Turbo: Effective Caching in Differentially-Private Databases

Abstract

Differentially-private (DP) databases allow for privacy-preserving analytics over sensitive datasets or data streams. In these systems, *user privacy* is a limited resource that must be conserved with each query. We propose *Turbo*, a novel, state-of-the-art caching layer for linear query workloads over DP databases. Turbo builds upon private multiplicative weights (PMW), a DP mechanism that is powerful in theory but very ineffective in practice, and transforms it into a highly-effective caching mechanism, *PMW-Bypass*, that uses prior query results obtained through an external DP mechanism to train a PMW to answer arbitrary future linear queries accurately and "for free" from a privacy perspective. Our experiments on public Covid19 and CitiBike datasets show that Turbo with PMW-Bypass conserves $1.7 - 15.9\times$ more budget compared to vanilla PMW and simpler cache designs, a significant improvement. Moreover, Turbo provides support for range query workloads, such as timeseries or streams, where opportunities exist to further conserve privacy budget through DP parallel composition and warm-starting of PMW state. Our work thus provides a theoretical foundation and general system design for effective caching in DP databases.

1 Introduction

Acme collects lots of user data from its digital products to analyze trends, improve existing products, and develop new ones. To protect user privacy, the company uses a restricted interface that removes personally identifiable information and only allows queries over aggregated data from multiple users. Internal analysts use interactive tools like Tableau to examine static datasets and run jobs to calculate aggregate metrics over data streams. Some of these metrics are shared with external partners for product integrations. However, due to data reconstruction attacks on similar "anonymized" and "aggregated" data from other sources, including the US Census Bureau [24] and Aircloak [13], the CEO has decided to pause external aggregate releases and severely limit the number of analysts with access to user data statistics until the company can find a more rigorous privacy solution.

Similar scenarios occur often in industry and government, leading to obstacles to data analysis or incomplete privacy solutions. In 2007, Netflix withdrew "anonymized" movie rating data and canceled a competition due to de-anonymization attacks [44]. In 2008, genotyping aggregate information from a clinical study led to the revelation of participants' membership in the diagnosed group, prompting the National Institutes of Health to advise against the public release of statistics from clinical studies [2]. In 2021, New York City excluded demographic information from datasets released from their CitiBike bike rental service due to vulnerabilities that could reveal sensitive user data [1]. The city's new, more restrained data release not only remains susceptible but also prevents analyses of how demographic groups use the service.

Differential privacy (DP) provides a rigorous solution to the problem of protecting user privacy while analyzing and sharing statistical aggregates over a database. DP guarantees that analysts cannot confidently learn anything about any individual in the database that they could not learn if the individual were not in the database. Industry and government have started to deploy DP for various use cases [54], including publishing trends in Google searches related to Covid19 [8], sharing LinkedIn user engagement statistics with outside marketers [48], enabling analyst access to Uber mobility data while protecting against insider attacks [30], and releasing the US Census' 2020 redistricting data [3]. To facilitate the application of DP, industry have developed a suite of systems, ranging from specialized designs like the US Census' Top-Down [3] and LinkedIn's Audience Engagements API [48] to more general DP SQL systems, like GoogleDP [59] and Uber's Chorus [30], or Map-Reduce analytics like Plume [5].

DP systems face a significant challenge that hinders their wider adoption: they struggle to handle large workloads of queries while maintaining a reasonable privacy guarantee. This is known as the "running out of privacy budget" problem and affects any system, whether DP or not, that aims to release multiple statistics from a sensitive dataset. A seminal paper by Dinur and Nissim [18] proved that releasing too many accurate linear statistics from a dataset fundamentally enables its reconstruction, setting a lower bound on the necessary error in queries to prevent such reconstruction. Successful reconstructions of the US Census 2010 data [24] and Aircloak's data [13] from the aggregate statistics released by these entities, exemplify this fundamental limitation. DP, while not immune to this limitation, provides a means of bounding the reconstruction risk. DP randomizes the output of a query to limit the influence of individual entries in the dataset on the result. Each new DP query increases this limit, consuming part of a *global privacy budget* that must not be exceeded, lest individual entries become vulnerable to reconstruction.

Recent work proposed treating the global privacy budget as a *system resource* that must be managed and conserved, similar to traditional resources like CPU [38]. When computation is expensive, *caching* is a go-to solution: it uses past results to save CPU on future computations. Caches are ubiquituous in all computing systems: from the processor, to operating systems, databases and content distribution networks. In this paper, we thus ask: *How should caching work in DP systems to significantly increase the number of queries they can support under a privacy guarantee?*

We propose *Turbo*, an effective caching layer for DP SQL databases that boosts the number of linear queries (such as

sums, averages, and counts) that can be answered accurately under a fixed, global DP guarantee. In addition to incorporating a traditional *exact-match cache* that saves past DP query results and reuses them if the same query reappears, Turbo builds upon a powerful theoretical construct, known as *private multiplicative weights (PMW)* [26], that leverages past DP query results to learn a histogram representation of the dataset that can go on to answer *arbitrary* future linear queries for free once it has converged. While PMW provides compelling convergence guarantees *in theory*, we find it *very ineffective* in practice, being overrun even by the basic exact-match cache!

We make three main contributions to PMW design to boost its effectiveness and applicability. First, we develop *PMW-Bypass*, a variant of PMW that bypasses it during the privacy-expensive learning phase of its histogram, and switches to it once it has converged to reap its free-query benefits. This change requires a new mechanism for updating the histogram despite bypassing the PMW, plus new theory to justify its convergence. The PMW-Bypass technique is highly effective, significantly outperforming the exact-match cache in the number of queries it can support. Second, we optimize our mechanisms for workloads of range queries that do not access the entire database. These types of queries are typical in timeseries databases and data streams. For such workloads, we organize the cache as a tree of multiple PMW-Bypass objects and demonstrate that this approach outperforms alternative designs. Third, for streaming workloads, we develop warm-starting procedures for tree-structured PMW-Bypass histograms, resulting in faster convergence.

We formally analyze each of our techniques, focusing on privacy, per-query accuracy, and convergence speed. Each technique represents a valuable contribution on its own and can be used separately, or, as we do in Turbo, as part of a coherent cache design suitable for both static databases and streams. We prototype Turbo on TimescaleDB, a timeseries database, and use Redis to store caching state. We evaluate Turbo on workloads based on Covid19 and CitiBike datasets. We show that Turbo significantly improves the number of linear queries that can be answered with less than 5% error (w.h.p.) under a global $(10, 0)$-DP guarantee, compared to not having a cache and alternative cache designs. Our approach outperforms the best-performing baseline in each workload by 1.7 to 15.9 times, and even more significantly compared to vanilla PMW and systems with no cache at all (such as most existing DP systems). These results demonstrate that our Turbo cache design is both general and effective in boosting workloads in DP SQL databases and streams, making it a promising solution for companies like Acme that seek an effective DP SQL system to address their user data analysis and sharing concerns. We plan to open-source Turbo.

2 Background

Threat model. We consider a threat model known as *centralized differential privacy*: one or more untrusted and potentially adversarial analysts can query a dataset or stream through a restricted, aggregate-only interface implemented by a trusted database engine of which Turbo is a trusted component. The goal of the database engine and Turbo is to provide accurate answers to the analysts' queries without compromising the privacy of individual users in the database. The two main adversarial goals that an analyst may have are membership inference and data reconstruction. Membership inference is when the adversary wants to determine whether a known data point is present in the dataset, while data reconstruction involves reconstructing unknown data points from a known subset of the dataset. To achieve their goals, the adversary can use composition attacks to single out contributions from individuals, collude with other analysts to coordinate their queries, link anonymized records to public datasets, and access arbitrary auxiliary information *except for* timing side-channel information. Such attacks have been practically demonstrated in previous research [44, 17, 23, 13, 24, 28].

Differential privacy (DP). DP [20] randomizes aggregate queries over a dataset to prevent membership inference and data reconstruction [58, 19]. DP randomization (a.k.a. noise) ensures that the probability of observing a specific result is stable to a change in one datapoint (e.g., if user x is removed or replaced in the dataset, the distribution over results remains similar). More formally, a query Q is (ϵ, δ)-DP if, for any two datasets D and D' that differ by one datapoint, and for any result S we have: $\mathbb{P}(Q(D) \in S) \leq e^{\epsilon}\mathbb{P}(Q(D') \in S) + \delta$. ϵ quantifies the privacy loss due to releasing the DP query's result (higher means less privacy), while δ can be interpreted as a failure probability and is set to a small enough value.

Two common mechanisms to enforce DP are the Laplace and Gaussian mechanisms. They add noise from an appropriately scaled Laplace/Gaussian distribution to the true query result, and return the noisy result. For counting queries of the type Turbo supports (see §4) and a database of size n, adding noise from Laplace$(0, 1/n\epsilon)$, ensures $(\epsilon, 0)$-DP (a.k.a. pure DP). Adding noise from Gaussian$(0, \sqrt{2\ln(1.25/\delta)}/n\epsilon)$ ensures (ϵ, δ)-DP. For linear queries, one can probabilistically control the accuracy of a query by converting it into the necessary (ϵ, δ) parameters for the Laplace/Gaussian mechanism.

Answering multiple queries on the same data fundamentally reduces the privacy guarantee [18]. DP quantifies this over a sequence of DP queries using the *composition property*, which in its basic form states that releasing two (ϵ_1, δ_1)-DP and (ϵ_2, δ_2)-DP queries is $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$-DP. When queries access disjoint subsets of the data, their DP guarantees are $(\max(\epsilon_1, \epsilon_2), \max(\delta_1, \delta_2))$-DP, which is called *parallel composition* [42]. Using composition, one can enforce a global (ϵ_G, δ_G)-DP guarantee over a workload, with each DP query "consuming" part of the global privacy budget [49].

Private multiplicative weights (PMW). PMW is a DP mechanism to answer online linear queries with bounded error [26]. It maintains an approximation of the dataset as a *histogram*: estimated counts of how many times any possible data point appears in the dataset. When a query arrives, an answer is

estimated using the histogram, and the *error of this estimate* is computed with DP on the real data, using a DP mechanism called *sparse vector (SV)* (described shortly). If the estimate's error is low, it is returned to the analyst, consuming no privacy budget (i.e., the query is answered "for free"). If the estimate's error is large, then PMW executes the DP query on the data with the Laplace or Gaussian mechanism. It returns the result to the analyst, and also uses it to update the histogram for more accurate estimates to future queries.

However, an additional cost in using PMWs stems from the SV, a stateful DP mechanism to test the error of a sequence of query estimates [21]. SV processes queries and their estimates one-by-one. While the estimates have error below a preset threshold with high probability, SV returns success and consumes zero privacy. However, if SV detects a large-error estimate, it requires a *reset*, which is expensive, costing 3× the privacy budget of executing one DP query on the data.

The theoretical vision of PMWs is as follows. Under a stream of queries, PMW first goes through a "training" phase, where its histogram is inaccurate, requiring frequent SV resets and consuming privacy budget. Failed estimation attempts update the histogram with low-error results obtained by running the DP query. Once the histogram becomes sufficiently accurate, the SV tests consistently pass, thereby ameliorating the initial training cost. Theoretical analyses provide a compelling *worst-case convergence* guarantee for the histogram, determining a worst-case number of updates required to train a histogram that can answer *any future linear query* with low error [27]. However, no one has examined whether this worst-case bound is practical and if PMW can deliver savings over a baseline as simple as an exact-match cache.

3 Turbo Overview

Turbo is a caching layer that can be integrated into a DP SQL engine, significantly increasing the number of linear queries that can be executed under a fixed, global (ϵ_G, δ_G)-DP guarantee. We focus on *linear queries* like sums, averages, and counts (defined in §4), which are widely used in interactive analytics and constitute the class of queries supported by approximate databases such as BlinkDB [4]. These queries enable powerful forms of caching like PMW, and also allow for accuracy guarantees, which are important when doing approximate analytics, as one does on a DP database.

3.1 Design Goals

In designing Turbo, we were guided by several goals:

- **(G1)** *Guarantee privacy:* Turbo must satisfy (ϵ_G, δ_G)-DP.
- **(G2)** *Guarantee accuracy:* Turbo must ensure (α, β)-*accuracy* for each query, defined for $\alpha > 0$, $\beta \in (0, 1)$ as follows: if R' and R are the returned and true results, then $|R' - R| \leq \alpha$ with $(1 - \beta)$ probability. If β is small, a result is α-*accurate w.h.p.* (with high probability).
- **(G3)** *Strive for convergence guarantees:* We aim to maintain PMW's theoretical convergence, allow for downgrades when necessary and analyze what is lost.
- **(G4)** *Prioritize empirical convergence:* While allowing for some form of worst-case convergence, Turbo's design should prioritize *empirical convergence*, measured on a workload as the number of updates needed to start answering (say) 90% of the queries for free.
- **(G5)** *Improve privacy budget consumption:* We aim for *significant improvements* in privacy budget consumption compared to both not having a cache and having an exact-match cache or a vanilla PMW.
- **(G6)** *Support multiple use cases:* Turbo should provide significant benefits for multiple important workload types, including static and streaming databases.
- **(G7)** *Easy to configure:* Turbo should include few knobs with fairly stable performance.

(G1) and (G2) are strict requirements. (G3) and (G4) are driven by our belief that DP systems should not only possess meaningful theoretical properties but also be optimized for practice. (G5) is our main objective. (G6) requires further attention, given shortly. (G7) is driven by the limited guidance from PMW literature on parameter tuning. PMW meets goals (G1-G3) but falls significantly short for (G4-G7). Turbo achieves all goals; we provide theoretical analyses for (G1-G3) in §4 and empirical evaluations for (G4-G7) in §5.

3.2 Use Cases

The DP literature is fragmented, with different algorithms developed for different use cases. We seek to create a coherent system that supports multiple settings, highlighting three here:
(1) Non-partitioned databases are the most common use case in DP. A group of untrusted analysts issue queries over time against a static database, and the database owner wishes to enforce a global DP guarantee. Turbo should allow a higher workload of queries compared to existing approaches.
(2,3) Partitioned databases are less frequently investigated in DP theoretical literature, but important to distinguish in practice [42, 52]. When queries tend to access different data ranges, it is worth partitioning the data and accounting for consumed privacy budget in each partition separately through DP's parallel composition. This lowers privacy budget consumption in each partition and permits more non- or partially-overlapping queries against the database. This kind of workload is inherent in *timeseries* and *streaming databases*, where analysts typically query the data by *windows of time*, such as how many new Covid cases occurred in the week after a certain event, or what is the average age of positive people over the past week. We distinguish two cases:

(2) Partitioned static database, where the database is static and partitioned by an attribute that tends to be accessed in ranges, such as time, age, or geo-location. All partitions are available at the beginning. Queries arrive over time, and most are assumed to run on some range of interest, which can involve one or more partitions. Turbo should provide significant benefit not only compared to the baseline caching techniques, but also compared to not having partitioning.

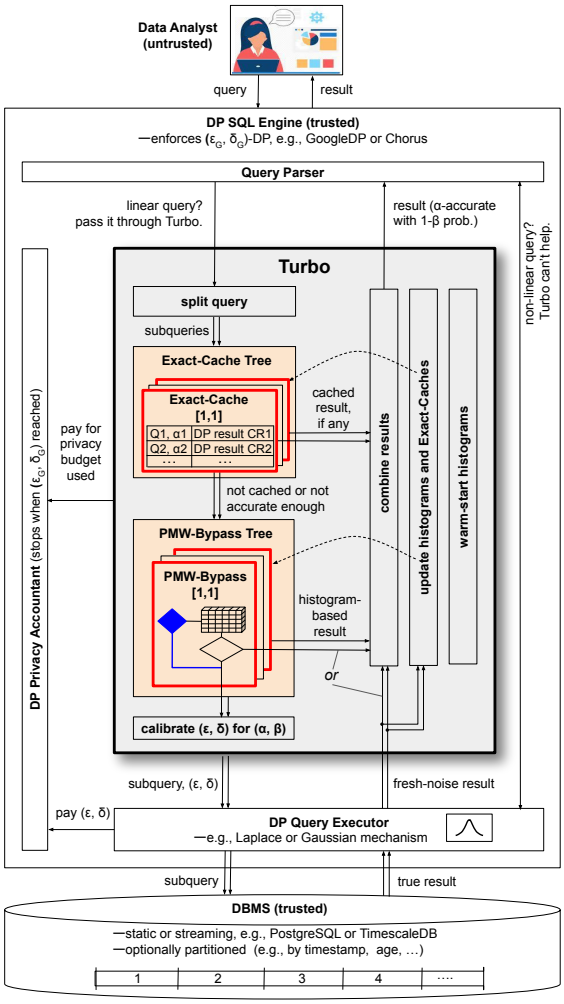


Fig. 1. Turbo architecture.

(3) Partitioned streaming database, where the database is partitioned by time and partitions arrive over time. In such workloads, queries tend to run *continuously* as new data becomes available. Hence, new partitions see a similar query workload as preceding partitions. Turbo should take advantage of this similarity to further conserve privacy.

3.3 Turbo Architecture

Fig. 1 shows the Turbo architecture. It is a caching layer that can be added to a DP SQL engine, like GoogleDP [59] or Uber's Chorus [30], to boost the number of linear queries that can be answered accurately under a fixed global DP guarantee. The filled-background components indicate our additions to the DP SQL engine, while the transparent-background components are standard in any DP SQL engine. Briefly, here is how a typical DP SQL engine works *without Turbo*. Analysts issue queries against the engine, which is trusted to enforce a global (ϵ_G, δ_G)-DP guarantee. The engine executes the queries using a DP query executor, which adds noise to query results with the Laplace/Gaussian mechanism and consumes a part of the global privacy budget. A budget accountant tracks the consumed budget; when it runs out, the DP SQL engine either stops responding to new queries or sacrifices privacy by "resetting" the budget (e.g., Uber's Chorus [30] does the former, LinkedIn's Audience Insights [47, 48] does the latter). Our work applies to both cases, but we focus on the former.

Turbo intercepts the queries before they go into the DP query executor and performs a very proactive form of caching for them, reusing prior results as much as possible to avoid consuming privacy budget for new queries. Turbo's architecture is organized in two types of components: *caching objects* (denoted in light-orange background in Fig. 1) and *functional components* that act upon them (denoted in grey background).

Caching objects. Turbo maintains several types of caching objects. First, the *Exact-Cache* stores previous queries and their DP results, allowing for direct retrieval of the result without consuming any privacy budget when the same query is seen again on the same database version. Second, the *PMW-Bypass* is an improved version of PMW that reduces privacy budget consumption during the training phase of its histogram (§4.2). Given a query, PMW-Bypass uses an effective heuristic to judge whether the histogram is sufficiently trained to answer the query accurately; if so, it uses it, thereby spending no budget. Critically, PMW-Bypass includes a mechanism to *externally update* the histogram even when bypassing it, to continue training it for future, free-budget queries.

Turbo aims to enable parallel composition for workloads that benefit from it, such as timeseries or streaming workloads, by supporting database partitioning. Partitions can be defined by attributes with public values that are typically queried by range, such as time, age, or geo-location. Turbo uses a *tree-structured PMW-Bypass* caching object, consisting of multiple histograms organized in a binary tree, to support linear range queries over these partitions effectively (§4.3). This approach conserves more privacy budget and enables larger workloads to be run when queries access only subsets of the partitions, compared to alternative methods.

Functional components. When Turbo receives a linear query through the DP SQL engine's query parser, it applies its caching objects to the query. If the database is partitioned, Turbo splits the query into multiple sub-queries based on the available tree-structured caching objects. Each sub-query is first passed through an Exact-Cache, and if the result is not found, it is forwarded to a PMW-Bypass, which selects whether to execute it on the histogram or through direct Laplace/Gaussian. For sub-queries that can leverage histograms, the answer is supplied directly without execution or budget consumption. For sub-queries that require execution with Laplace/Gaussian, the (ϵ, δ) parameters for the mechanism are computed based on the desired (α, β) accuracy, using the "calibrate (ϵ, δ) for (α, β)" functional component in Fig. 1. Then, each sub-query and its privacy parameters are passed to the DP query executor for execution.

Turbo combines all sub-query results obtained from the caching objects to form the final result, ensuring that it is within α of the true result with probability $1 - \beta$ (functional

component "combine results"). New results computed with fresh noise are used to update the caching objects (functional component "update histograms and Exact-Caches"). Additionally, Turbo includes cache management functionality, such as "warm-start of histograms," which reuses trained histograms from previous partitions to warm-start new histograms when a new partition is created (§4.4). This mechanism is effective in streams where the data's distribution and query workload are stable across neighboring partitions. Theoretical and experimental analyses show that external histogram updates and warm-start mechanisms give convergence properties similar to, but slightly slower than, vanilla PMW.

4 Detailed Design

We next detail the novel caching objects and mechanisms in Turbo, using different use cases from §3.2 to illustrate each concept. We describe PMW-Bypass in the static, non-partitioned database, then introduce partitioning for the tree-structured PMW-Bypass, followed by the addition of streaming to discuss warm-start procedures. We focus on the Laplace mechanism and basic composition, thus only discussing pure $(\epsilon, 0)$-DP and ignoring δ. We also assume β is small enough for Turbo results to count as α-accurate w.h.p. Supplementary Material adds back δ, β, Gaussian, and Rényi composition.

First, we give some notation. Given a data domain $\mathcal{X}$, a database x with n rows can be represented as a histogram $h \in \mathbb{N}^{\mathcal{X}}$: for a data point $v \in \mathcal{X}$, $h(v)$ is the number of rows in x that are equal to v. $h(v)$ is v's *bin* in the histogram. We denote $N = |\mathcal{X}|$ the size of the data domain, and n the size of the database. When $\mathcal{X}$ has the form $\{0,1\}^d$, we call d the data domain dimension. Example: a database with 3 binary attributes has domain $\mathcal{X} = \{0,1\}^3$ of dimension $d = 3$ and size $N = 8$; $h(0,0,1)$ is the number of rows equal to $(0,0,1)$.

Second, we define *linear queries* as SQL queries that can be transformed or broken into the following form:

SELECT AVG$(*)$ **FROM** (**SELECT** f(A, B, C, ...) **FROM Table**),

where `f` takes d arguments (one for each attribute of `Table`) and outputs a value in $[0,1]$. When `f` has values in $\{0,1\}$, a query returns the fraction of rows satisfying predicate `f`. To get raw counts, we multiply by n, which we assume is public.

4.1 Running Example

Fig. 2 gives a running example inspired by our evaluation Covid dataset. Analysts run queries against a database consisting of Covid test results over time. Fig. 2(a) shows a simplified version of the database, with only three attributes: the test's date, T; the outcome, P, which can be 0 or 1 for negative/positive; and subject's age bracket, A, with one of four values as in the figure. The database could be either static or actively streaming in new test data. Initially, we assume it static and ignore the T attribute. Our example database has $n = 100$ rows and data domain size $N = 8$ for P and A.

Fig. 2(b) shows two queries that were previously executed. While queries in Turbo return the *fraction* of entries satisfying a predicate, for simplicity we show raw counts. $Q1$ requests the positivity rate and $Q2$ the fraction of tested minors. Fig. 2(d) shows the next query that will be executed, $Q3$, requesting the fraction of positive minors. These queries are not identical, but they are *correlated* as they access overlapping data. Thus, while $Q1$ and $Q2$ results cannot be used to directly answer $Q3$, intuitively, they should help. That is the promise of PMW (and PMW-Bypass) through its histogram.

(a) "Covid" Table:

Time (T)	Positive (P)	Age Bracket (A)
02/01/21	0	0 (1-17)
02/01/21	1	1 (18-49)
02/02/21	1	2 (50-64)
02/02/21	1	3 (65+)
… say n=100 total rows …		

(b) Previously executed queries:

Q1: SELECT COUNT(*) FROM Covid WHERE Positive=1
Q2: SELECT COUNT(*) FROM Covid WHERE AgeBracket=0

(c) Histogram state after Q1, Q2:

	A = 0	A = 1	A = 2	A = 3
P = 0	Est.: 8 (Real: 13) C: 1	Est.: 21.7 (Real: 27) C: 0	Est.: 21.7 (Real: 15) C: 0	Est.: 21.7 (Real: 25) C: 0
P = 1	Est.: 2.9 (Real: 3) C: 2	Est.: 8 (Real: 5) C: 1	Est.: 8 (Real: 8) C: 1	Est.: 8 (Real: 4) C: 1

(d) Next query: Q3: SELECT COUNT(*) FROM Covid WHERE Positive=1 AND AgeBracket=0

Fig. 2. Running example. (a) Simplified version of Covid dataset. (b) Two queries that were previously run. (c) State of the histogram, showing for each histogram bin: `Est.`: estimated count; `Real`: real count (not available, just shown for reference); `C`: a PMW-Bypass-specific variable counting the number of previous queries that have updated a bin. (d) Next query to run.

Fig. 2 (c) shows the state of the histogram after executing $Q1$ and $Q2$ but before executing $Q3$. Each bin in the histogram stores an *estimation* of the number of rows equal to (p, a). This is the `Est.` field in the figure. Initially, `Est.` in all bins is set assuming a uniform distribution over $P \times A$; in this case the initial value was $n/N = 12.5$. The figure also shows the real (non-private) count for each bin (denoted `Real`), which is *not* part of the histogram, but we include it as a reference. As queries are executed, `Est.` values are updated with fresh-noise results, based on which bins are accessed. $Q1$ and $Q2$ have already been executed, and both are assumed to have resorted to the Laplace mechanism, so they both contributed fresh-noise results to specific bins. $Q1$ accessed, and hence updated, data in the $P = 1$ bins (the bottom row of the histogram). $Q2$ did so in the $A = 0$ bins (the left column of the histogram). Through a renormalization step, these queries have also changed the other bins, though not necessarily in a query-informed way. The C variable in each bin shows the number of queries that have purposely updated that bin. We can see that estimates in the $C > 0$ bins are a bit more accurate compared to those in the $C = 0$ bins. The only bin that has been updated twice is $(P = 1, A = 0)$, as it lies at the intersection of both queries; that bin has diverged from

its neighboring, singly-updated bins and is getting closer to its true value. (Bin $(P = 1, A = 2)$ looks even more accurate despite being updated once, but that is purely by chance.)

Our last query, $Q3$, which accesses $(P = 1, A = 0)$, may be able to leverage its estimation "for free," assuming the estimation's error – which itself must be assessed privately – is within α w.h.p. (in reality, this is highly unlikely after just a couple of updates, but we use very aggressive learning here for demonstration purposes). Assessing that the error is within α – *without consuming privacy budget if it is* – is the purview of the SV mechanism incorporated in a PMW. The catch is that the SV consumes privacy budget, in copious amounts, if this test fails. This is what makes vanilla PMW impractical, a problem that we address next.

4.2 PMW-Bypass

PMW-Bypass addresses practical inefficiencies of the vanilla, mostly theoretical PMW, which we show with an experiment.

Demo experiment. Using a four-attribute Covid dataset with domain size 128 (so a bit larger than in our running example), we generate a query pool of over 34K unique queries by taking all possible combinations of values over the four attributes. From this pool, we sample uniformly 35K queries to form a workload. This workload should be an *ideal showcase* for PMW: there are many unique queries relative to the small data domain size (giving the PMW ample chance to train), and while most queries are unique, they tend to overlap in the data they touch (giving the PMW ample chance to reuse information from previous queries). We evaluate the cumulative privacy budget spent as queries are executed, comparing the case where we execute them through PMW vs. directly with Laplace, with and without an exact-match cache.

Fig. 3 shows the results. As expected for this workload, the PMW works, as it converges after roughly the first 10K queries and consumes very little budget afterwards. However, before converging, the PMW consumes enormous budget. In contrast, direct execution through Laplace grows linearly, but much more slowly compared to PMW's beginning. The PMW eventually becomes better than Laplace, but only after about 27K queries. Moreover, if instead of always doing direct execution through Laplace, we trivially cached the results in an exact-match cache for future reuse if the same query reappeared – a rare event in this workload – then the PMW would *never* become significantly better than this simple baseline! §5 expands this analysis for other workloads and datasets, with a similar outcome: *PMWs are ineffective in practice*.

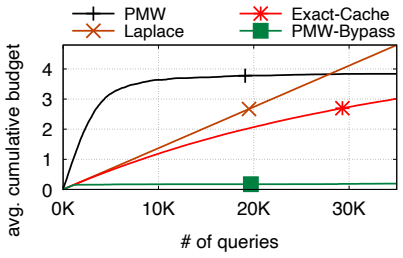


Fig. 3. Demo experiment.

We propose PMW-Bypass, a re-design for PMWs that makes them, in fact, *very effective*. We make multiple changes

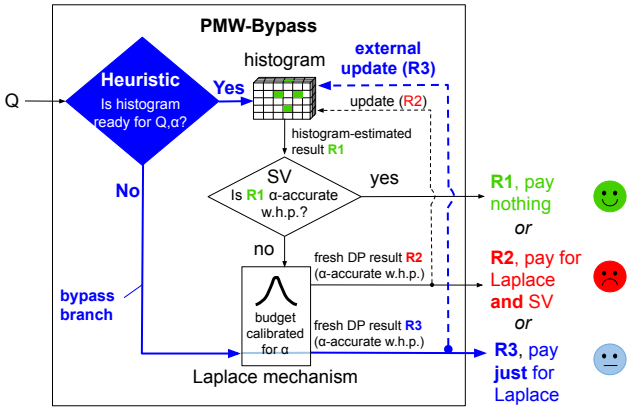


Fig. 4. PMW-Bypass. New components over vanilla PMW are in blue/bold.

to PMWs, but the main one involves *bypassing* the PMW while it is training (and hence expensive) and instead executing directly with Laplace (which is less expensive). Importantly, we do this while still updating the histogram with the Laplace results so that eventually the PMW becomes good enough to switch to it and reap its zero-privacy query benefits. The PMW-Bypass line in Fig. 3 shows just how effective this design is in our demo experiment: PMW-Bypass follows the low, direct-Laplace curve instead the PMW's up until the histogram converges, after which it follows the flat shape of PMW's convergence line. In this experiment, as well as in further experiments in §5, the outcome is the same: *our changes make PMWs very effective*. For this reason, we believe that PMW-Bypass should replace PMW in most settings where the latter is studied, not just in our system's design.

PMW-Bypass. Fig. 4 shows the functionality of PMW-Bypass, with the main changes shown in blue and bold. Without our changes, a vanilla PMW works as follows. Given a query Q, PMW first estimates its result using the histogram ($R1$) and then uses the SV protocol to test whether it is α-accurate w.h.p. The test involves comparing $R1$ to the *exact result* of the query executed on the database. If a noisy version of the absolute error between the two is within a threshold comfortably far from α, then $R1$ is considered accurate enough w.h.p. and outputted directly. This is the good case, because the query need not consume *any privacy*. The bad case is when the SV test fails. First, the query must be executed directly through Laplace, giving a result $R2$, whose release of course costs privacy. But beyond that, the SV must be *reset*, which consumes privacy. In total, if the direct Laplace execution costs ϵ, then releasing $R2$ costs $4 * \epsilon$! This is what causes the extreme privacy consumption during the training phase for vanilla PMW, when the SV test mostly fails. Still, in theory, after paying handsomely for this histogram "miss," $R2$ can at least be used to update the histogram (the arrow denoted "update (R2)" in Fig. 4), with the hope that the histogram will answer future correlated queries.

PMW-Bypass adds three components to PMW: (1) a *heuristic* that assesses whether the histogram is likely ready to answer Q with the desired accuracy; (2) a *bypass branch*, taken

if the histogram is deemed not ready and direct query execution with Laplace instead of going through (and likely failing) the SV test; and (3) an *external update* procedure that updates the histogram with the bypass branch result. Given Q, PMW-Bypass first consults the heuristic, which only inspects the histogram, so its use is free. Two cases arise:

Case 1: If the heuristic says the histogram is ready to answer Q with α-accuracy w.h.p., then the PMW is used, $R1$ is generated, and the SV is invoked to test $R1$'s actual accuracy. If the heuristic's assessment was correct, then this test will succeed, and hence the free, $R1$ output branch will be taken. Of course, no heuristic that lacks access to the raw data can guarantee that $R1$ will be accurate enough, so if the heuristic was actually wrong, then the SV test will fail and the expensive $R2$ path is taken. Thus, a key design question is whether there exist heuristics good enough to make PMW-Bypass effective. We discuss heuristic designs below, but the gist is that simple and easily tunable heuristics work surprisingly well, enabling the significant privacy budget savings in Fig. 3.

Case 2: If the heuristic says the histogram is not ready to answer Q with α-accuracy w.h.p., then the bypass branch is taken and Laplace is invoked directly, giving result $R3$. Now, PMW-Bypass must pay for Laplace, but because it bypassed the PMW, it does not risk an expensive SV reset. A key design question here is whether we can still reuse $R3$ to update the histogram, even though we did not, in fact, consult the SV to ensure that the histogram is truly insufficiently trained for Q. Perhaps surprisingly, performing the same kind of update as the PMW would do, from outside the protocol, would break its convergence! Instead, for PMW-Bypass, we design an *external update* procedure that *can* be used to update the histogram with $R3$ while preserving the same flavor of worst-case convergence as PMW, albeit with slower speed.

Heuristic IsHistogramReady. One option to assess if a histogram is ready to answer a query accurately is to check that it has received at least C updates in each bin the query needs, for some threshold C. This approach is imprecise, as some bins may require more updates to estimate with the same absolute error. Thus, we must use a separate threshold value per bin, raising the question of how to configure all these thresholds. To keep configuration easy (goal **(G6)**), we use an *adaptive per-bin threshold*. For each bin, we initialize its threshold C with a value C_0 and increment C by an additive step S_0 every time the heuristic makes a mistake (i.e. predicts it is ready when it is in fact not ready for that query). While the threshold is too small, the heuristic keeps getting penalized until it reaches a threshold high enough to avoid mistakes. For queries that span multiple bins, when the heuristic fails, we only penalize the least-updated bins; this ensures that a single, inaccurate bin will not set back the histogram for other queries using accurate bins. With these self-tuning, adaptive per-bin thresholds, we only configure initial parameters C_0 and S_0, which we find experimentally easy to do. While other heuristic designs exist, we find ours very effective.

Algorithm 1 PMW-Bypass algorithm.

1: **Cfg.:** PrivacyAccountant, Heuristic, accuracy params (α, β), histogram convergence params lr, τ, database Data with n rows.
2: **function** Update(h, q, η)
3: Multiply bin values: $\forall w \in \mathcal{X}$, $g(w) \leftarrow h(w)\exp(\eta q(w))$
4: Renormalize: $\forall w \in \mathcal{X}$, $h(w) \leftarrow g(w) / \sum_{v\in\mathcal{X}} g(v)$
5: **return** h
6: **function** CalibrateBudget(α, β)
7: **return** $\frac{4\ln(1/\beta)}{n\alpha}$
8: Initialize histogram h to uniform distribution on $\mathcal{X}$
9: $\epsilon \leftarrow$ CalibrateBudget(α, β)
10: PrivacyAccountant.pay($3 \cdot \epsilon$) // Pay to initialize first SV
11: **while** PrivacyAccountant.HasBudget() **do**
12: $\hat{\alpha} \leftarrow \alpha/2 + \text{Lap}(1/\epsilon n)$ // SV reset
13: $SV \leftarrow$ NotConsumed
14: **while** $SV ==$ NotConsumed **do**
15: Receive next query q
16: **if** Heuristic.IsHistogramReady(h, q, α, β) **then**
17: **// Regular PMW branch:**
18: **if** $|q(\text{Data}) - q(h)| + \text{Lap}(1/\epsilon n) < \hat{\alpha}$ **then** // SV test
19: Output $R1 = q(h)$ →**R1**, pay nothing
20: **else**
21: PrivacyAccountant.pay($4 * \epsilon$) → **R2**, pay for
22: Output $R2 = q(\text{Data}) + \text{Lap}(1/\epsilon n)$ Laplace, SV
23: // Update histogram (R2):
24: $\eta \leftarrow \begin{cases} lr & \text{if } R2 > q(h) \\ -lr & \text{if } R2 < q(h) \end{cases}$
25: $h \leftarrow$ Update(h, q, η)
26: $SV \leftarrow$ Consumed // force SV reset
27: Heuristic.Penalize(q, h)
28: **else**
29: **// Bypass branch:**
30: PrivacyAccountant.pay(ϵ) → **R3**, pay for
31: Output $R3 = q(\text{Data}) + \text{Lap}(1/\epsilon n)$ Laplace
32: // External update of histogram (R3):
33: $\eta \leftarrow \begin{cases} lr & \text{if } R3 > q(h) + \tau\alpha \\ -lr & \text{if } R3 < q(h) - \tau\alpha \\ 0 & \text{otherwise // no updates if we're not confident!} \end{cases}$
34: $h \leftarrow$ Update(h, q, η)

External updates. While we want to bypass the PMW when the histogram is not "ready" for a query, we still want to update the histogram with the result from the Laplace execution (R3); otherwise, the histogram will never get trained. That is the purpose of our external updates (lines 33-34 in Alg. 1). They follow a similar structure as a regular PMW update (lines 24-25 in Alg. 1), with a key difference. In vanilla PMW, the histogram is updated with the result $R2$ from Laplace *only when* the SV test fails. In that case, PMW updates the relevant bins in one direction or another, depending on the sign of the error $R2 - q(h)$. For example, if the histogram is underestimating the true answer, then R2 will likely be higher than the histogram-based result, so we should increase the value of the bins (case $R2 > q(h)$ of line 24 in Alg. 1).

In PMW-Bypass, external updates are performed not just when the authoritative SV test finds the histogram estimation inaccurate, but also when our heuristic predicts it to be

inaccurate, but may actually be accurate. In the latter case, performing external updates in the same way as PMW updates would add bias into the histogram and forfeit its convergence guarantee. To prevent this, in PMW-Bypass, external updates are executed only when we are quite confident, based on the direct-Laplace result $R3$, that the histogram overestimates or underestimates the true result. Line 33 shows the change: the term $\tau\alpha$ is a *safety margin* that we add to the comparison between the histogram's estimation and $R3$, to be confident that the estimation is wrong and the update warranted. This lets us prove worst-case convergence akin to PMW.

Learning rate. In addition to the bypass option, we make another key change to PMW design for practicality. When updating a bin, we increase or decrease the bin's value based on a learning rate parameter, lr, which determines the size of the update step taken (line 3 in Alg. 1). Prior PMW works fix learning rates that minimize theoretical convergence time, typically $\alpha/8$ [56]. However, our experiments show that larger values of lr can lead to much faster convergence, as dozens of updates may be needed to move a bin from its uniform prior to an accurate estimation. However, increasing lr beyond a certain point can impede convergence, as the updates become too coarse. Taking cue from deep learning, PMW-Bypass uses a scheduler to adjust lr over time. We start with a high lr and progressively reduce it as the histogram converges.

Guarantees. (G1) *Privacy:* PMW-Bypass preserves ϵ_G-DP across the queries it executes (Thm. A.1 in Supplementary Material). **(G2)** *Accuracy:* PMW-Bypass is α-accurate with $1-\beta$ probability for each query (Thm. A.3). This property stems from how we calibrate Laplace budget ϵ to α and β. This is function CALIBRATEBUDGET in Alg. 1 (lines 6-7). For n datapoints, setting $\epsilon = \frac{4\ln(1/\beta)}{n\alpha}$ ensures that each query is answered with error at most α with probability $1-\beta$. **(G3)** *Worst-case convergence:* We prove that PMW-Bypass has similar worst-case convergence as PMW. If $lr/\alpha < \tau \leq 1/2$, then w.h.p. PMW-Bypass needs to perform at most $\frac{\ln|\mathcal{X}|}{lr(\tau\alpha - lr)/2}$ updates (Thm. A.4). PMW-Bypass converges roughly $1/2\tau$ times slower than PMW with comparable parameters. In §5.2, we evaluate an empirical measure of convergence and confirm that PMW and PMW-Bypass converge similarly in practice.

4.3 Tree-Structured PMW-Bypass

We now switch to the partitioned-database use cases, focusing on time-based partitions, as in timeseries databases, whether static or dynamic. Rather than accessing the entire database, analysts tend to query specific time windows, such as requesting the Covid positivity rate over the past week, or the fraction of minors diagnosed with Covid in the two weeks following school reopening. This allows the opportunity to leverage DP's parallel composition: the database is partitioned by time (say a week's data goes in one partition), and privacy budget is consumed at the partition level. Queries can run at finer or coarser granularity, but they will consume privacy against the partition(s) containing the requested data.

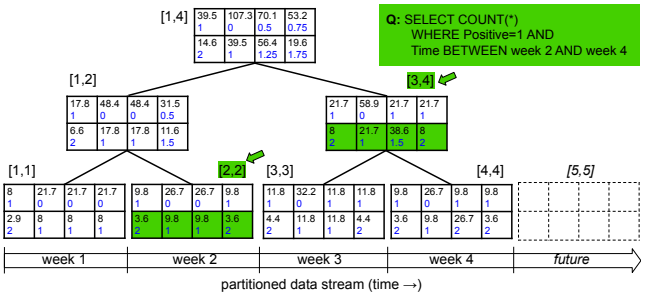


Fig. 5. Example of tree-structured histograms.

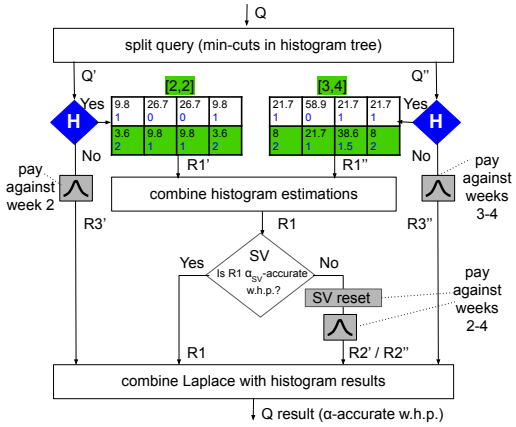


Fig. 6. Tree-structured PMW-Bypass design.

With this approach, a system can answer more queries under a fixed global (ϵ_G, δ_G)-DP guarantee compared to not partitioning [38, 34, 42, 51]. We implement support for partitioning and parallel composition in Turbo through a new caching object called a *tree-structured PMW-Bypass*.

Example. Fig. 5 shows an extension of the running example in §4.1, with the database partitioned by week. Denote n_i the size of each partition. A new query, Q, asks for the positivity rate over the past three weeks. How should we structure the histograms we maintain to best answer this query? One option would be to maintain *one histogram per partition* (i.e., just the leaves in the figure). To resolve Q, we query the histograms for weeks 2, 3, 4. Assume the query results in an update. Then, we need to update histograms, computing the answer with DP within our α error tolerance. Updating histograms for weeks 2, 3, and 4 requires querying the result for each of them with parallel composition. Given that $\text{Var}[\text{Laplace}(1/n\epsilon)] = 2/n\epsilon$, for week 4 for instance, we need noise scaled to $1/n_4\epsilon$. Thus, we consume a fairly large ϵ for an accurate query to compensate for the smaller n_4.

Instead, our approach is to maintain a *binary-tree-structured set of histograms*, as shown in Fig. 5. For each partition, but also for a binary tree growing from the partitions, we maintain a separate histogram. To resolve Q, we split the query into two sub-queries, one running on the histogram for week 2 ([2,2]) and the other running on the histogram for the range week 3 to week 4 ([3,4]). That last sub-query would then run on a larger dataset of size $n_3 + n_4$, requiring a smaller budget consumption to reach the target accuracy.

Design. Fig. 6 shows our design. Given a query Q, we split it into sub-queries based on the histogram tree, applying the min-cuts algorithm to find the smallest set of nodes in the tree that covers the requested partitions. In our example, this gives two sub-queries, Q' and Q'', running on histograms [2,2] and [3,4], respectively. For each sub-query, we use our heuristic to decide whether to use the histogram or invoke Laplace directly. If both histograms are "ready," we compute their estimations and combine them into one result, which we test with an SV against an accuracy goal. In our example, there are only two sub-queries, but in general there can be more, some of which will use direct Laplace while others use histograms. We adjust the SV's accuracy target to an $(\alpha_{SV}, \beta_{SV})$ calibrated to the aggregation that we will need to do among the results of these different mechanisms. We pay for any Laplace's and SV resets against the queried data partitions and finally combine Laplace results with histogram-based results.

Guarantees. (G1) *Privacy* and **(G2)** *accuracy* are unchanged (Thm. A.5, A.6 in Supplementary Material). **(G3)** *Worst-case convergence:* For T partitions, if $lr/\alpha < \tau \leq 1/2$, then w.h.p. we perform at most $\frac{(\lceil \log T \rceil + 1) \ln |\mathcal{X}|}{\eta(\tau\alpha - \eta)/2}$ updates (Thm. A.8).

4.4 Histogram Warm-Start

An opportunity exists in streams to warm-start histograms from previously trained ones to converge faster. Prior work on PMW initialization [37] only justifies using a public dataset close to the private dataset to learn a more informed initial value for histogram bins than a uniform prior. We prove that warm-starting a histogram by copying an entire, trained histogram preserves the worst-case convergence. In Turbo, we use two procedures: for new leaf histograms, we copy the previous partition's leaf node; for non-leaf histograms, we take the average of children histograms.

Guarantees. (G1) *Privacy* and **(G2)** *accuracy* guarantees are unchanged. **(G3)** *Worst-case convergence:* If there exists $\lambda \geq 1$ such that the initial histogram h_0 in Alg. 1 satisfies $\forall x \in \mathcal{X}, h_0(x) \geq \frac{1}{\lambda|\mathcal{X}|}$, then we show that each PMW-Bypass converges, albeit at a slower pace (Thm. A.9 in Supplementary Material). The same properties hold for the tree.

5 Evaluation

We prototype Turbo using TimescaleDB as the underlying database and Redis for storing the histograms and exact-cache. Using two public timeseries datasets – Covid and CitiBike – we evaluate Turbo in the three use cases from §3.2. Each use case lets us do *system-wide evaluation*, answering the critical question: *Does Turbo significantly improve privacy budget consumption compared to reasonable baselines for each use case?* This corresponds to evaluating our §3.1 design goals **(G5)** and **(G6)**. In addition, each setting lets us evaluate a different set of caching objects and mechanisms:

(1) Non-partitioned database: We configure Turbo with a single PMW-Bypass and Exact-Cache, letting us evaluate the PMW-Bypass object, including its empirical convergence and the impact of its heuristic and learning rate parameters.

(2) Partitioned static database: We partition the datasets by time (one partition per week) and configure Turbo with the tree-structured PMW-Bypass and Exact-Cache. This lets us evaluate the tree-structured cache.

(3) Partitioned streaming database: We configure Turbo with the tree-structured PMW-Bypass, Exact-Cache, and histogram warm-up, letting us evaluate warm-up.

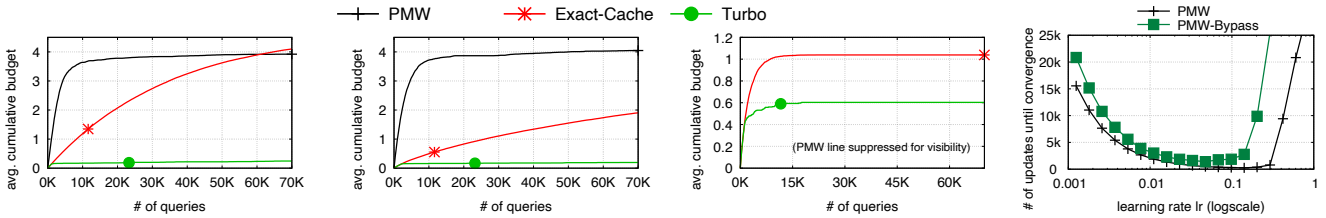
As highlighting, our results show that PMW-Bypass unleashes the power of PMW, enhancing privacy budget consumption for linear queries well beyond the conventional approach of using an exact-match cache (goal **(G5)**). Moreover, Turbo as a whole seamlessly applies to multiple settings, with its novel tree-structured PMW-Bypass structure scoring significant benefit in the case of timeseries and other workloads where range queries pervade, hence the database can be partitioned to leverage parallel composition (goal **(G6)**). Configuration of our objects and mechanisms is straightforward (goal **(G7)**), and we tune them based on empirical convergence rather than theoretical convergence, boosting their practical effectiveness (goal **(G4)**). Furthermore, while our primary focus is on privacy budget performance, we also provide a basic runtime and memory evaluation. Although PMW-Bypass and Turbo perform reasonably for our datasets, further research is necessary to prepare them for larger-domain data.

5.1 Methodology

For each dataset, we create query workloads by (1) generating a pool of linear queries and (2) sampling queries from this pool based on a parameterized Zipfian distribution. We provide details on the datasets and workload generation, followed by a definition of the relevant evaluation metrics. It is noteworthy that Covid uses a completely synthetic query pool, while CitiBike leverages a query pool based on real-user queries from prior CitiBike analyses. We thus use the former as a microbenchmark and the latter as a macrobenchmark.

Covid. *Dataset:* We take a California dataset of Covid-19 tests from 2020, which provides daily *aggregate information* of the number of Covid tests and their positivity rates for various demographic groups defined by age × gender × ethnicity. By combining this data with US Census data, we generate a *synthetic dataset* that contains $n = 50,426,600$ per-person test records, each with the date and four attributes: positivity, age, gender, and ethnicity. These attributes have domain sizes of 2, 4, 2 and 8, respectively, resulting in a dataset domain size of $N = 128$. The dataset spans 50 weeks, meaning that in partitioning cases, we will have up to 50 partitions. *Query pool:* We create a synthetic and very rich pool of correlated queries comprising all possible count queries that can be posed on Covid. This gives $34,425$ unique queries, plenty for us to microbenchmark Turbo in different settings.

CitiBike. *Dataset:* We take a dataset of NYC bike rentals from 2018-2019, which includes information about individual rides, such as start/end date, start/end geo-location, and renter's gender and age. The original data is too granular



(a) Turbo on Covid $k_{\text{zipf}} = 0$ (b) Turbo on Covid $k_{\text{zipf}} = 1$ (c) Turbo on CitiBike $k_{\text{zipf}} = 0$ (d) Empirical convergence

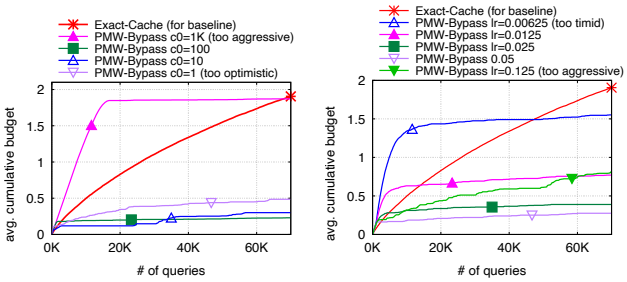
Fig. 7. Non-partitioned database: (a-c) system-wide evaluation (Question 1); (d) empirical convergence for PMW-Bypass vs. PMW (Question 2). (a-c) Turbo, instantiated with one PMW-Bypass and Exact-Cache, significantly improves budget consumption compared to both baselines. (d) Uses Covid $k_{\text{zipf}} = 1$. PMW-Bypass has similar empirical convergence to PMW, and both converge faster with much larger lr than anticipated by worst-case convergence.

with 4,000 geo-locations and 100 ages, making it impractical for PMWs. Since all the real-user analyses we found consider the data at coarser granularity (e.g. broader locations and age brackets), we group geo-locations into ten neighborhoods and ages into four brackets. This yields a dataset with $n = 21,096,261$ records, domain size $N = 604,800$, and spanning 50 weeks. *Query pool:* We collect a set of pre-existing CitiBike analyses created by various individuals and made available on Public Tableau [55]. An example is here [53]. We extract 30 distinct queries, most containing 'GROUP BY' statements that can be broken down into multiple *primitive queries* that can interact with Turbo histograms. This gives us a 2,485-query pool, which is smaller than Covid's but more realistic, so we use it as macrobenchmark.

Workload generation. As is customary in caching literature [14, 60, 6], we use a Zipfian distribution to control the skewness of query distribution, which affects hit rates in the exact-match cache. We generate queries with a Poisson process where each new query is independently sampled from a Zipf distribution over a pool of Q queries. The PDF of the Zipf distribution is: $f_{k_{\text{zipf}}}(x) \propto x^{-k_{\text{zipf}}}$ for query number $x \in [1, Q]$ where $k_{\text{zipf}} \geq 0$ is the parameter that controls skewness. We evaluate with several k_{zipf} values but report only results for $k_{\text{zipf}} = 0$ (uniform) and $k_{\text{zipf}} = 1$ (skewed) for Covid. For CitiBike, we evaluate only for $k_{\text{zipf}} = 0$ to avoid reducing the small query pool further with skewed sampling.

Metrics. • *Average cumulative budget*: the average budget consumed across all partitions. • *Systems metrics:* traditional runtime, process RAM. • *Empirical convergence*: We periodically evaluate the quality of Turbo's histogram by running a validation workload sampled from the same query pool. We measure the accuracy of the histogram as the fraction of queries that are answered with error $\geq \alpha/2$ by the histogram. We define *empirical convergence* as the number of histogram updates necessary to reach 90% validation accuracy.

Default parameters. Unless stated otherwise, we use the following parameter values: privacy ($\epsilon_G = 10, \delta_G = 0$); accuracy ($\alpha = 0.05, \beta = 0.001$); for Covid: {learning rate lr starts from 0.25 and decays to 0.025, heuristic ($C_0 = 100, S_0 = 5$), external updates $\tau = 0.05$}; for CitiBike: {learning rate $lr = 0.5$, heuristic ($C_0 = 5, S_0 = 1$), external updates $\tau = 0.01$}.

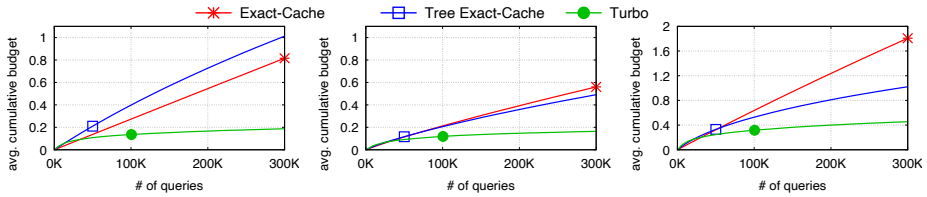


(a) Impact of heuristic C_0 ($S_0 = 1$) (b) Impact of learning rate lr

Fig. 8. Impact of parameters (Question 3). Uses Covid $k_{\text{zipf}} = 1$. Being too optimistic or pessimistic about the histogram's state (a), or too aggressive or timid in learning from each update (b), give poor performance.

5.2 Use Case (1): Non-partitioned Database

System-wide evaluation. *Question 1: In a non-partitioned database, does Turbo significantly improve privacy budget consumption compared to vanilla PMW and a simple Exact-Cache?* Fig. 7(a)-7(c) show the cumulative privacy budget used by three workloads as they progress to $70K$ queries. Two workloads correspond to Covid, one uniform ($k_{\text{zipf}} = 0$) and one skewed ($k_{\text{zipf}} = 1$), and one uniform workload for CitiBike. Turbo surpasses both baselines across all three workloads. The improvement is enormous when compared to vanilla PMW: $15.9 - 37.4\times$! PMW's convergence is rapid but consumes lots of privacy; Turbo uses little privacy during training and then executes queries for free. Compared to just an Exact-Cache, the improvement is less dramatic but still significant. The greatest improvement over Exact-Cache is seen in the uniform Covid workload: $16.7\times$ (Fig. 7(a)). Here, queries are relatively unique, resulting in low hit rate for the Exact-Cache. That hit rate is higher for the skewed workload (Fig. 7(b)), leaving less room for improvement for Turbo: $9.7\times$ better than Exact-Cache. For CitiBike (Fig. 7(c)), the query pool is much smaller ($< 2.5K$ queries), resulting in many exact repetitions in a large workload, even if uniform. Nevertheless, Turbo gives a $1.7\times$ improvement over Exact-Cache. And in this workload, Turbo outperforms PMW by $37.4\times$ (omitted from figure for visualization reasons). Overall, then, Turbo significantly reduces privacy budget consumption in non-partitioned databases, achieving $1.7 - 15.9\times$ improvement over the best baseline for each workload (goal **(G5)**).



(a) Turbo on Covid $k_{\text{zipf}} = 0$ (b) Turbo on Covid $k_{\text{zipf}} = 1$ (c) Turbo on CitiBike $k_{\text{zipf}} = 0$

Fig. 9. Partitioned static database: system-wide evaluation (Question 4). Turbo is instantiated with tree-structured PMW-Bypass and Exact-Cache. Turbo significantly improves budget consumption compared to both a single Exact-Cache and a tree-structured set of Exact-Caches.

PMW-Bypass evaluation. Using Covid $k_{\text{zipf}} = 1$, we microbenchmark PMW-Bypass to understand the behavior of this key Turbo component. *Question 2: Does PMW-Bypass converge similarly to PMW in practice?* Through theoretical analysis, we have shown that PMW-Bypass achieves similar worst-case convergence to PMW, albeit with inferior parameters (§4.2). Fig. 7(d) compares the *empirical convergence* (defined in §5.1) of PMW-Bypass vs. PMW, as a function of the learning rate lr, which impacts the convergence speed. We make three observations, two of which agree with theory, and the last differs. First, the results confirm the theory that (1) PMW-Bypass and PMW converge similarly, but (2) for "good" values of lr, vanilla PMW converges slightly faster: e.g., for $lr = 0.025$, PMW-Bypass converges after 1853 updates, while PMW after 944. Second, as theory suggests, very large values of lr (e.g., $lr \geq 0.4$) impede convergence in practice. Third, although theoretically, $lr = \alpha/8 = 0.00625$ is optimal for worst-case convergence, and it is commonly hard-coded in PMW protocols [56], we find that empirically, larger values of lr (e.g., $lr = 0.05$, which is 8× larger) result in substantially faster convergence. This is true for both PMW and PMW-Bypass, and across all our workloads. This justifies the need to adapt and tune mechanisms based on not only theoretical but also empirical behavior (goal **(G4)**).

Question 3: How do PMW-Bypass heuristic, learning rate, and external update parameters impact consumed budget? We experimented with all parameters and found that the two most impactful are (a) C_0, the initial threshold for the number of updates each bin involved in a query must have received to use the histogram, and (b) the learning rate. Fig. 8 shows their effects. *Heuristic C_0 (Fig. 8(a)):* Higher C_0 results in a more pessimistic assessment of histogram readiness. If it's too pessimistic ($C_0 = 1K$), PMW is never used, so we follow a direct Laplace. If it's too optimistic ($C_0 = 1$), errors occur too often, and the histogram's training overpays. $C_0 = 100$ is a good value for this workload. *Learning rate* lr *(Fig. 8(b)):* Higher lr leads to more aggressive learning from each update. Both too aggressive ($lr = 0.125$) and too timid ($lr = 0.00625$) learning slow down convergence. Good values hover around $lr = 0.025$. Overall, only a few parameters affect performance, and even for those, performance is relatively stable around good values, making them easy to tune (goal **(G7)**).

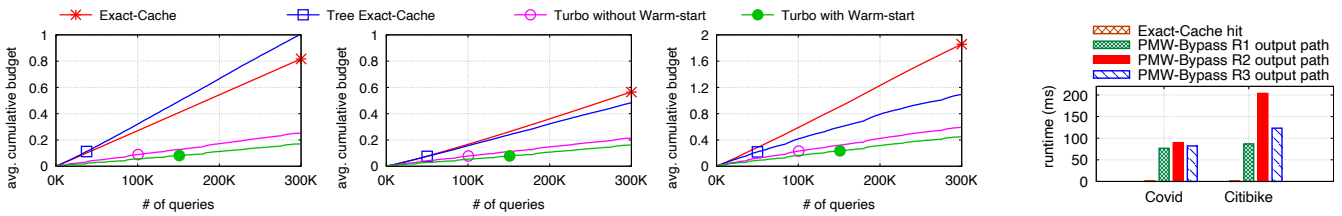
5.3 Use Case (2): Partitioned Static Database

System-wide evaluation. *Question 4: In a partitioned static database, does Turbo significantly improve privacy budget consumption, compared to a single Exact-Cache and a tree-structured set of Exact-Caches?* We divide each database into 50 partitions and select a random contiguous window of 1 to 50 partitions for each query. Fig. 9(a)-9(c) show the average budget consumed per partition up to 300K queries. In contrast to the static use case, Turbo can support more queries under the fixed ($\epsilon_G = 10$)-DP guarantee since each query only consumes privacy from the accessed partitions due to parallel composition. Turbo further boosts privacy budget consumption by $2.2 - 4.4\times$ compared to the best-performing baseline for each workload, demonstrating its effectiveness as a caching strategy for the static partitioned use case.

Tree structure evaluation. *Question 5: When is does the tree structure for histograms outperform a flat structure that maintains one histogram per partition?* We vary the average size of the windows requested by queries from 1 to 50 partitions based on a Gaussian distribution with std-dev 5. We find the tree structure for histograms is beneficial when queries tend to request more partitions (25 partitions or more). Because the tree structure maintains more histograms than the flat structure, it fragments the query workload more, resulting in fewer histogram updates per histogram and more use of direct-Laplace. The tree's advantage in combining fewer results makes up for this privacy overhead caused by histogram maintenance when queries tend to request larger windows of partitions, while the linear structure is more justified when queries tend to request smaller windows of partitions.

5.4 Use Case (3): Partitioned Streaming Database

System-wide evaluation. *Question 6: In streaming databases partitioned by time, does Turbo significantly improve privacy budget consumption compared to baselines? Does warm-start help?* Fig. 10(a)-10(c) show Turbo's privacy budget consumption compared to the baselines. The experiments simulate a streaming database, where partitions arrive over time and queries request the latest P partitions, with P chosen uniformly at random between 1 and the number of available partitions. Turbo outperforms both baselines significantly for all workloads, particularly when warm-start is enabled. Without warm-start, Turbo improves performance by $1.8 - 3.2\times$ at the end of the workload. With warm-start, Turbo gives $2.4 - 4.8\times$ improvement over the best baseline for each workload, showing its effectiveness for the streaming use case. This concludes our evaluation across use cases (goal **(G6)**).



(a) Turbo on Covid $k_{\text{zipf}} = 0$ (b) Turbo on Covid $k_{\text{zipf}} = 1$ (c) Turbo on CitiBike $k_{\text{zipf}} = 0$ (d) Runtime evaluation

Fig. 10. (a-c) Partitioned streaming database: system-wide consumed budget (Question 6); (d) PMW-Bypass runtime in non-partitioned setting (Question 7). (a-c) Turbo is instantiated with tree-structured PMW-Bypass and Exact-Cache, with and without warm-start. (d) Uses Covid $k_{\text{zipf}} = 1$ and one Exact-Cache and PMW-Bypass. Shows execution runtime for different execution paths. Most expensive is when the SV test fails.

5.5 Runtime and Memory Evaluation

Question 7: What are Turbo's runtime and memory bottlenecks? We evaluate Turbo's runtime and memory consumption to identify areas for efficiency and scalability improvements. Fig. 10(d) shows the average runtime of Turbo's main execution paths in a non-partitioned database, where the Exact-Cache hit path is the cheapest and the other paths are more expensive. Histogram operations are the bottlenecks in CitiBike due to the larger domain size (N), while query execution in TimescaleDB is the bottleneck in Covid due to the larger database size (n). The $R1$ path is similar across the two datasets because their distinct bottlenecks compensate. Failing the SV check (output path $R2$) is the costliest path for both datasets due to the extra operations needed to update the heuristic's per-bin thresholds. We also conduct an end-to-end experiment in a partitioned streaming database and find that TimescaleDB remains the runtime bottleneck for Covid, while histogram operations remain the bottleneck for CitiBike. Finally, we report Turbo's memory consumption: 5.21MB for Covid and 1.43GB for CitiBike. We conclude that more work remains to scale Turbo for larger data domain sizes, including building histograms on demand by replaying history, storing "cold" bins on disk rather than in memory, partitioning the space into cross-product if some attributes are never queried together, and speeding up histogram estimations with a GPU.

6 Related Work

The theoretical DP literature contains a wealth of algorithms relying on synthetic datasets or histograms to answer linear queries, such as SmallDB [10], FEM [57], RAP [7]. Particularly close to PMW [26] are MWEM [25], PMW-Pub [37] and PMWG [16]. Apart from PMW and PMWG, all these methods assume that queries are *known upfront*, a big limitation in real systems. The same applies to the matrix mechanism [36, 41], another approach to efficiently answer a batch of correlated queries. PMWG extends PMW to work on dynamic "growing" databases, but assumes that each query requests the *entire database*. This precludes having unbounded timestamps as an attribute and does not allow parallel composition. Other algorithms focus on continuously releasing specific statistics, such as the streaming counter [29] that inspired our tree structure. Cardoso et al. [11] generalize this algorithm to continuously release top-k queries instead of a single count. These works do not support arbitrary linear queries, and answer predefined queries at every time step, while we only pay budget for queries that are actually asked.

On the systems side, the majority of general-purpose DP SQL engines, such as PINQ [42], Flex [31], GoogleDP [59] or Tumult Analytics [9], only offer basic DP primitives (e.g. Laplace and Gaussian mechanisms) or DP interfaces for common operations (e.g. counts or quantiles). They have *no caching capabilities* or advanced mechanisms that jointly answer correlated queries. Domain-specific DP analytics engines, such as LinkedIn's Audience Engagements API [48], Orchard [50] or Plume [5] have the same limitations. OpenDP's SmartNoise SQL engine [45] can generate synthetic datasets to answer many correlated queries using the MWEM algorithm, but this approach does not apply to online queries, data streams, or caching. Chorus [30], Uber's DP SQL engine, implements the SV protocol and a simple version of PMW. However its PMW operates on a single attribute and does not support the streaming of data. PrivateSQL [32] uses a more advanced mechanism to simultaneously answer many known-upfront queries, but does not handle online queries or data. Pioneer [46] provides a form of Exact-Cache. CacheDP [40] combines an Exact-Cache with repeated calls to the Matrix Mechanism over batches of queries. It does not scale well to datasets with more than a trivial number of dimensions, nor does it support data streams or parallel composition.

7 Conclusions

Turbo is a caching layer for differentially-private databases that increases the number of linear queries that can be answered accurately with a fixed privacy guarantee. It employs a PMW, which learns a histogram representation of the dataset from prior query results and can answer future linear queries at no additional privacy cost once it has converged. To enhance the practical effectiveness of PMWs, we bypass them during the privacy-expensive training phase and only switch to them once they are ready. This transforms PMWs from "utterly ineffective" to "very effective" compared to simpler cache designs. Moreover, Turbo includes a tree-structured set of histograms that supports timeseries and streaming use cases, taking advantage of fine-grained privacy budget accounting and warm-starting opportunities to further enhance the number of answered queries.

References

[1] Citibike system data. https://www.citibikenyc.com/system-data, 2018.

[2] NOT-OD-17-110: Request for Comments: Proposal to Update Data Management of Genomic Summary Results Under the NIH Genomic Data Sharing Policy, Apr. 2023. [Online; accessed 17. Apr. 2023].

[3] 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*, (Special Issue 2), 2022.

[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European conference on computer systems*, pages 29–42, 2013.

[5] K. Amin, J. Gillenwater, M. Joseph, A. Kulesza, and S. Vassilvitskii. Plume: Differential Privacy at Scale. *arXiv*, Jan. 2022.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[7] S. Aydöre, W. Brown, M. Kearns, K. Kenthapadi, L. Melis, A. Roth, and A. A. Siva. Differentially private query release through adaptive projection. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 457–467. PMLR, 2021.

[8] S. Bavadekar, A. Dai, J. Davis, D. Desfontaines, I. Eckstein, K. Everett, A. Fabrikant, G. Flores, E. Gabrilovich, K. Gadepalli, S. Glass, R. Huang, C. Kamath, D. Kraft, A. Kumok, H. Marfatia, Y. Mayer, B. Miller, A. Pearce, I. M. Perera, V. Ramachandran, K. Raman, T. Roessler, I. Shafran, T. Shekel, C. Stanton, J. Stimes, M. Sun, G. Wellenius, and M. Zoghi. Google COVID-19 Search Trends Symptoms Dataset: Anonymization Process Description (version 1.0). *arXiv*, Sept. 2020.

[9] S. Berghel, P. Bohannon, D. Desfontaines, C. Estes, S. Haney, L. Hartman, M. Hay, A. Machanavajjhala, T. Magerlein, G. Miklau, A. Pai, W. Sexton, and R. Shrestha. Tumult analytics: a robust, easy-to-use, scalable, and expressive framework for differential privacy. *CoRR*, abs/2212.04133, 2022.

[10] A. Blum, K. Ligett, and A. Roth. A learning theory approach to non-interactive database privacy. In C. Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 609–618. ACM, 2008.

[11] A. R. Cardoso and R. Rogers. Differentially private histograms under continual observation: Streaming selection into the unknown. In G. Camps-Valls, F. J. R. Ruiz, and I. Valera, editors, *International Conference on Artificial Intelligence and Statistics, AISTATS 2022, 28-30 March 2022, Virtual Event*, volume 151 of *Proceedings of Machine Learning Research*, pages 2397–2419. PMLR, 2022.

[12] T.-H. H. Chan, E. Shi, and D. Song. Private and continual release of statistics. *ACM Trans. Inf. Syst. Secur.*, 14(3), nov 2011.

[13] A. Cohen and K. Nissim. Linear program reconstruction in practice. *Journal of Privacy and Confidentiality*, 2020.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[15] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Oct. 2001.

[16] R. Cummings, S. Krehbiel, K. A. Lai, and U. Tantipongpipat. Differential privacy for growing databases. *CoRR*, abs/1803.06416, 2018.

[17] Y.-A. De Montjoye, C. A. Hidalgo, M. Verleysen, and V. D. Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 2013.

[18] I. Dinur and K. Nissim. Revealing information while preserving privacy. 2003.

[19] J. Dong, A. Roth, and W. J. Su. Gaussian differential privacy. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 2022.

[20] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. of the Theory of Cryptography Conference (TCC)*, 2006.

[21] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[22] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 2014.

[23] S. R. Ganta, S. Kasiviswanathan, and A. Smith. Composition attacks and auxiliary information in data privacy. 2008.

[24] S. Garfinkel, J. M. Abowd, and C. Martindale. Understanding database reconstruction attacks on public data. *Communications of the ACM*, 2019.

[25] M. Hardt, K. Ligett, and F. Mcsherry. A simple and practical algorithm for differentially private data release. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[26] M. Hardt and G. N. Rothblum. A multiplicative weights mechanism for privacy-preserving data analysis. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 61–70, 2010.

[27] M. Hardt and G. N. Rothblum. A multiplicative weights mechanism for privacy-preserving data analysis. In *Symposium on Foundations of Computer Science*, 2010.

[28] N. Homer, S. Szelinger, M. Redman, D. Duggan, W. Tembe, J. Muehling, J. V. Pearson, D. A. Stephan, S. F. Nelson, and D. W. Craig. Resolving individuals contributing trace amounts of DNA to highly complex mixtures using high-density SNP genotyping microarrays. *PLoS Genetics*, 2008.

[29] T.-H. Hubert Chan, E. Shi, and D. Song. Private and continual release of statistics. In S. Abramsky, C. Gavoille, C. Kirchner, F. Meyer auf der Heide, and P. G. Spirakis, editors, *Automata, Languages and Programming*, pages 405–417, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[30] N. Johnson, J. P. Near, J. M. Hellerstein, and D. Song. Chorus: a programming framework for building scalable differential privacy mechanisms. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 535–551, 2020.

[31] N. M. Johnson, J. P. Near, and D. Song. Towards practical differential privacy for SQL queries. *Proc. VLDB Endow.*, 11(5):526–539, 2018.

[32] I. Kotsogiannis, Y. Tao, X. He, M. Fanaeepour, A. Machanavajjhala, M. Hay, and G. Miklau. Privatesql: A differentially private sql query engine. *Proc. VLDB Endow.*, 12(11):13711384, jul 2019.

[33] M. Lécuyer. Practical Privacy Filters and Odometers with Rnyi Differential Privacy and Applications to Differentially Private Deep Learning. In *arXiv*, 2021.

[34] M. Lécuyer, R. Spahn, K. Vodrahalli, R. Geambasu, and D. Hsu. Privacy Accounting and Quality Control in the Sage Differentially Private ML Platform. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[35] M. Lecuyer, R. Spahn, K. Vodrahalli, R. Geambasu, and D. Hsu. Privacy accounting and quality control in the sage differentially private ML platform. Online Supplements (also available on https://arxiv.org/abs/1909.01502), 2019.

[36] C. Li, G. Miklau, M. Hay, A. McGregor, and V. Rastogi. The matrix mechanism: optimizing linear counting queries under differential privacy. *VLDB J.*, 24(6):757–781, 2015.

[37] T. Liu, G. Vietri, T. Steinke, J. Ullman, and S. Wu. Leveraging public data for practical private query release. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine*

Learning, volume 139 of *Proceedings of Machine Learning Research*, pages 6968–6977. PMLR, 18–24 Jul 2021.

[38] T. Luo, M. Pan, P. Tholoniat, A. Cidon, R. Geambasu, and M. Lécuyer. Privacy budget scheduling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 55–74. USENIX Association, July 2021.

[39] M. Lyu, D. Su, and N. Li. Understanding the sparse vector technique for differential privacy. *Proc. VLDB Endow.*, 10(6):637–648, 2017.

[40] M. Mazmudar, T. Humphries, J. Liu, M. Rafuse, and X. He. Cache me if you can: Accuracy-aware inference engine for differentially private data exploration. *Proc. VLDB Endow.*, 16(4):574–586, 2022.

[41] R. McKenna, G. Miklau, M. Hay, and A. Machanavajjhala. Optimizing error of high-dimensional statistical queries under differential privacy. *Proc. VLDB Endow.*, 11(10):1206–1219, 2018.

[42] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 19–30. ACM, 2009.

[43] I. Mironov. Rényi Differential Privacy. In *Computer Security Foundations Symposium (CSF)*, 2017.

[44] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2008.

[45] opendp. smartnoise-sdk, Apr. 2023. [Online; accessed 10. Apr. 2023].

[46] S. Peng, Y. Yang, Z. Zhang, M. Winslett, and Y. Yu. Query optimization for differentially private data management systems. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1093–1104, 2013.

[47] R. Rogers. A differentially private data analytics API at scale. In *2020 USENIX Conference on Privacy Engineering Practice and Respect (PEPR 20)*. USENIX Association, Oct. 2020.

[48] R. Rogers, S. Subramaniam, S. Peng, D. Durfee, S. Lee, S. K. Kancha, S. Sahay, and P. Ahammad. Linkedin's audience engagements api: A privacy preserving data analytics system at scale. *arXiv preprint arXiv:2002.05839*, 2020.

[49] R. M. Rogers, A. Roth, J. Ullman, and S. Vadhan. Privacy odometers and filters: Pay-as-you-go composition. 2016.

[50] E. Roth, H. Zhang, A. Haeberlen, and B. C. Pierce. Orchard: Differentially private analytics at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1065–1081. USENIX Association, 2020.

[51] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[52] J. Smith, H. J. Asghar, G. Gioiosa, S. Mrabet, S. Gaspers, and P. Tyler. Making the most of parallel composition in differential privacy. *Proc. Priv. Enhancing Technol.*, 2022(1):253–273, 2022.

[53] Unknown. Citibike tableau story. https://public.tableau.com/app/profile/james.jeffrey/viz/CitiBikeRideAnalyzer/CitiBikeRdeAnalyzer. Accessed: 2023-04-13.

[54] Unknown. Real world dp use-cases. https://desfontain.es/privacy/real-world-differential-privacy.html. Accessed: 2023-04-13.

[55] Unknown. Tableau. https://public.tableau.com/app/discover. Accessed: 2023-04-13.

[56] S. Vadhan. The Complexity of Differential Privacy. In *Tutorials on the Foundations of Cryptography*, pages 347–450. Springer, Cham, Switzerland, Apr. 2017.

[57] G. Vietri, G. Tian, M. Bun, T. Steinke, and Z. S. Wu. New oracle-efficient algorithms for private synthetic data release. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 9765–9774. PMLR, 2020.

[58] L. Wasserman and S. Zhou. A statistical framework for differential privacy. *Journal of the American Statistical Association*, 2010.

[59] R. J. Wilson, C. Y. Zhang, W. Lam, D. Desfontaines, D. Simmons-Marengo, and B. Gipson. Differentially private sql with bounded user contribution. *Proceedings on Privacy Enhancing Technologies*, 2020(2):230–250, 2020.

[60] J. Yang, Y. Yue, and K. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 191–208, 2020.

[61] Y. Zhu and Y.-X. Wang. Improving Sparse Vector Technique with Renyi Differential Privacy. *Advances in Neural Information Processing Systems*, 33:20249–20258, 2020.