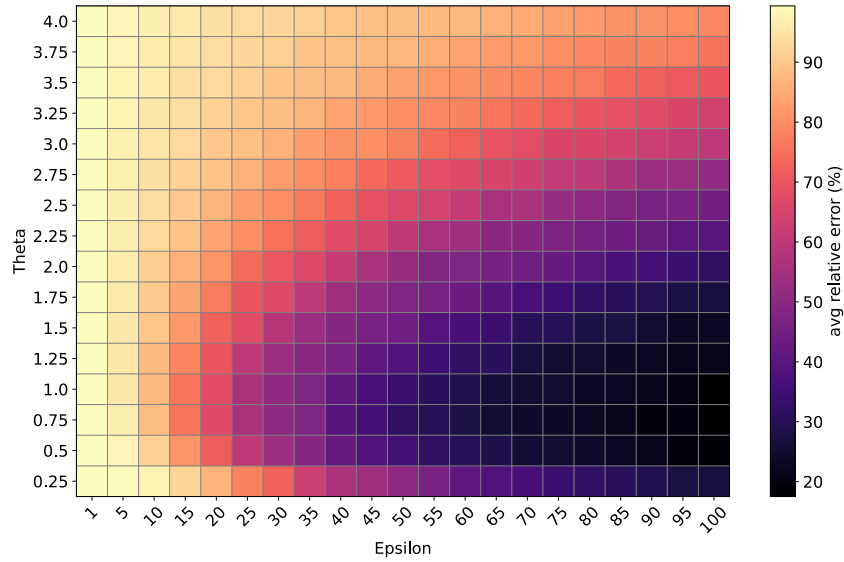
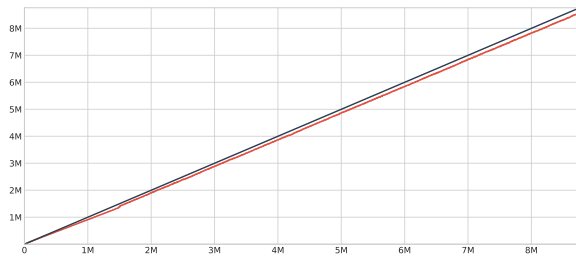
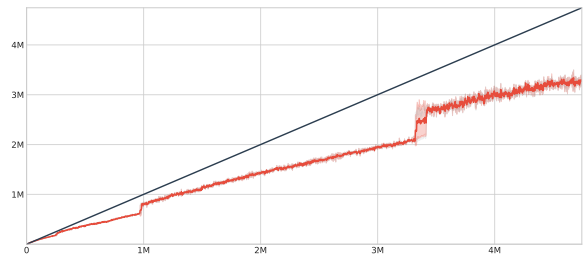


Figure A.2: Average relative error (%) on the LastFM dataset as a function of the total privacy budget ε and the telescoping decay rate θ . Darker colors represent lower average relative error.

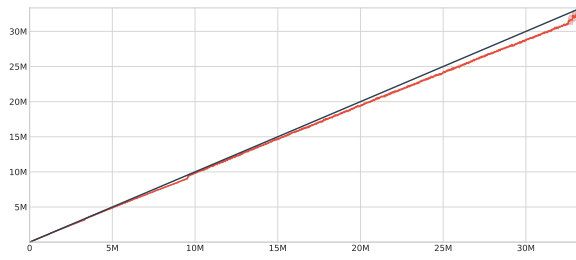
A.5 Dataset Selection



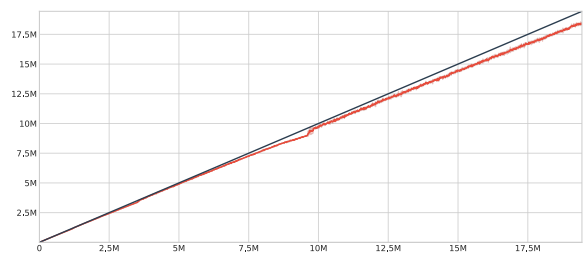
(a) Amazon Movies



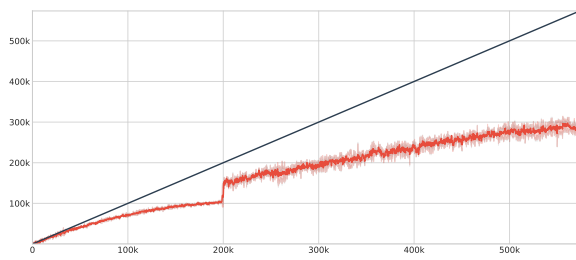
(b) Brightkite



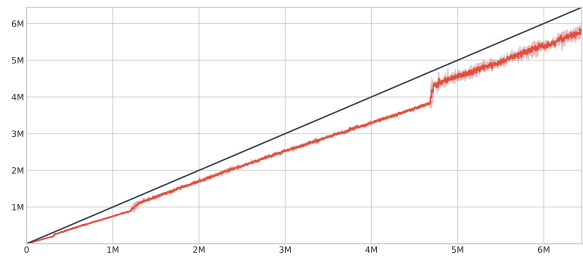
(c) Foursquare Global



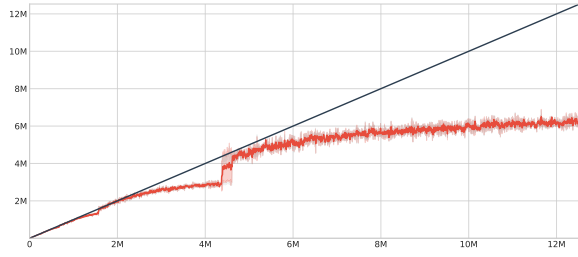
(d) AOL



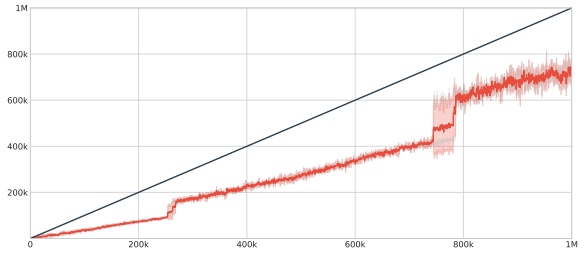
(e) Foursquare TKY



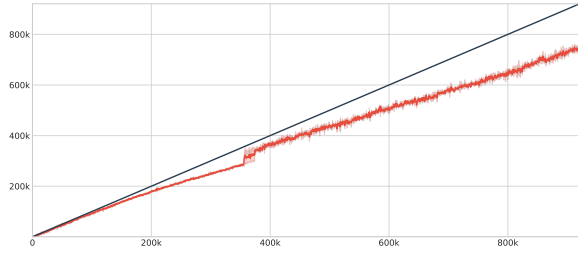
(f) Gowalla



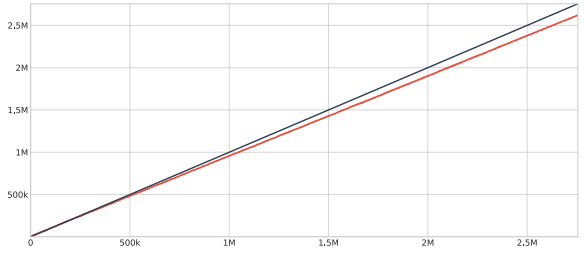
(a) Kuairec



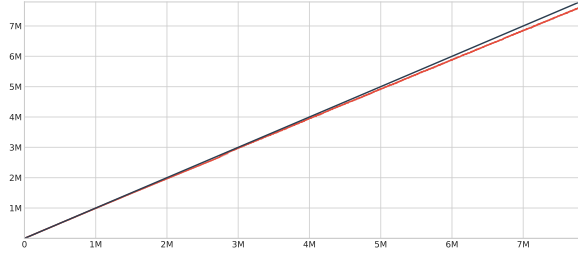
(b) Movielens 100K



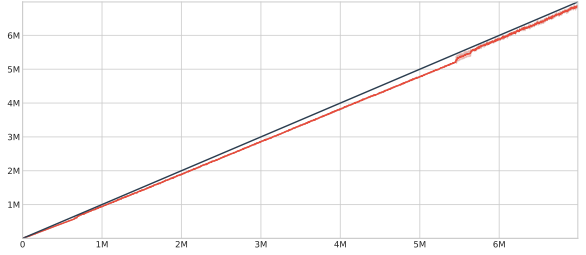
(c) Movietweetings



(d) Retailrocket

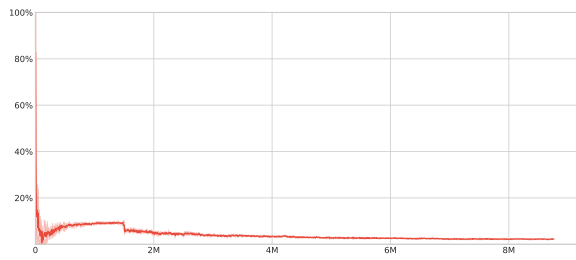


(e) Steam

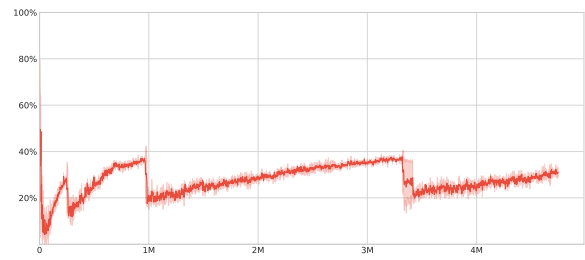


(f) Yelp

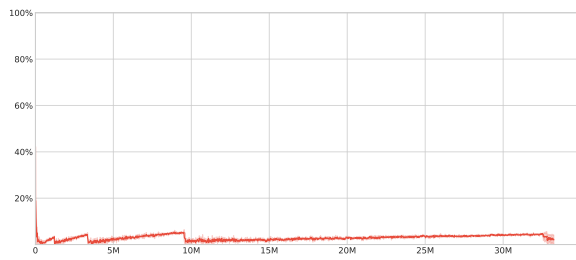
Figure A.4: Real count (black) vs DP count (red) across time: $\varepsilon = 1$, $\theta = 1$, $\beta = 0.1$



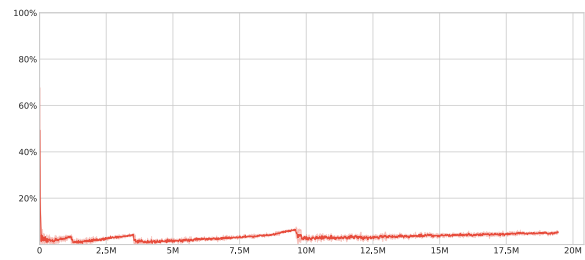
(a) Amazon Movies



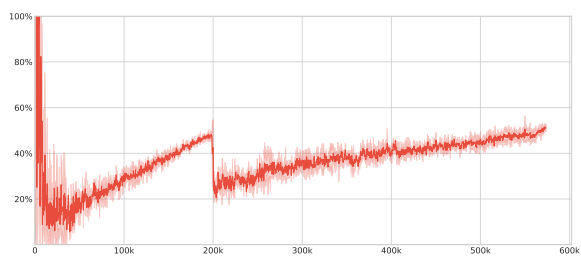
(b) Brightkite



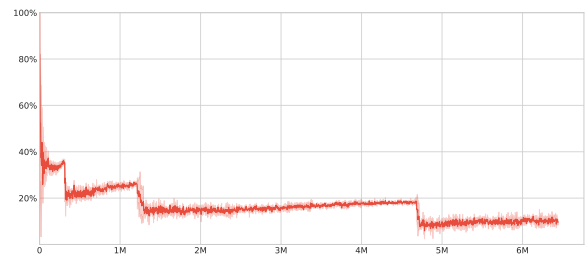
(c) Foursquare Global



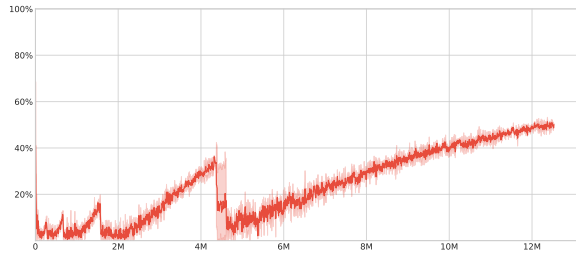
(d) AOL



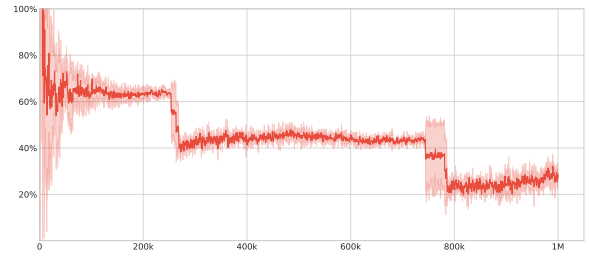
(e) Foursquare TKY



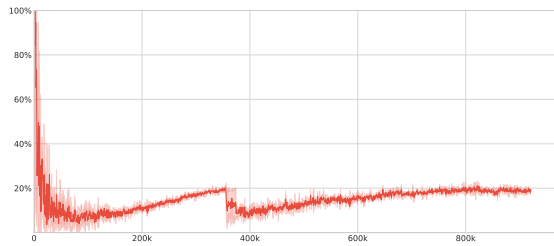
(f) Gowalla



(a) Kuairec



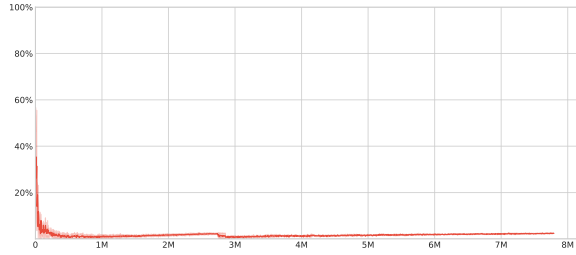
(b) Movielens 100K



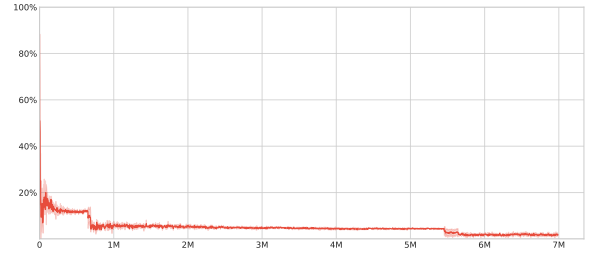
(c) Movietweetings



(d) Retailrocket



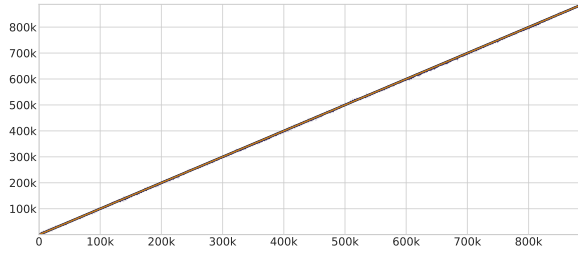
(e) Steam



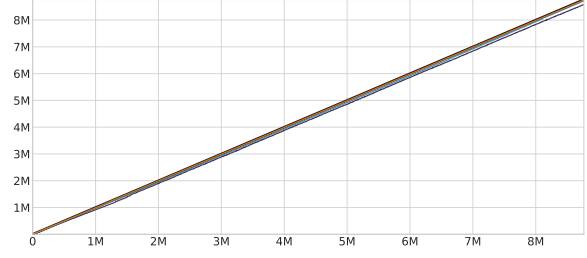
(f) Yelp

Figure A.6: Relative error across time with $\varepsilon = 1$, $\theta = 1$ and $\beta = 0.1$

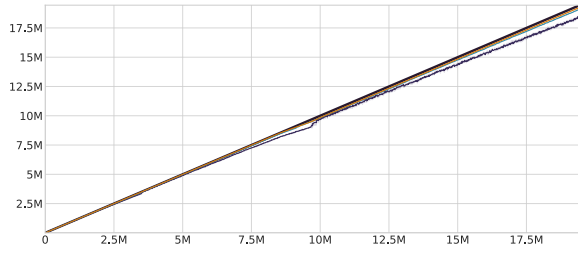
A.6 Utility Evaluation



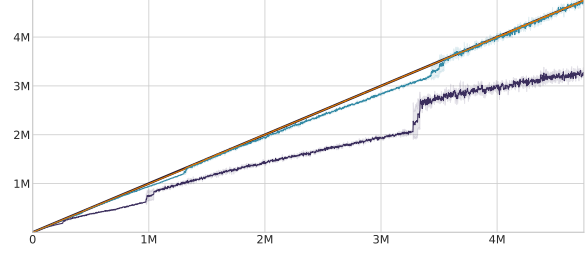
(a) Amazon Fashion



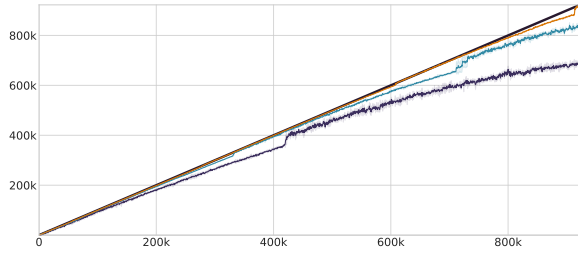
(b) Amazon Movies



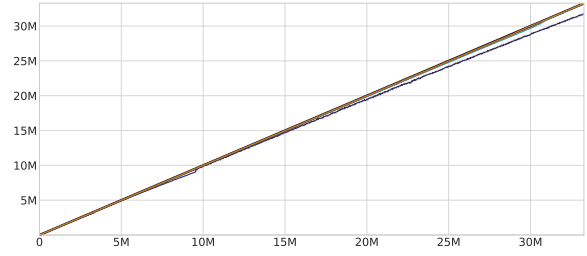
(c) AOL



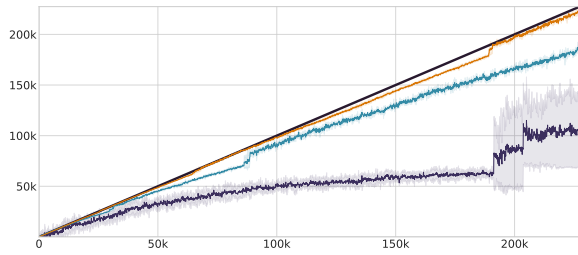
(d) Brightkite



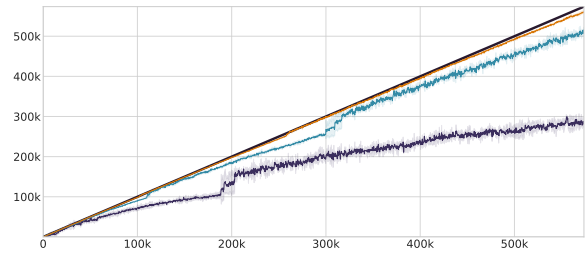
(e) Epinions



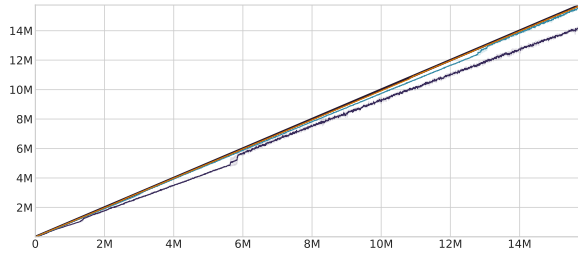
(f) Foursquare Global



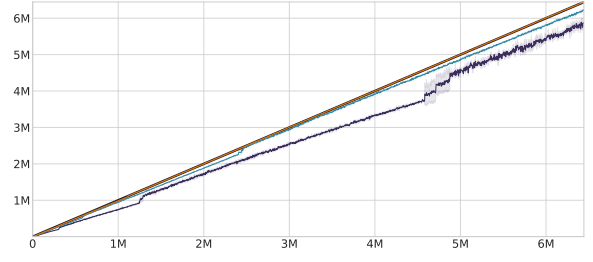
(g) Foursquare NYC



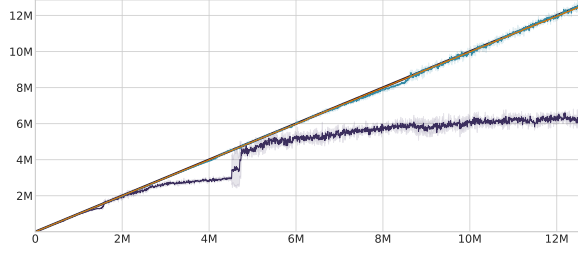
(h) Foursquare TKY



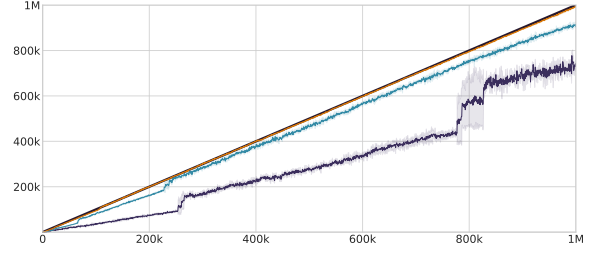
(a) Goodreads



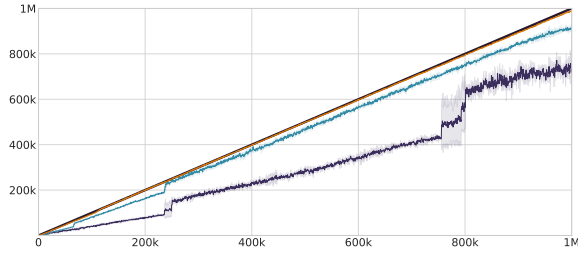
(b) Gowalla



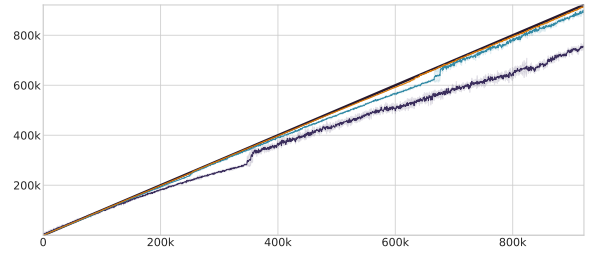
(c) KuaiREC



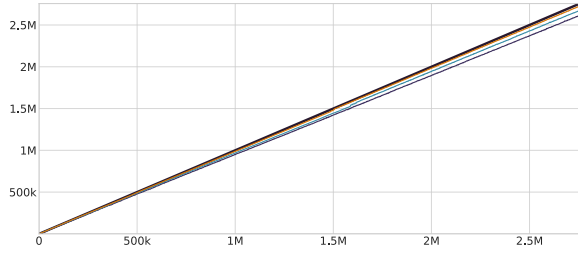
(d) MovieLens 100K



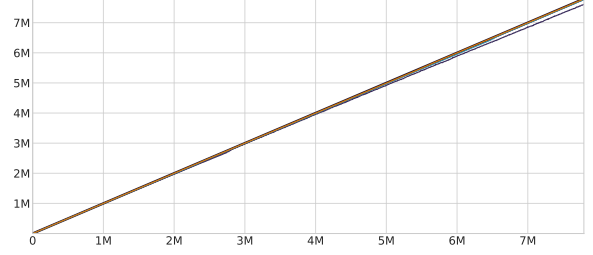
(e) MovieLens 1M



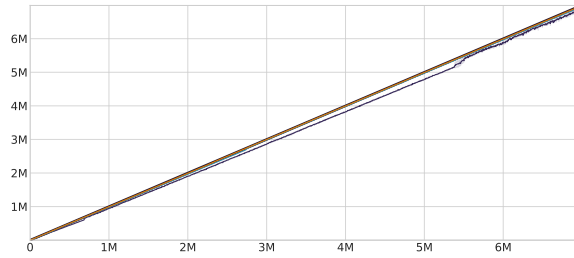
(f) Movietweetings



(g) Retailrocket

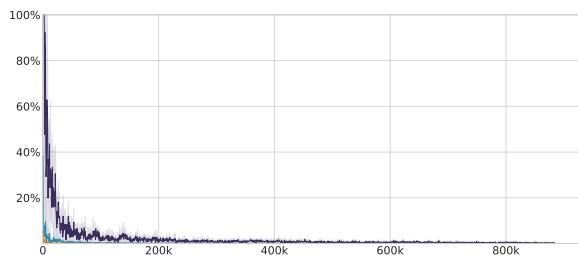


(h) Steam

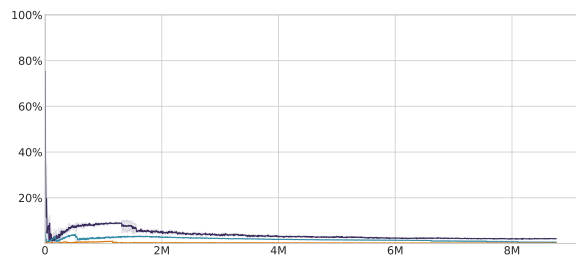


(i) Yelp

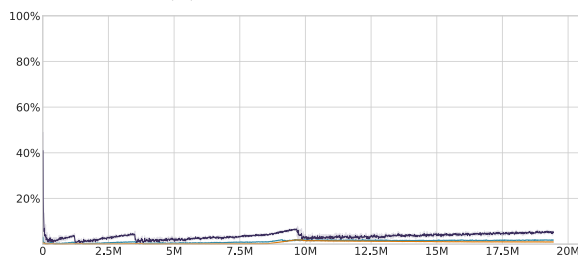
Figure A.8: Real count (black) vs DP count across time: $\varepsilon \in \{1, 10, 100\}$ (dark blue, teal, orange), $\theta = 1$, $\beta = 0.1$



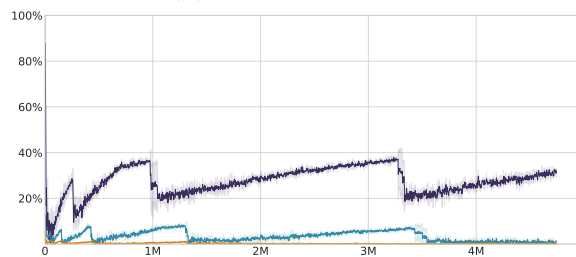
(a) Amazon Fashion



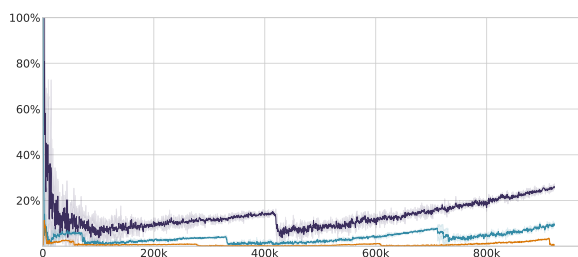
(b) Amazon Movies



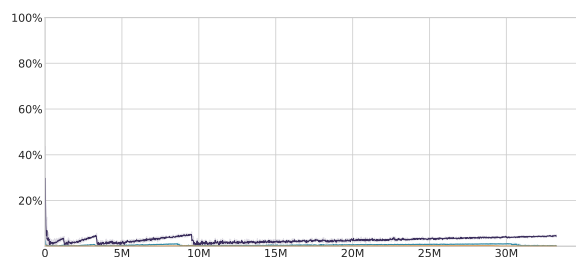
(c) AOL



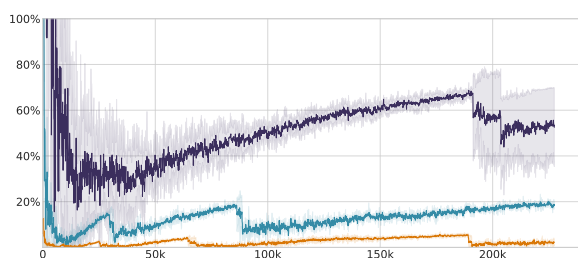
(d) Brightkite



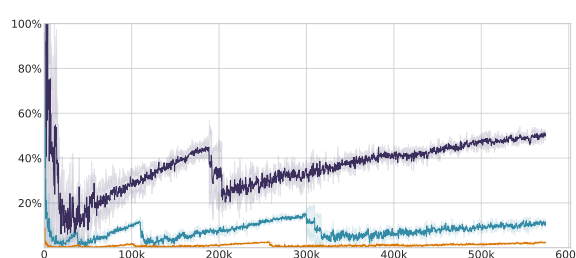
(e) Epinions



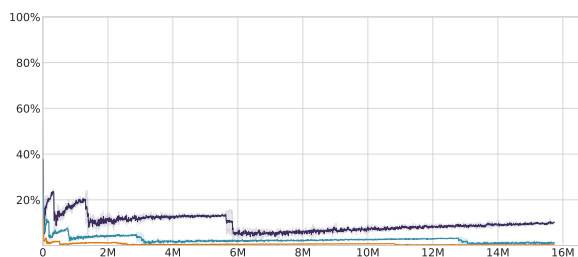
(f) Foursquare Global



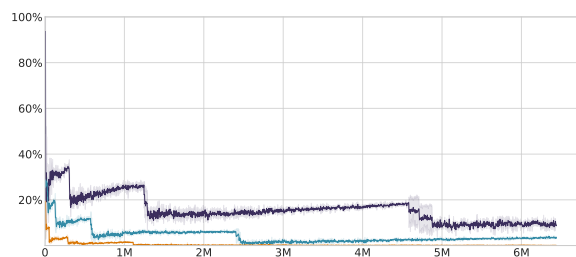
(g) Foursquare NYC



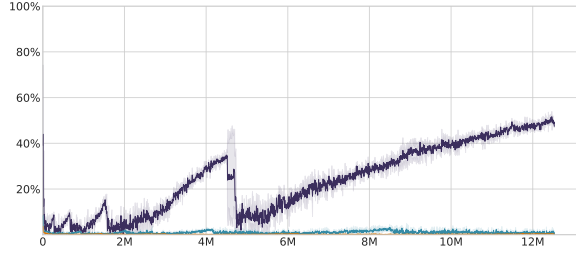
(h) Foursquare TKY



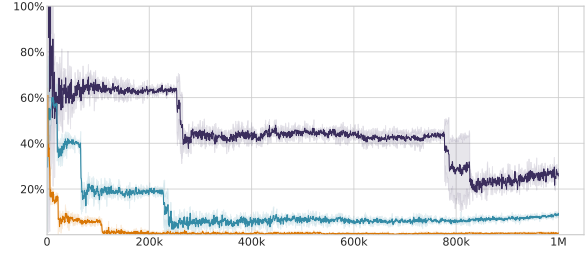
(i) Goodreads



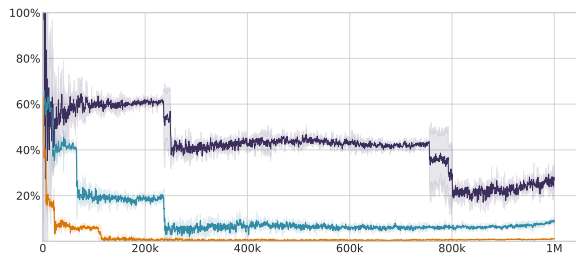
(j) Gowalla



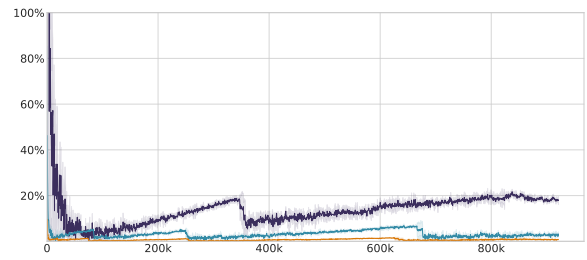
(a) KuaiREC



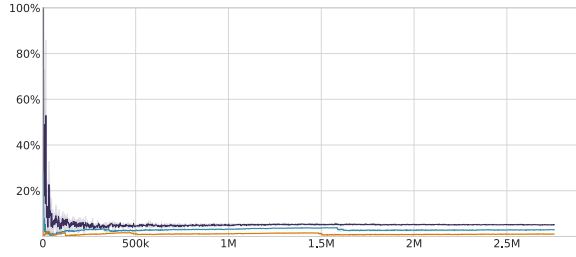
(b) Movielens 100K



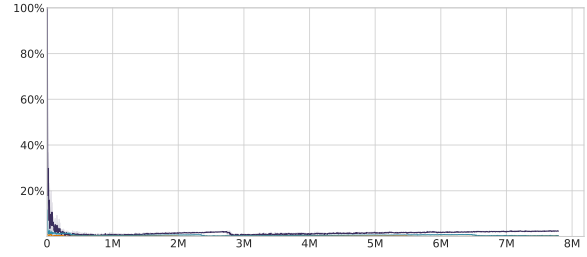
(c) Movielens 1M



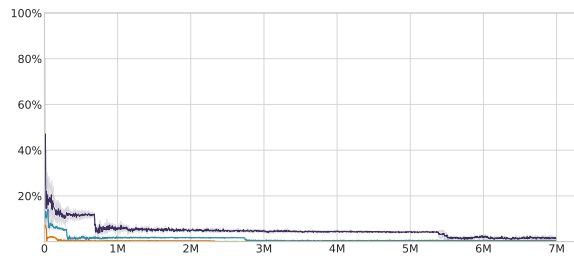
(d) Movietweetings



(e) Retailrocket



(f) Steam



(g) Yelp

Figure A.10: Relative error across time with $\varepsilon \in \{1, 10, 100\}$ (dark blue, teal, orange), $\theta = 1$ and $\beta = 0.1$

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl