[Optimizing Privacy Budget Management in Differentially Private Systems]

[Kelly Kostopoulou]

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2025

Abstract

[Optimizing Privacy Budget Management in Differentially Private Systems]

[Kelly Kostopoulou]

Modern computing systems increasingly operate under stringent resource constraints—whether in the form of traditional hardware resources like CPU and memory, or novel, non-traditional resources such as user privacy. This thesis explores systems and algorithmic techniques for efficient resource management in two distinct domains: (1) the emerging field of privacy-preserving data analytics, where privacy itself becomes a scarce and quantifiable resource to be allocated; and (2) distributed transaction processing, where lock-based contention and commit coordination determine throughput under high load.

In the first part of the thesis, we present a series of systems—DPack, Turbo, and Cookie Monster—that treat differential privacy budgets as consumable system resources. Each system targets a different layer of the privacy-preserving computing stack, from workload schedulers to database query caches to browser-based advertising measurement. Despite the diversity of applications, they all aim to improve the efficiency with which private data can be used, supporting more useful computation under fixed privacy guarantees.

The second part of the thesis shifts domains to distributed databases and introduces Sangria, an adaptive protocol that dynamically switches between conservative and pipelined commit strategies based on runtime conditions. Although this work is unrelated to privacy, it shares a common methodological theme: maximizing efficiency under contention and resource pressure.

Together, these contributions illustrate the importance—and the diversity—of efficient resource allocation across modern computing systems, from privacy-aware data processing to classical transaction management.

Table of Contents

Acknowledgments xii

Chapter 1: Introduction 1

Chapter 2: DPack: Efficiency-Oriented Privacy Budget Scheduling 3

2.1 Overview 3

2.2 Introduction 3

2.3 Background 6

2.3.1 Threat Model 6

2.3.2 DP Background 7

2.3.3 Privacy Scheduling Background 10

2.4 Efficiency-Oriented Privacy Scheduling 11

2.4.1 Efficient Scheduling with Traditional DP 12

2.4.2 Efficient Scheduling Under RDP Accounting 15

2.4.3 DPack Algorithm 19

2.4.4 Adapting to the Online Case 21

2.5 Applicability 22

2.6 Implementation 23

2.7 Evaluation 24

2.7.1 Methodology 25
2.7.2 Offline Microbenchmark (Q1, Q2) 26
2.7.3 Online Plausible Workload (Q3) 28
2.7.4 Kubernetes Implementation Evaluation (Q4) 32
2.8 Related Work 34
2.9 Conclusions 35

Chapter 3: Turbo: Effective Caching for Differentially-Private Databases 36
3.1 Overview 36
3.2 Introduction 36
3.3 Background 40
3.4 Turbo Overview 42
3.4.1 Design Goals 43
3.4.2 Use Cases 44
3.4.3 Turbo Architecture 45
3.5 Detailed Design 48
3.5.1 Notation 48
3.5.2 Running Example 49
3.5.3 PMW-Bypass 51
3.5.4 Tree-Structured PMW-Bypass 57
3.5.5 Histogram Warm-Start 59
3.6 Prototype Implementations 60
3.7 Evaluation 65

3.7.1 Methodology 66
3.7.2 Use Case (1): Non-partitioned Database 69
3.7.3 Use Case (2): Partitioned Static Database 72
3.7.4 Use Case (3): Partitioned Streaming Database 73
3.7.5 Runtime and Memory Evaluation 74
3.8 Discussion 75
3.9 Related Work 77
3.10 Conclusion 79

Chapter 4: Cookie Monster: Efficient On-device Budgeting for Differentially-Private Ad-Measurement Systems 80
4.1 Overview 80
4.2 Introduction 80
4.3 Review of Ad-Measurement APIs 83
4.3.1 Example Scenario 83
4.3.2 Ad-Measurement Systems 85
4.3.3 Improvement Opportunity 87
4.4 Cookie Monster Overview 88
4.4.1 Architecture 89
4.4.2 Execution Example 91
4.4.3 Algorithm 92
4.4.4 Bias Implications of IDP 94
4.5 Formal Modeling and Analysis 96
4.5.1 Formal System Model 96

4.5.2 IDP Formulation and Guarantees . . . 100
4.5.3 IDP Optimizations . . . 103
4.6 Chrome Prototype . . . 105
4.7 Evaluation . . . 105
4.7.1 Methodology . . . 105
4.7.2 Microbenchmark Evaluation (Q1) . . . 107
4.7.3 PATCG Evaluation (Q1, Q2) . . . 109
4.7.4 Criteo Evaluation (Q1, Q2) . . . 111
4.7.5 Bias Measurement (Q3) . . . 113
4.8 Related Work . . . 114
4.9 Conclusion . . . 115

Chapter 5: Dances with Locks: An Adaptive Commit Protocol for Distributed Transactions 116
5.1 Overview . . . 116
5.2 Introduction . . . 117
5.3 Background . . . 121
5.4 Dependency Tracking with Resolver . . . 124
5.5 Sangria . . . 127
5.5.1 Overview . . . 128
5.5.2 Coordinator Commit Protocol . . . 129
5.5.3 Participant Prepare Procedure . . . 129
5.5.4 Discussion . . . 130
5.5.5 Adaptive Decision Logic . . . 131

5.5.6 Correctness Guarantees . . . 131
5.6 Evaluation . . . 132
5.6.1 Methodology . . . 132
5.6.2 Workloads . . . 133
5.6.3 Contention vs. Resolver Capacity (Q1) . . . 134
5.6.4 Online Adaptation (Q2) . . . 138
5.6.5 Mixed Workloads (Q3) . . . 140
5.6.6 Resolver Performance (Q4) . . . 142
5.7 Related Work . . . 144
5.8 Future Work . . . 146
5.9 Conclusions . . . 146
Conclusion . . . 147
References . . . 148

List of Figures

2.1 **Example of allocations with basic DP accounting.** Task T_1 requests privacy budget from 3 blocks, B_1, B_2, B_3. Tasks T_2, T_3, T_4 request slightly more privacy budget, but each one from one distinct block: B_1, B_2, B_3, respectively. In (a), DPF sorts these tasks based on their dominant shares: T_1 first (because its dominant share is lower, even though it demands budget from all the blocks), then T_2, T_3, T_4 in arbitrary order. After T_1 is scheduled, there is no more budget for other tasks. Meanwhile, in (b) an efficient scheduler can allocate 3 tasks. 13

2.2 **Example RDP curves and DP translation.** (a) RDP curves for Gaussian, subsampled Gaussian, and Laplace mechanisms, each with std-dev $\sigma = 2$, plus their composition. (b) Translation to $(\epsilon_{DP}, 10^{-6})$-DP. The "best" (i.e., tightest) alpha differs among mechanisms. For composition, best is $\alpha = 6$, giving $\epsilon_{DP} = 5.5$. 16

2.3 Example of allocations with RDP accounting. 18

2.4 **(Q1) DPack under workloads with variable heterogeneity using our microbenchmark.** Global efficiency of the algorithms (y axes) in the offline setting, as heterogeneity increases on the x axes: (a) variation in number of blocks requested, (b) variation in best alphas for the tasks' RDP curves. *Q1 Answer: DPack tracks Optimal closely and significantly outperforms DPF on workloads with high heterogeneity: 0–161% improvement for Fig. 2.4a and 0–67% for Fig. 2.4b.* 27

2.5 **(Q2) Scalability under increasing load from the microbenchmark.** (a) Scheduler runtime and (b) number of allocated tasks, as a function of offered load (x axes). *Q2 Answer: Optimal becomes intractable quickly while DPack and DPF remain practical even at high load.* 28

2.6 **(Q3) Efficiency evaluation on the online Alibaba-DP workload.** Number of allocated tasks as a function of (a) offered load for 90 blocks and (b) available blocks for 60k tasks. *Q3 Answer: Alibaba-DP exhibits sufficient heterogeneity for DPack to present a significant improvement (1.3–1.7×) over DPF.* 30

2.7 **Evaluation on Amazon Reviews workload from [8].** (a) The original synthetic workload exhibits limited heterogeneity, so there is no room for DPack to improve over DPF. (b) Adding randomly selected weights to the tasks creates sufficient heterogeneity for DPack to show an improvement. Global efficiency is measured as the sum of weights of allocated tasks (y axis). 31

2.8 **(Q4) Evaluation on Kubernetes with Alibaba-DP.** DPack has only a modestly higher runtime than DPF, as system-related overheads dominate. In the online setting, scheduling delays are nearly identical across schedulers. 33

3.1 **Turbo architecture.** 46

3.2 **Running example**. (a) Simplified Covid tests dataset with $n = 100$ rows and data domain size $N = 8$ for the two non-time attributes, test outcome P and subject's age bracket A. (b) Two queries that were previously run. (c) State of the histogram as queries are executed. (d) Next query to run. 49

3.3 **Demo experiment.** 52

3.4 **PMW-Bypass.** New components over vanilla PMW are in blue/bold. 53

3.5 **Example of tree-structured histograms.** 57

3.6 **Tree-structured PMW-Bypass.** 59

3.7 **(a) Turbo integration into Tumult. (b) Turbo API.** 60

3.8 **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. 67

3.9 **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), gives poor performance. 69

3.10 **Partitioned static database: system-wide evaluation (Question 5).** 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. 70

3.11 **(a-c) Partitioned streaming database: system-wide consumed budget (Question 7); (d) PMW-Bypass runtime in non-partitioned setting (Question 8).** (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. 73

4.1 **Privacy loss dashboard.** Screenshot from our Chrome implementation of Cookie Monster (minimally edited for visibility). 84

4.2 **Architectures of ad-measurement systems.** Common structure, with a key difference in where attribution and DP budgeting occur: off-device (IPA) vs. on-device (ARA, PAM, Hybrid). 85

4.3 **Cookie Monster architecture and example execution (red overlay).** §4.4.1 describes the architecture and §4.4.2 the example execution. Notation: $@e_1 : I_1$ indicates that Ann's device receives an impression I_1 of a Nike shoe ad from nytimes.com in epoch e_1. Red dotted arrows show the attribution function's search for impressions over epochs $e_1 - e_4$. 89

4.4 **Budget consumption on the microbenchmark.** (a) and (b) show average and maximum budget consumption across all device-epochs, respectively, as a function of the fraction of users that participate per query (knob1); value of knob2 is constant 0.1. (c) and (d) show the same metrics as a function of user impressions per day (knob2); value of knob1 is constant 0.1. 108

4.5 **Budget consumption and query accuracy on the PATCG dataset.** (a) Average budget consumption across all device-epochs as a function of the number of queries submitted by the advertiser. (b) CDF of RMSRE with a 7-day epoch. (c) RMSRE median (horizontal lines), first and third quartiles (boxes), and max/min (top/bottom range markers) as epoch length increases. 109

4.6 **Budget consumption and query accuracy on Criteo.** (a) CDF of per-device average budget consumption across epochs for all devices and advertisers. (b) CDF of RMSREs for a 7-day epoch. (c) RMSRE metrics with varying epoch length (see Fig. 4.5c for format). (d) The same CDF as in (a), but for Criteo++, showing the impact of synthetic impression augmentation on Cookie Monster's performance. . 111

4.7 **Budget consumption and query accuracy with bias measurement on the microbenchmark.** (a) Average budget consumed across all device-epochs. (b) CDF of true RMSRE for executed queries, alongside Cookie Monster's RMSRE estimation from bias measurement (light-purple line). (c) Quartiles of true RMSRE, where queries with error estimate above a given cutoff are rejected by Cookie Monster with bias measurement. 112

5.1 Heatmap illustrating the regimes where each protocol is most effective as a function of workload contention (vertical axis) and Resolver capacity (horizontal axis). Red regions indicate scenarios where pipelining (Pipelined-2PC) outperforms Strict-2PC, while blue regions indicate the opposite. The color intensity reflects the magnitude of the performance advantage. 118

5.2 Strict 2PC vs Pipelined 2PC. (a) In Strict 2PC, locks are held throughout the entire commit protocol, resulting in long lock hold times and increased contention. (b) In Pipelined 2PC, locks are released earlier — immediately after the prepare record is appended to the WAL buffer — allowing subsequent transactions to proceed sooner and reducing contention, but introducing commit-time dependencies that require additional coordination to ensure correctness. 122

5.3 Resolver architecture showing communication between resolver, coordinator, and participants 125

5.4 (Q1) Throughput of the three protocols as a function of workload contention (x-axis: concurrency level) under three different Resolver capacity settings (a) high capacity (no background load), (b) medium capacity (moderate background load), and (c) low capacity (heavy background load). Sangria is able to adapt its behavior based on the Resolver's capacity and workload contention, matching or exceeding the throughput of the baselines in all regimes. 134

5.5 (Q1) YCSB: Throughput comparison as contention increases (by increasing the Zipf Constant) under varying Resolver capacities. 137

5.6 (Q2) Throughput of each protocol as workload contention alternates between low and high phases at runtime, under three different resolver capacities (high, medium, low). Sangria adapts to changing contention, matching or exceeding the best static baseline in each regime. 139

5.7 (Q2) Throughput of each protocol as resolver capacity alternates between high and low phases at runtime, under three different concurrency levels (5, 50, 500). Sangria adapts to changing resolver capacity, matching or exceeding the best static baseline in each regime. 140

5.8 (Q3) Throughput of each protocol under a mixed workload with both high-contention (hot) and low-contention (cold) key regions, across three resolver capacities (high, medium, low). Sangria dynamically applies pipelining for hot keys and strict commit for cold keys, matching or exceeding the best baseline in each region. 141

5.9 (Q4) Cumulative distribution function (CDF) of batch sizes for commit groups formed by the Resolver. (a) Varying Resolver capacity under high contention (concurrency = 500) shows that lower capacity leads to larger batch sizes due to more transactions accumulating before being unblocked. (b) Varying workload contention under maximum Resolver capacity demonstrates that higher concurrency increases batching opportunities, while low contention results in mostly single-transaction commits. 143

List of Tables

2.1 **Workload and methodology of each evaluation question.** 24

2.2 **Efficiency on Kubernetes prototype with Alibaba-DP.** 32

Acknowledgments

Acknowledgements

I gratefully acknowledge the support of the Onassis Foundation, from which I have been a recipient of a scholarship during my doctoral studies. The part of this dissertation related to differential privacy was conducted in close collaboration with my co-first author, Pierre Tholoniat.

Chapter 1: Introduction

Efficiency is a fundamental goal in systems design. As computing environments evolve, the need to make effective use of limited resources remains constant. Traditionally, these resources have included compute, memory, storage, and bandwidth. More recently, new forms of constraints have emerged, such as user privacy, which must be carefully managed in systems that handle sensitive data.

This thesis investigates efficient resource management across two distinct domains:
(a) Differentially-private computing, where privacy loss is modeled as a bounded budget that must be allocated carefully; and (b) Distributed transaction processing, where system throughput depends on how effectively a system handles contention and commit coordination under concurrent access.

The bulk of the thesis focuses on the privacy domain. As organizations seek to extract insights from sensitive datasets, differential privacy (DP) has become the gold standard for protecting individuals against leakage. However, DP comes with a hard constraint: once a dataset's privacy budget is exhausted, no further queries can be answered safely. We develop three systems that address this constraint from different angles:

DPack proposes an efficiency-oriented scheduler that maximizes the number of machine learning models trained under a fixed privacy budget, formulating the problem as a multidimensional knapsack variant.

Turbo introduces a caching mechanism for DP databases that leverages previous query results—via both traditional caching and private multiplicative weights—to answer new queries with little or no additional privacy cost.

Cookie Monster designs a rigorous on-device budgeting scheme for DP-based ad measurement systems, improving both privacy guarantees and utility in modern browsers.

Although varied in implementation, these systems all treat privacy as a scarce system resource, and focus on using it as efficiently as possible.

In contrast, the final chapter of the thesis shifts focus to a very different problem: optimizing distributed commit protocols in transactional systems. While not thematically tied to privacy, this work shares a conceptual alignment with the rest of the thesis. Here, contention for locks and coordination bottlenecks limit system performance. We present Sangria, a distributed commit protocol that dynamically toggles between conservative and pipelined execution strategies based on local contention and resource availability, thereby improving transaction throughput across workload regimes.

Together, the contributions in this thesis underscore a broader theme: how to allocate limited resources—whether privacy budgets or coordination capacity—efficiently and accurately. The systems and algorithms presented here offer new insights into managing modern computing resources.

Chapter 2: DPack: Efficiency-Oriented Privacy Budget Scheduling

2.1 Overview

Machine learning (ML) models can leak information about users, and differential privacy (DP) provides a rigorous way to bound that leakage under a given budget. This DP budget can be regarded as a new type of computing resource in workloads of multiple ML models training on user data. Once it is used, the DP budget is forever consumed. Therefore, it is crucial to allocate it most efficiently to train as many models as possible. This paper presents a scheduler for the privacy resources that optimizes for efficiency. We formulate privacy scheduling as a new type of multidimensional knapsack problem, called *privacy knapsack*, which maximizes DP budget efficiency. We show that privacy knapsack is NP-hard, hence practical algorithms are necessarily approximate. We develop an approximation algorithm for privacy knapsack, *DPack*, and evaluate it on microbenchmarks and on a new, synthetic private-ML workload we developed from the Alibaba ML cluster trace. We show that DPack: (1) often approaches the efficiency-optimal schedule, (2) consistently schedules more tasks compared to a state-of-the-art privacy scheduling algorithm that focused on fairness instead of efficiency (1.3–1.7× in Alibaba, 1.0–2.6× in microbenchmarks), but (3) sacrifices some level of fairness for efficiency. Using DPack, DP ML operators should be able to train more models on the same amount of user data while offering the same privacy guarantee to their users.

2.2 Introduction

Machine learning (ML) models are consuming an essential resource – *user privacy* – but they are typically not accounting for or bounding this consumption. A large company may train thousands of models over user data per week, continuously updating its models as it collects new data.

Some of the models may be released to mobile devices or distributed globally to speed up inference. Unfortunately, there is increasing evidence that ML models can reveal specific entries from their original training sets [1, 2, 3, 4, 5], both through parameters and predictions, thereby potentially leaking user data to adversaries. Intuitively, the more one learns from aggregate user data, the more one should expect to also learn (and hence leak) about individual users whose data is used. This intuition has been proven formally for simple statistics [6] and repeatedly demonstrated experimentally for ML models [2, 3, 5]. Therefore, user privacy can be viewed as a *resource* that is consumed by tasks in an ML workload, and whose consumption should be accounted for and bounded to limit data leakage risk.

Differential privacy (DP) [7] provides a rigorous way to define the privacy resource, and to account for it across multiple computations or tasks, be they ML model training tasks or statistic calculations. DP randomizes a computation over a dataset (e.g. training an ML model or computing a statistic) to bound the leakage of entries in the dataset through the output of the computation [8]. Each DP computation increases this bound on data leakage, consuming some of the data's *privacy budget*, a pre-set quantity that should never be exceeded to maintain the privacy guarantee. In workloads with a large number of tasks that continuously train models on a private corpus or stream, the data's privacy budget is a very scarce resource that must be efficiently allocated to enable the execution of as many tasks as possible.

In our prior work [8, 9], we began exploring how to expose data privacy as a new *computing resource* that is inherently being consumed by the tasks in an ML cluster and which must therefore be allocated and managed by the cluster's resource manager similarly to how other, more traditional computing resources – CPU, GPU, and RAM – are managed. Other researchers proposed Cohere [10] an alternative approach for treating privacy as a computing resource. A common conclusion of these prior works is that because the privacy resource behaves differently from traditional computing resources (e.g. it is finite), scheduling it requires new algorithms. To this end, we proposed DPF [8], the first scheduling algorithm for the privacy resource, which adapted the well-known dominant resource fairness (DRF) algorithm to the privacy resource. Our focus was on

fairness as the key objective for our algorithm design. DPF guarantees a form of max-min fairness for the privacy budget when multiple tasks compete for it.

Unfortunately, as is often the case in scheduling [11, 12, 13, 14, 15, 16], fairness can come at the cost of allocation *efficiency*, measured as the total number of tasks that are allocated over a unit of time. For privacy, we find that this inefficiency is especially evident in workloads that exhibit a high degree of *heterogeneity* either in the data segments they request (e.g., a workload containing tasks that run on data collected from different time ranges), or in the types of tasks they contain (e.g. a workload mixing different types of statistics and ML algorithms). In such cases, we show that a scheduler that optimizes for efficiency rather than fairness can schedule up to 2.6× more tasks than DPF for the same privacy budget.

In this paper, we explore the first practical *efficiency-oriented privacy schedulers*, which aim to maximize the number of scheduled tasks, or the total utility of scheduled tasks if tasks are assigned utility weights (§2.4). We first introduce a new formulation of the DP scheduling problem, which optimizes for efficiency, and show that it maps to the NP-hard multidimensional knapsack problem, requiring practical approximations to solve in practice. We demonstrate that (1) our prior DPF algorithm, which optimizes for fairness, can be seen as an inefficient heuristic to solve this problem, and that (2) a better heuristic for multidimensional knapsack yields more efficient DP scheduling. We then show that instantiating the privacy scheduling problem to Rényi DP (RDP) accounting, a state-of-the-art, efficient DP accounting mechanism, introduces a new dimension with unusual semantics to the scheduling problem. To support this new dimension, we define a new knapsack problem that we call the *privacy knapsack*, which we show is also NP-hard. Finally, we propose a new RDP-aware heuristic for the privacy knapsack, instantiate it into a new scheduling algorithm called *DPack*, provide a formal analysis of its approximation properties, and discuss when one should expect to see significant efficiency gains from it (§2.5).

We implement DPack in a Kubernetes-based orchestrator for data privacy [8] and an easily-configurable simulator (§3.6). Using both microbenchmarks and a new, synthetic, DP-ML workload we developed from the Alibaba's ML cluster trace [17], we compare DPack to DPF, the

optimal privacy knapsack solver, and first-come-first-serve (FCFS) (§5.6). DPack schedules significantly more tasks than DPF (1.3–1.7× in Alibaba and 1.0–2.6× in microbenchmarks), and closely tracks the optimal solution, at least up to a small number of blocks and tasks where it is feasible for us to obtain the optimal solution. DPack on Kubernetes can scale to thousands of tasks, and incurs a relatively modest scheduler runtime overhead. Still, by focusing on efficiency, DPack sacrifices some level of fairness compared to DPF: in the Alibaba workload, DPF is able to schedule 90% of tasks that request less or equal than their privacy budget "fair-share", while DPack schedules only 60% of such tasks. This is inevitable given the rather fundamental tradeoff between efficiency and fairness in scheduling. Our work thus fills in an important gap on algorithms that prioritize efficiency over fairness, as we believe will be desirable given the scarcity of this essential new resource in ML systems, user privacy.

This paper is organized as follows. §5.3 provides background on the threat model we are addressing, DP, and prior work on DP scheduling. Much of this section builds upon our prior papers in this space [8, 9], so there is considerable redundancy in the statements with those papers', which we include for the purposes of making this paper self-contained. §2.4 begins our main contributions in this work, consisting of the definitions and hardness properties of the efficiency-oriented DP scheduling problem, its adaptation for RDP, and the DPack algorithm we propose for both efficient and practical DP resource scheduling. §2.5 describes the applicability of our approach, highlighting cases when DPack is likely to give substantial efficiency benefit compared to DPF, as well as cases when it will not do so. §3.6 presents our implementation of DPack, while §5.6 provides our evaluation. Finally, §4.8 reviews related works and §4.9 concludes. We make our prototype and experimental code available at `https://github.com/columbia/dpack`.

2.3 Background

2.3.1 Threat Model

We adopt the same threat model as in our prior work [8]. We are concerned with the sensitive data exposure that may occur when pushing models trained over user data to untrusted locations,

such as end-user devices or inference servers all around the world. We operate under a centralized-DP model: a trusted curator collects and stores all user data and executes tasks, which consist of ML training procedures or pipelines that are explicitly programmed to satisfy a particular (ϵ, δ)-DP guarantee. We trust that the curator and the programmers of the tasks are not malicious and will not want to inspect, steal, or sniff the data. However, we do not trust the recipients of results released by the system, or the locations in which they are stored. Those results may be statistical aggregates, ML model predictions, or entire ML models. Accessing them may allow malicious activities that compromise sensitive personal information. We impose DP guarantees across all the processes that generate them. *Membership inference* attacks [18, 19, 20, 3] allow the adversary to infer whether an individual is in the data used to generate the output. *Data reconstruction attacks* [2, 6, 1] allow the adversary to infer sensitive attributes about individuals that exist in this data. We tackle both types of attack.

Our focus is not on single models or statistics, released once, but rather on *workloads of many models or statistics*, trained or updated periodically over windows of data from user streams. For example, a company may train an auto-complete model daily or weekly to incorporate new data from an email stream, distributing the updated models to mobile devices for fast prediction. Moreover, the company may use the same email stream to periodically train and disseminate multiple types of models, for example for recommendations, spam detection, and ad targeting. This creates ample opportunities for an adversary to collect models and perform privacy attacks to siphon personal data. To prevent such attacks, our goal is to *maintain a global (ϵ^G, δ^G)-DP guarantee over the entire workload consisting of many tasks*.

2.3.2 DP Background

We present background on DP theory that is necessary to understand our scheduling algorithm. DP addresses both membership inference and data reconstruction attacks [3, 1, 2, 21]. Intuitively, both attacks work by finding data points (which can range from individual events to entire users) that make the observed model more likely: if those points were in the training set, the likelihood

of the observed model increases. DP prevents these attacks by ensuring that no specific data point can drastically increase the likelihood of the model produced by the training procedure.

DP randomizes a computation over a dataset (such as the training of ML model) to bound a quantity called *privacy loss*, defined as some measure of the change in the distribution over the outputs of the randomized computation incurred when a single data point is added to or removed from the input dataset. Privacy loss is a formalization of what one might colloquially call "leakage" through a model. Satisfying DP means bounding privacy loss by some fixed, parameterized value, $\epsilon > 0$, which is called *privacy budget*. This bound is enforced through the virtue of the randomness (often called noise) added into the computation. There are multiple ways to define privacy loss, corresponding to various ways to define the distance between two output distributions. These different privacy loss definitions lead to different DP definitions, each with different interpretations, strengths and weaknesses. We review two DP definitions here.

Traditional differential privacy (ϵ-DP and (ϵ, δ)-DP). The original definition proposed by Dwork, et al. [7] defines privacy loss as follows. Given a randomized algorithm, $\mathcal{A} : \mathcal{D} \to \mathcal{Y}$, for any datasets $\mathcal{D}, \mathcal{D}'$ that differ in one entry (called *neighboring datasets*) and for any output $y \in \mathcal{Y}$:

$$\text{PrivacyLoss}(y, \mathcal{D}, \mathcal{D}') = \log\Big(\frac{P(\mathcal{A}(\mathcal{D}) = y)}{P(\mathcal{A}(\mathcal{D}') = y)}\Big). \tag{2.1}$$

The traditional, pure ϵ-DP definition requires an algorithm to satisfy $|\text{PrivacyLoss}(y, \mathcal{D}, \mathcal{D}')| \leq \epsilon$ for any y, $\mathcal{D}$, $\mathcal{D}'$ as above. A variation of this definition, popularly used in ML, is (ϵ, δ)-DP: for $\delta \in [0, 1)$, it requires an algorithm to satisfy $=P(\mathcal{A}(\mathcal{D}) \in \mathcal{S}) \leq \exp(\epsilon)P(\mathcal{A}(\mathcal{D}') \in \mathcal{S}) + \delta$ for all $\mathcal{S} \subseteq \text{Range}(\mathcal{A})$, for each neighboring $\mathcal{D}, \mathcal{D}'$.

These traditional DP definitions have the strength of being relatively interpretable: for a small value of ϵ (e.g., $\epsilon \leq 1$), ϵ-DP can be interpreted as a guarantee that an attacker who inspects the output of an ϵ-DP computation will not learn anything new with confidence about any entry in the training set that they would not otherwise learn if the entry were not in the training set [22]. Similarly, for small δ (e.g., $\delta < \frac{1}{n^2}$ for dataset size n), (ϵ, δ)-DP guarantee is roughly a high-probability ϵ-DP guarantee. The advantage of (ϵ, δ)-DP is support for a richer set of randomization mecha-

nisms, such as adding noise from a Gaussian distribution, which pure DP cannot, and which often provide better privacy-utility tradeoffs. That is why (ϵ, δ)-DP is the reference privacy definition for DP ML.

Rényi DP Accounting ((α, ϵ)-RDP). More recent DP definitions define privacy loss differently, usually sacrificing interpretability for tighter analysis of randomization mechanisms and how they compose with each other, yielding even better privacy-utility tradeoffs, especially in DP ML. A state-of-the-art definition is RDP [23], which has been adopted internally by most DP ML platforms [24, 25, 26]. Instead of defining the privacy loss based on probability ratios as traditional DP does, RDP defines it in terms of the Rényi divergence, a particular distance between the distributions over all possible outcomes for $\mathcal{A}(\mathcal{D})$ and $\mathcal{A}(\mathcal{D}')$. Rényi divergence has a parameter, $\alpha > 1$, called *order*:

$$\text{PrivacyLoss}_\alpha(\mathcal{D}, \mathcal{D}') = \frac{1}{\alpha - 1} \log \mathop{\mathbb{E}}_{y \sim \mathcal{A}(\mathcal{D})} \left(\frac{P(\mathcal{A}(\mathcal{D}) = y)}{P(\mathcal{A}(\mathcal{D}') = y)} \right)^\alpha.$$

As before, (α, ϵ)-RDP requires that $|\text{PrivacyLoss}_\alpha(\mathcal{D}, \mathcal{D}')| \leq \epsilon$ for any datasets $\mathcal{D}, \mathcal{D}'$ differing in one entry.

RDP is less interpretable than traditional DP due to the complexity of Rényi divergence. However, one can always translate from (ϵ, α)-RDP to (ϵ_{DP}, δ)-DP [23] for any appropriately ranged values of α, ϵ, and δ:

$$\epsilon_{DP} = \epsilon + \frac{\log(1/\delta)}{\alpha - 1}. \tag{2.2}$$

RDP's greatest advantage over traditional DP – and the reason for its recent adoption by most major DP ML platforms as well as for our special consideration of it in this paper – is its support for *both efficient and convenient composition*. All successful DP definitions are closed under composition; i.e., running multiple DP computations satisfies the DP definition, albeit with a worse ϵ parameter. However, whereas with traditional DP, composing m mechanisms degrades the global guarantee linearly with m, with RDP, the global guarantee degrades with $\sqrt{m}$ when applying composition followed by conversion to traditional DP through Eq. 2.2. RDP's tighter analysis can allow

composition of more DP computations with the same ϵ guarantees; the advantage is particularly significant with a large m.

Since popular DP ML algorithms, such as DP SGD, consist of tens of thousands iterations of the same rudimentary DP computation (computing one gradient step over a sample batch), they require the most efficient composition accounting method. This is why most DP ML platforms internally operate on RDP to compose across training steps and then translate the cumulative RDP guarantee into traditional DP (with Eq. 2.2) to provide an interpretable privacy semantic externally. Similarly, since our goal is to develop *efficient scheduling algorithms* – that pack as many DP ML tasks as possible onto a fixed privacy budget – it is incumbent on us to consider RDP accounting in our scheduling formulations.[1] We do so in a similar way: internally, some of the algorithms we propose use RDP accounting (albeit to compose across ML training tasks, *not* across gradient steps within a task) but externally we will always expose a traditional DP guarantee. As it turns out, operating on RDP internally creates interesting challenges for scheduling, about which we discuss in §2.4.2.

2.3.3 Privacy Scheduling Background

In a recent line of work [9, 8], we have argued for the global privacy budget to be managed as a new type of computing resource in workloads operating on user data: its use should be tracked and carefully allocated to competing tasks. We adopt the same focus on ML platforms for continuous training on user data streams, such as Tensorflow-Extended (TFX), and build on the same basic operational model [9] and key abstractions and algorithms [8] for monitoring and allocating privacy in DP versions of these platforms. The operational model is as follows. Similar to TFX, the user data stream is split into multiple non-overlapping *blocks* (called *spans* in TFX [28]), for example by time, with new blocks being added over time. Blocks can also correspond to partitions given by SQL `GROUP BY` statements over public keys, such as in Google's DP SQL system [29] or in the DP library used for the U.S. Census [30]. There are multiple tasks, dynamically arriving over

[1] We considered, and discarded, advanced composition for traditional DP, which is also efficient but involves complex arithmetic that is untenable to incorporate in a scheduler [27].

time, that request to compute (e.g., train ML models) on subsets of the blocks, such as the most recent N blocks. The company owning the data wants to enforce a global traditional DP guarantee, (ϵ^G, δ^G)-DP, that cannot be exceeded across these tasks. Each data block is associated with a global privacy budget (fixed a priori), which is consumed as DP tasks compute on that block until it is depleted.

In Luo et al. [8], we incorporated *privacy blocks*, i.e., data blocks with privacy budget, as a new compute resource into Kubernetes, to allocate privacy budget from these blocks to tasks that request them. The resulting system, which is a drop-in extension of Kubernetes, is called *PrivateKube*. To request privacy budget from a privacy block, a task i sets a demand vector (d_i) of length m, equal to the number of blocks in the system. The demand vector specifies the privacy budget that task i requests for each individual block in the system (with a zero demand for blocks that it is not requesting). If task i is allocated, then its demand vector is consumed from the blocks' privacy budgets. When a block's privacy budget reaches zero, no more tasks can be allocated for that block and the block is removed. This ensures that a block of user data will not be used to extract so much information that it risks leaking information about the users. In this sense, each privacy block is a *non-replenishable* or finite resource. It is therefore important to carefully allocate budget from privacy blocks across tasks, so as to pack as many tasks as possible onto the blocks available at any time. That's the goal of *efficiency-oriented privacy scheduling* and it is in contrast (and as we shall see, at odds) with fairness-oriented scheduling, which we previously explored in PrivateKube with an algorithm called *DPF* (Dominating Privacy-block Fairness). We defer a description of DPF and the tradeoffs between fairness and efficiency in privacy scheduling until after we have formulated the efficiency-oriented privacy scheduling problem in what follows.

2.4 Efficiency-Oriented Privacy Scheduling

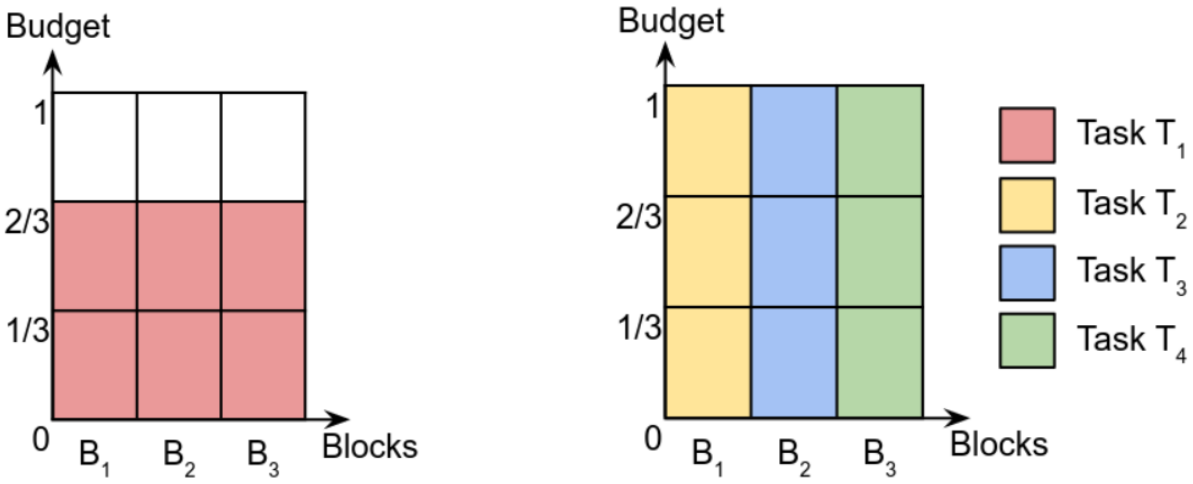
A key contribution of this work is the formalization of the efficiency-oriented DP scheduling problem. We first develop an *offline* version of this problem, in which the entire workload is assumed to be fixed and known a priori, and study efficient DP scheduling under traditional DP and

basic composition (§2.4.1). We show that offline DP scheduling maps to the NP-hard multidimensional knapsack problem, requiring practical approximations to solve in practice. Describing how our previous DPF algorithm works, we show that it can be seen as an inefficient heuristic for the efficiency-oriented scheduling problem, albeit one that has fairness guarantees. We then show that a better heuristic yields more efficient DP scheduling with multiple data blocks. In §2.4.2 we move onto a more complex RDP formulation of the efficiency-oriented allocation problem, but one that has the potential to boost efficiency significantly compared to traditional DP thanks to RDP's composition benefits. We prove the new RDP formulation as also NP-hard and develop a second, RDP-aware heuristic that leverages some unusual characteristics of this problem. In §2.4.3, we describe *DPack*, our proposed efficiency-oriented scheduling algorithm that incorporates both of our heuristics and in special settings can be shown to be a proper approximation of the efficiency-optimal solution to the RDP privacy knapsack problem. Finally, in §2.4.4 we adapt DPack to the online case.

2.4.1 Efficient Scheduling with Traditional DP

We define the *global efficiency* of a scheduling algorithm as either the number of scheduled tasks or, more generally, the sum of *weights* w_i of scheduled tasks, for cases when different tasks have different utilities (a.k.a. profits or weights) to the organization. When the goal is to optimize global efficiency, we can model privacy budget scheduling in a multi-block system such as TFX as a multidimensional knapsack problem. First, recall that traditional DP composes, in its simplest form, using an additive arithmetic: the composition of two (ϵ_1, δ_1)-DP and (ϵ_2, δ_2)-DP tasks is $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$-DP. In this paper we assume δ is extremely small (as it should always be, since it is a failure probability of the pure DP guarantee), hence we ignore the additive effects on the δ parameters and instead focus on the additive effects of the ϵ parameters, which are typically many orders of magnitude larger than the δ parameters.

Knapsack problem formulation. Consider a fixed number of n tasks $(t_1, \ldots, t_n)$ that need to be scheduled over m blocks, each with c_j remaining capacity. Each task has a demand vector d_{ij},



(a) Inefficient allocation with DPF

(b) Efficient allocation

Fig. 2.1: **Example of allocations with basic DP accounting.** Task T_1 requests privacy budget from 3 blocks, B_1, B_2, B_3. Tasks T_2, T_3, T_4 request slightly more privacy budget, but each one from one distinct block: B_1, B_2, B_3, respectively. In (a), DPF sorts these tasks based on their dominant shares: T_1 first (because its dominant share is lower, even though it demands budget from all the blocks), then T_2, T_3, T_4 in arbitrary order. After T_1 is scheduled, there is no more budget for other tasks. Meanwhile, in (b) an efficient scheduler can allocate 3 tasks.

which represents the ϵ demand by task i for block j, and a weight w_i if it is successfully scheduled (when w_i is equal across all tasks, the problem is to maximize the number of scheduled tasks). We can formulate this problem as the standard multidimensional knapsack problem [31], where x_i are binary variables:

$$\max_{x_i \in \{0,1\}} \sum_{i=1}^{n} w_i x_i \text{ subject to } \forall j \in [m] : \sum_{i=1}^{n} d_{ij} x_i \leq c_j. \tag{2.3}$$

W.l.o.g., we assume there is not enough budget to schedule all tasks: $\forall j \in [m] : \sum_{i=1}^{n} d_{ij} > c_j$. Otherwise, the knapsack problem is trivial to solve. If some blocks have enough budget but not others, we can set the blocks with enough budget aside, solve the problem only on the blocks with contention, and incorporate the remaining blocks at the end.

The need for heuristics. The multidimensional knapsack problem is known to be NP-hard [31], so DP scheduling cannot be solved exactly, even in the offline case. There exist some general-purpose polynomial approximations for this problem, but they are exponential in the approximation parameter and become prohibitive for large numbers of dimensions (for us, many blocks). In §2.7.2, we show that the Gurobi [32] solver quickly becomes intractable with just 7 blocks!

A standard approach to practically solve knapsack problems is to develop specialized approximations for a specific domain of the problem, typically using a *greedy algorithm* that sorts tasks according to a *task efficiency metric* (denoted e_i), and then allocates tasks in order, starting from the highest-efficiency tasks, until the algorithm cannot pack any new tasks [31]. In such algorithms, the main challenge is coming up with good task efficiency metrics that leverage domain characteristics to meaningfully approximate the optimal solution while remaining practical in terms of runtime.

Inefficiencies under DPF, seen as a scheduling heuristic. Turns out we can model DPF – our previous, fairness-oriented algorithm and still the state-of-the-art privacy scheduling algorithm – as a *greedy heuristic for privacy knapsack*. DPF schedules tasks with the smallest dominant share ($\max_j \frac{d_{ij}}{c_j}$) first. Folding in task weights, this becomes equivalent to a greedy algorithm with an efficiency metric defined as: $e_i := \frac{w_i}{\max_j \frac{d_{ij}}{c_j}}$. Unfortunately, given this efficiency metric, DPF can stray arbitrarily far from the optimal even in simple cases. The reason lies in the maxima over j, which is crucial to ensure the fair distribution of DP budget, but causes DPF to ignore multidimensionality in data blocks. Fig. 2.1 gives an example using traditional DP and a workload of 4 tasks. DPF sorts tasks by dominant share and schedules only one task. Meanwhile, a better efficiency metric would consider the "area" of a task's demand, thereby sorting tasks T_2, T_3 and T_4 before T_1, resulting in 3 tasks getting scheduled. Thus, DPF, despite its compelling weighted fairness guarantees, is merely a greedy heuristic when it comes to optimizing for efficiency; it is not even a proper approximation of the efficiency-optimal allocation, as it can stray arbitrarily far from it.

Area-based metric for efficient scheduling over blocks. We take inspiration from single-dimensional knapsacks, in which the efficiency e_i of task i is usually defined as the task's weight-to-demand ratio: $e_i := w_i/d_i$. A natural extension to multiple blocks uses a known heuristic for multidimensional knapsacks [33] to capture the entire demand of a task:

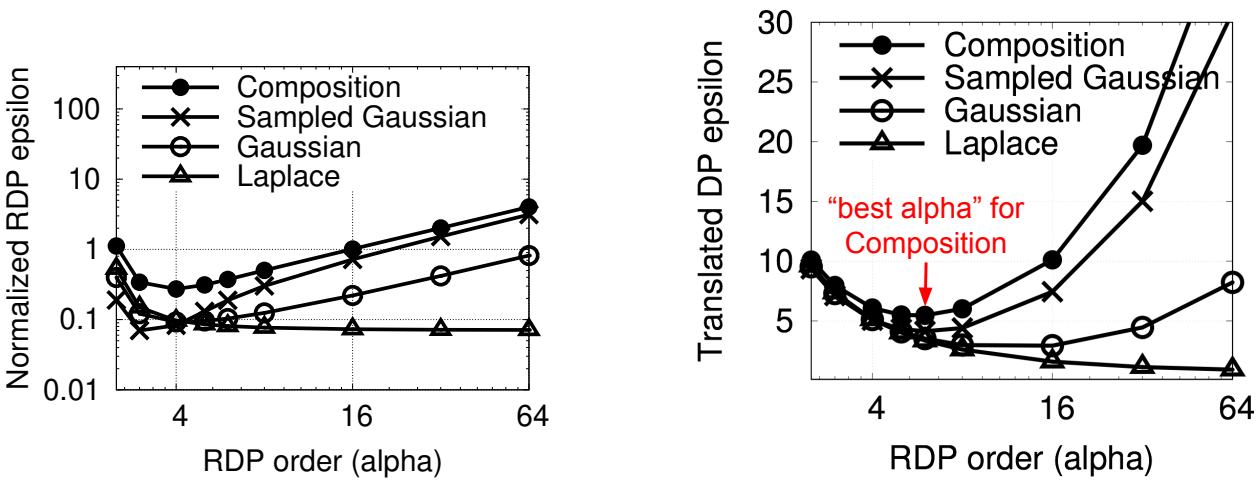
$$e_i := \frac{w_i}{\sum_j \frac{d_{ij}}{c_j}}, \tag{2.4}$$

where $\frac{d_{ij}}{c_j}$ is task i's DP budget demand for block j, normalized by the remaining capacity of block j. This normalization is important to express the scarcity of a demanded resource. Unlike the DPF fair scheduling metric, Eq. 2.4 considers the entire "area" of a task's demand to compute its efficiency, addressing the inefficiency from Fig. 2.1. A task requesting a large budget across blocks is not scheduled even if its demand on any block (dominant share) is small. As we shall see in experimental evaluation, this heuristic leads to more efficient scheduling than under DPF under traditional DP.

2.4.2 Efficient Scheduling Under RDP Accounting

The above heuristic is satisfactory for traditional DP accounting, but practitioners and state-of-the-art ML algorithms use the much more efficient RDP accounting. With RDP, multiple bounds on the privacy loss can be computed, for various RDP orders α (Eq. 2.3.2). This yields an *RDP order curve* $\epsilon(\alpha)$ for that computation. For instance, adding noise from a Gaussian with standard deviation σ into a computation results in $\epsilon(\alpha) = \frac{\alpha}{2\sigma^2}$. Other mechanisms, such as subsampled Gaussian (used in DP-SGD) or Laplace (used in simple statistics), induce other RDP curves. These curves are highly non-linear and their shapes differ among each other. This makes it difficult to know analytically what the privacy loss function will look like when composing multiple computations with heterogeneous RDP curves. For this reason, typically the RDP ϵ bound is computed on a few discrete α values ($\{1.5, 1.75, 2, 2.5, 3, 4, 5, 6, 8, 16, 32, 64\}$ [23]), on which the composition is performed. Importantly, composition of ϵ parameters at each α value is still additive, a key element of RDP's practicality.

Fig. 2.2a shows RDP curves for three example computations, each using a popular DP mechanism: the Gaussian would be used for a multidimensional statistic (a histogram); the subsampled Gaussian would be used in DP-SGD training; and Laplace would be used for a simple statistic (an average). All these are plausible to co-exist as tasks in an ML/data analytics cluster. These different computations exhibit different RDP curves, with different orderings of the Rényi divergence bound at different α's. The subsampled Gaussian is tighter at lower α values; the Laplace is tighter



(a) RDP curves

(b) DP translation

Fig. 2.2: **Example RDP curves and DP translation.** (a) RDP curves for Gaussian, subsampled Gaussian, and Laplace mechanisms, each with std-dev $\sigma = 2$, plus their composition. (b) Translation to $(\epsilon_{DP}, 10^{-6})$-DP. The "best" (i.e., tightest) alpha differs among mechanisms. For composition, best is $\alpha = 6$, giving $\epsilon_{DP} = 5.5$.

for large α's. The figure also shows the RDP curve for the composition of the three computations.

Fig. 2.2b shows the translation of these four curves into traditional DP (using Eq. 2.2). For each computation, any value of $\alpha > 1$ will translate into a different traditional ϵ. Some traditional ϵ translations are very loose, others are tighter, but they are all valid simultaneously. Because of this, we can pick the α that gives us the best traditional ϵ guarantee and disregard the rest as loose bounds. This *best alpha* differs from computation to computation: in our example, for the Gaussian it is $\alpha \approx 16$; for the subsampled Gaussian $\alpha \approx 6$; and for the Laplace $\alpha \geq 64$. The best alpha for the composition of all three computations is $\alpha \approx 6$, yielding $(\epsilon = 5.5, \delta = 10^{-6})$-DP. If we were to analyze and compose the three computations directly in traditional DP, we would obtain a looser global guarantee of $(\epsilon = 7.8, \delta = 10^{-6})$-DP. This gap grows fast with the number of computations. Herein lies RDP's power, but also a significant challenge when trying to allocate its privacy budget across competing computations.

Notice that when translating from RDP to traditional DP with Eq. 2.2, one chooses the most advantageous α for the final traditional DP guarantee, ignoring all other RDP orders. This new α dimension therefore has a different semantic than the traditional multidimensional knapsack one.

Indeed, the traditional knapsack dimension semantic is that an allocation has to be within budget *along all dimensions*. This is a good fit for our block dimension, as we saw in §2.4.1. Instead, an allocation is valid along the α dimension as long as the allocation is within budget for *at least one dimension*. This creates opportunities for efficient scheduling, as the allocator can go over-budget for all but one α order. It also creates a new challenge, as the α order that will yield the most efficient allocation is unknown a priori and depends on the chosen combination of tasks. Since the traditional multidimensional knapsack does not encode this new semantic, we define a new multidimensional knapsack problem for efficient RDP scheduling.

The RDP privacy knapsack problem. To accommodate RDP, we need to modify the standard multidimensional knapsack problem to support alpha orders for each block and task demand. We express the capacity as $c_{j\alpha}$ (the available capacity of block j on order α), each demand vector as $d_{ij\alpha}$ (the demand of task i on block j's order α), and require that the sum of the demands will be smaller or equal to the capacity for *at least one of the alpha orders*. We thus formulate the privacy knapsack as follows:

$$\max_{x_i \in \{0,1\}} \sum_{i=1}^{n} w_i x_i \text{ subject to } \forall j \in [m], \exists \alpha \in A : \sum_{i=1}^{n} d_{ij\alpha} x_i \leq c_{j\alpha}. \tag{2.5}$$

Property 1. *The decision problem for the privacy knapsack problem is NP-hard.*

Property 2. *In the single-block case, there is a fully polynomial time approximation scheme (FPTAS) for privacy knapsack, i.e., with $w^{\max}$ the highest possible global efficiency, for any $\eta > 0$ we can find an allocation with global efficiency $\hat{w}$ such that $w^{\max} \leq (1+\eta)\hat{w}$, with a running time polynomial in n and $1/\eta$.*

Property 3. *For $m \geq 2$ blocks, there is no FPTAS for the privacy knapsack problem unless P=NP.*

While Prop. 1 and 3 are disheartening (though perhaps unsurprising), Prop. 2 gives a glimmer of hope that at least for single-block instances, we can solve the problem tractably. Indeed, as we shall see, this property is crucial for our solution.

DPF with multiple RDP alpha orders. Fair scheduling with DPF for RDP can once again be

(a) Inefficient allocation with DPF

(b) Efficient allocation

Fig. 2.3: **Example of allocations with RDP accounting.** In (a), DPF treats RDP orders like a regular resource and orders tasks by dominant share, allocating only 2 tasks in this example. Meanwhile, (b) leverages the fact that only one order per block has to be below the capacity (here, α_1 for block B_1 and α_2 for block B_2). Tasks T_3 and T_5 have a large dominant share of 1.5 but are efficient because they request only 0.5 for B_1's best alpha, α_1.

expressed as an ordering heuristic for the privacy knapsack, in which efficiency is defined as $e_i := \frac{w_i}{\max_{j\alpha} \frac{d_{ij\alpha}}{c_{j\alpha}}}$. However, this approach is even more inefficient than under traditional DP.

In addition to the previous multi-block inefficiency (§2.4.1), this fair scheduling approach exhibits a new inefficiency under RDP, regardless of the number of blocks it is invoked on (e.g., even if applied to non-block-based DP systems, such as DP SQL databases). Fig. 2.3 gives an example using two blocks and a workload of 6 tasks, each requesting only 1 block. In Fig. 2.3a, DPF sorts tasks by the highest demands across all α's and allocates only 2 tasks. A better efficiency metric would sort tasks by demands at the α value that can pack the most tasks (a.k.a., best alpha for composition), ultimately scheduling 4 tasks in Fig. 2.3b. Note that the best alpha is not necessarily the same for each block.

We conclude that an efficiency metric that simply takes the maximum of the dominant shares is neither efficient for scheduling multiple privacy blocks, nor for scheduling privacy budget in systems that use RDP accounting. However, a direct extension of our "area based" efficiency metric in Eq. 2.4 does not appropriately handle RDP alpha orders either, as it does not account for

the specific semantic of the α order. We next describe our new efficiency metric, that is optimized for efficiently scheduling tasks across multiple blocks and supports RDP.

2.4.3 DPack Algorithm

Intuitively, to support the "at least one" semantic of the α order from RDP, we need an efficiency metric that makes it less attractive to pack a task that consumes a lot of budget at what will ultimately be the *best alpha*, defined as the RDP order that packs the most tasks (or the most weight) while remaining under budget. That best alpha is ultimately the only one for which the demands of tasks matter and hence should be the one used for computing an efficiency metric. The challenge is that for workloads consisting of tasks with heterogeneous RDP curves, the best alpha is not known a priori. Our idea is to approximate it on a smaller set of RDP curves, and to focus a task's efficiency metric on that best alpha as the only relevant dimension. Recall from Prop. 2 that in the single-block case, we can solve privacy knapsack with polynomial-time η-approximation for arbitrarily small $\eta > 0$. This means we can solve a single-block knapsack problem *separately for each block j* that determines the best alpha that will pack the most tasks (or maximal weight) among tasks requesting block j, taking only their request for that block into account. We define the maximum utility for block j and order α as $w_{j\alpha}^{\max} := \max_{x_i} \sum_{i:d_{ij\alpha}>0}^{n} x_i w_i$ subject to $\sum_i x_i d_{ij\alpha} \leq c_{j\alpha}$. We take $\hat{w}_{j\alpha}^{\max}$ a $\frac{2}{3}\eta$-approximation of $w_{j\alpha}^{\max}$ ($\frac{2}{3}$ is justified by proof below).

Based on this, we define the efficiency of task i as:

$$e_i := \frac{w_i}{\sum_{j\alpha}(\frac{d_{ij\alpha}}{c_{j\alpha}} \text{ if } (\alpha == \arg\max_{\alpha'} \hat{w}_{j\alpha'}^{\max}) \text{ else } 0)} \tag{2.6}$$

Alg. 1 shows *DPack*, our greedy approximation with the efficiency metric in Eq. 2.6. This algorithm addresses both of the problems we identified with DPF. Moreover, we show that the manner in which DPack handles RDP is not just better than DPF in particular, but rather has two important generally desirable properties. First, DPack reduces to the traditional multidimensional

Algorithm 1 DPack Offline Algorithm

Input: tasks i, blocks j, RDP orders α capacities $c_{j\alpha}$
Input: approximation factor η, demands $d_{ij\alpha}$, weights w_i
function COMPUTEBESTALPHA(block j)
  for $\forall \alpha$ **do**
    $\hat{w^{\max}}_{j\alpha} \leftarrow$ SINGLEBLOCKKNAPSACK($c_\alpha, d_{ij\alpha}, w_i, \frac{2}{3}\eta$)
  return $\arg\max_\alpha \hat{w^{\max}}_{j\alpha}$
function COMPUTEEFFICIENCY(task i, best alphas $\hat{\alpha}_j^{\max}$)
  return $w_i / \sum_j (d_{ij\hat{\alpha}_j^{\max}} / c_{j\hat{\alpha}_j^{\max}})$
function CANRUN(task i)
  return $\forall j, \exists \alpha : \sum_{i'=1}^{i} d_{i'j\alpha} \leq c_{j\alpha}$
function SCHEDULE(tasks i)
  for $\forall j$ **do**
    $\hat{\alpha}_j^{\max} \leftarrow$ COMPUTEBESTALPHA($c_{j\alpha}, d_{ij\alpha}, w_i$)
  sorted_tasks $\leftarrow$ tasks.sortBy(COMPUTEEFFICIENCY($\hat{\alpha}_j^{\max}$))
  for i in sorted_tasks **do**
    if CANRUN($d_{ij\alpha}$) **then**
      Run task i, consuming the demanded budget

knapsack efficiency metric of Eq. 2.4 when only one α exists, e.g. for traditional DP:

Property 4. *If the dimension of α values is one (e.g., with traditional DP), DPack reduces to the traditional multidimensional knapsack heuristic from Eq. 2.4.*

Proof. With one dimension, $\alpha = \arg\max_{\alpha'} \hat{w}^{\max}_{j\alpha'}$. □

Second, DPack is a *guaranteed approximation* of the optimal in the specific cases when such an approximation is possible, the single-block case:

Property 5. *In the single-block case, DPack is a ($\frac{1}{2} + \eta$)-approximation algorithm for privacy knapsack.*

Proof. Call $\hat{\alpha} \triangleq \arg\max_{\alpha'} \hat{w}^{\max}_{j\alpha'}$. By construction we have $w^{\max}_{j\hat{\alpha}} \leq (1 + \frac{2}{3}\eta)\hat{w}^{\max}_{j\hat{\alpha}}$. In the single-block (index j) case, Eq. 2.6 means that tasks are greedily allocated by decreasing $\frac{w_i}{d_{ij\hat{\alpha}}}$, a well known 1/2-approximation to the one dimensional knapsack problem [31]. Hence, $w^{\max}_{j\hat{\alpha}} \leq (1 + \frac{2}{3}\eta)\hat{w}^{\max}_{j\hat{\alpha}} \leq (1 + \frac{2}{3}\eta)(1 + \frac{1}{2}) \sum_{i=1}^{n} x_i w_i = (1 + \frac{1}{2} + \eta) \sum_{i=1}^{n} x_i w_i$. □

Because of Prop. 3, a similar multi-block efficiency guarantee cannot be formulated (for DPack as well as any other poly-time algorithm). However, §5.6 shows that in practice, DPack performs close to the optimal solution of privacy knapsack in terms of global efficiency, yet it is a computationally cheap alternative to that intractable optimal solution.

2.4.4 Adapting to the Online Case

In practice, new tasks and blocks arrive dynamically in a system such as TFX, motivating the need for an online scheduling algorithm. We adapt our offline algorithm to the online case by scheduling a batch of tasks on the set of available blocks every T units of time. To prevent expensive tasks from consuming all the budget prematurely, similar to DPF, we schedule each batch on a fraction of the total budget capacity: at each scheduling step we unlock an additional $1/N$ fraction of the block capacity. More precisely, at each scheduling time $t = kT$, we execute Alg. 1 on the tasks and blocks currently in the system, but we replace block j's capacity by:

$$c^t_{j\alpha} = \frac{\min(\lceil (t - t_j)/T \rceil, N)}{N} \epsilon_{j\alpha} - \sum_{i' \in A_t} d_{i'j\alpha},$$

where $\epsilon_{j\alpha}$ is the total capacity of block j (computed from Prop. 2.2), t_j is the arrival time of block j, $\lceil (t - t_j)/T \rceil$ is the number of scheduling steps the block has witnessed so far (including the current step), and A_t is the set of tasks previously allocated.

As with the offline algorithm, at the time of scheduling all the tasks are sorted by the scheduling algorithm. The scheduler tries to schedule tasks one-by-one in order. Any tasks that did not get scheduled remain in the system until the next scheduling time, and any unused unlocked budget remains available for future tasks. Users also specify a per-task timeout after which the task is evicted. T is a parameter of the system that controls how many tasks get batched (and delayed) before getting scheduled. We evaluate its effect empirically in Fig. **??**, and show that beyond a reasonable batch size all algorithms we study are relatively insensitive to T.

Finally, to support a global (ϵ, δ)-DP guarantee for online tasks over continuous data streams,

we use the data block composition introduced by Sage [9, 34]: each data block is associated with a privacy filter, a DP accounting mechanism enabling adaptive composition under a preset upper-bound on the privacy loss [35, 36, 37]. Each filter is initiated with ϵ, δ for traditional DP, or $\epsilon(\alpha) = \epsilon - \frac{\log(1/\delta)}{\alpha-1}$ for RDP. The RDP initial value ensures that translating back to traditional DP with Eq. 2.2 guarantees (ϵ, δ)-DP. A task is granted if, and only if, all filters grant the request (all blocks have enough budget left). This ensures the following property:

Property 6. *DPack enforces (ϵ, δ)-DP over adaptively chosen computations and privacy demands $\epsilon_i(\alpha)$.*

Proof. We provide a proof sketch following the structure used in [9, Theorem 4.2] for basic composition. Each task has an (adaptive) RDP requirement for all blocks, with $\epsilon(\alpha) = 0$ for non-requested blocks. Each data block is associated with a privacy filter [37, Algorithm 1]. A task runs if and only if all filters accept the task: applying [37, Theorem 1] ensures $\epsilon(\alpha) = \epsilon - \frac{\log(1/\delta)}{\alpha-1}$-RDP holds for each block. Applying Eq. 2.2 concludes the proof. □

2.5 Applicability

It is worth reflecting on the characteristics of workloads under which DPack provides the most benefit compared to alternatives such as DPF. §2.4.1 gives examples of inefficient DPF operation with multiple blocks and alpha orders. However, DPF does not *always* behave inefficiently when invoked on multiple blocks or with multiple alpha orders. For example, if all the tasks in Fig. 2.1 uniformly demanded three blocks, then DPF would make the optimal choice. The same would happen if all the tasks in Fig. 2.3 had RDP curves that were all ordered in the same way across alphas, so that the ordering of highest demands is the same as the ordering of demands at the best alpha order. In such cases, DPack's "intelligence" – its appropriate treatment of the multiple blocks and focus on the best alpha – would not provide any benefit over DPF.

Instead, DPack should be expected to improve on DPF when the workload exhibits *heterogeneity* in one or both of the following two dimensions: (1) number of demanded blocks and (2) best alphas. (1) The example in Fig. 2.1 exhibits high heterogeneity in demanded blocks, with Task 1

demanding three blocks while all the others demanding just one block. (2) The example in Fig. 2.3 exhibits heterogeneity in the best alpha for the different curves. In evaluation (§2.7.2), we demonstrate this effects using a microbenchmark that is able to explore a wide range of more or less heterogeneous workloads, showing that indeed, in workloads with more heterogeneity DPack significantly outperforms DPF while in cases of homogeneity among all dimensions, DPack performs similarly to DPF.

For real-world DP ML workloads, we believe it is likely that heterogeneity of demands in both dimensions – number of blocks and best alphas – would be realistic. For example, a pipeline that computes some summary statistics over a dataset might run daily on just the latest block, while a large neural network may need to retrain on data from the past several blocks. This would result in heterogeneity in number of demanded blocks. Similarly, pipelines that compute simple statistics would likely employ a Laplace mechanism, while a neural network training would employ subsampled Gaussian. This would inevitably result in heterogeneity in best alphas, because, as shown in Fig. 2.2, different mechanisms exhibit very different RDP curves.

Thus, DPack is broadly applicable to: (1) systems that exhibit both of these dimensions of heterogeneity (as would DP ML workloads in TFX-like systems, or static SQL databases with multiple partitions); (2) systems that operate on a single block (such as non-partitioned SQL databases) but perform RDP accounting; (3) systems that operate on multiple blocks but perform other types of DP accounting, including traditional DP. For all these settings, DPack would provide a benefit when the workload exhibits heterogeneity.

2.6 Implementation

We implement DPack in two artifacts that we open-source at `https://github.com/columbia/dpack`. The first is a **Kubernetes-based implementation** of DPack. We extend PrivateKube's extension to Kubernetes in multiple ways. We add support for batched scheduling (i.e. schedule tasks every T time units) and task weights. We implement DPack, and add support for solving the single block knapsack using Gurobi with the Go goop interface [38]. The

	Sec.	Workload	Setting	Prototype	Results
Q1	§2.7.2	microbenchmark	offline	simulator	Fig. 2.4
Q2	§2.7.2	microbenchmark	offline	simulator	Fig. 2.5
Q3	§2.7.3	Alibaba, Amazon	online	simulator	Fig. 2.6-2.7
Q4	§2.7.4	Alibaba	online	Kubernetes	Fig. 2.8

Tab. 2.1: **Workload and methodology of each evaluation question.**

Kubernetes-based implementation has 924 lines of Go. The second artifact is a **simulator** that lets users easily specify and evaluate scheduling algorithms for the offline and online settings under different workloads. We use a discrete event simulator [39] to efficiently support arbitrarily fine time resolutions. Users use configuration files to define the workload and resource characteristics to parameterize scheduling for both online and offline cases. For example, they can define block and task arrival frequencies, the scheduling period and the block unlocking rate. The simulator also supports plugging different definitions of efficiency, and different block selection patterns for tasks (policies). Currently, the simulator supports two patterns: a random selection of blocks without replacement, and a selection of most recent blocks. The simulator has 6,718 lines of Python.

2.7 Evaluation

We seek to answer four evaluation questions:

Q1: On what types of workloads does DPack improve over DPF, and how close is DPack to Optimal?

Q2: How do the algorithms scale with increasing load?

Q3: Does DPack present an efficiency improvement for plausible workloads? How much does it trade fairness?

Q4: How does our implementation perform in a realistic setting?

These questions are best answered with distinct workloads and settings, summarized in Tab. 2.1. First, Q1 and Q2 are best addressed in an offline setting with a simple, tunable workload. To this end, we develop a microbenchmark consisting of multiple synthetic tasks with distinct RDP curves and a knob that controls the heterogeneity in demanded blocks and RDP curves (§2.7.2). Second,

Q3 and Q4 require a more realistic, online setting and realistic workloads. In absence of a production trace of DP ML tasks, we develop a workload generator, called *Alibaba-DP*, based on Alibaba's 2022 ML cluster trace [17]. We map the Alibaba trace to a DP ML workload by mapping system metrics to privacy parameters (§2.7.3). While we cannot claim Alibaba-DP is realistic, it is the first *objectively-derived* DP task workload generator, and we believe it is a more plausible workload than those previously used in related works. We plan to release it publicly. Third, Q1-Q3 are algorithmic-level questions independent of implementation and hence we evaluate them in the simulator. However, Q4 requires an actual deployment on Kubernetes, so we dedicate the last part of this section to an evaluation on Kubernetes with the Alibaba-DP workload (§2.7.4).

2.7.1 Methodology

Baselines. The main baseline, common across all experiments, is *DPF*. We consider two other baselines: *Optimal*, which is the exact Gurobi-derived privacy knapsack solution for the offline setting, and *FCFS* (first-come-first-serve), which schedules tasks in an online setting based on their order of arrival. The former is relevant for offline experiments of small scale (few tasks/blocks), since it is not tractable for larger ones. The latter is relevant for online experiments only.

Metrics. *Global efficiency:* defined as either the number of allocated tasks or the sum of weighted allocated tasks. *Scheduler runtime:* measures how fast (in seconds), computationally, a scheduling algorithm is. *Scheduling delay:* measures how long tasks are blocked in the waiting queue, for example because of insufficient unlocked budget or because of the batching period T; it is measured in block inter-arrival periods (e.g., if blocks arrive daily, the unit is days). In real life, the total waiting time for a task will be the scheduling delay plus scheduler runtime; for our experiments, since the two are in different units, we never combine them. We expect in reality scheduler runtimes to be small compared to scheduling delays, for all the evaluated algorithms except for Optimal.

Machine. We use a server with 2 Intel Xeon CPUs E5-2640 v3 @ 2.60GHz (16 cores) and 110GiB RAM.

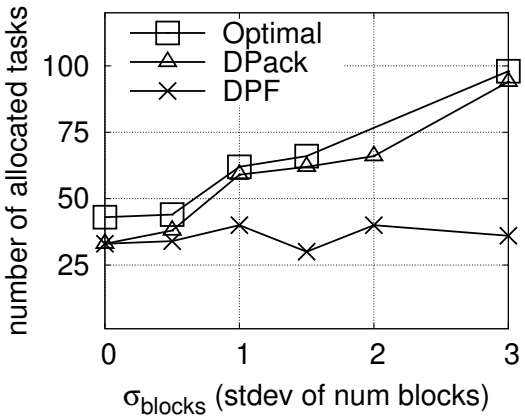
2.7.2 Offline Microbenchmark (Q1, Q2)

Microbenchmark. We design the microbenchmark to expose knobs that let us systematically explore a spectrum of workloads ranging from less to more heterogeneous in demanded blocks and RDP curve characteristics. The microbenchmark consists of 620 RDP curves corresponding to five realistic DP mechanisms often incorporated in DP ML workloads: {*Laplace, Subsampled Laplace, Gaussian, Subsampled Gaussian, composition of Laplace and Gaussian*}. We sample and parameterize these curves with the following methodology meant to expose two heterogeneity knobs:

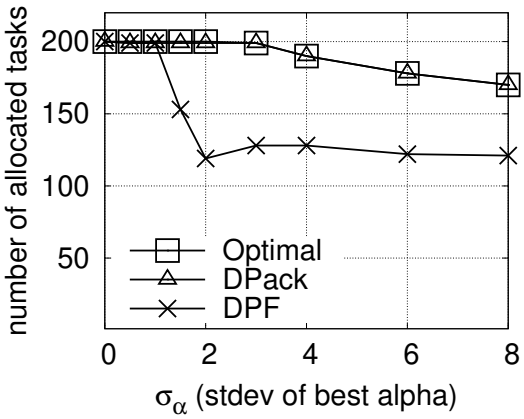
Knob σ_{blocks}: To exercise heterogeneity in requested blocks, we sample the number of requested blocks from a discrete Gaussian with mean μ_{blocks} and standard deviation σ_{blocks}. The requested blocks are then chosen randomly from the available blocks. Increasing σ_{blocks} increases heterogeneity in demanded blocks.

Knob σ_α: To exercise heterogeneity in best alphas, we first normalize the demands (for a block with initial budget $(\epsilon, \delta) = (10, 10^{-7})$) and enforce that there is at least one curve with best alpha α for each $\alpha \in \{3, 4, 5, 6, 8, 16, 32, 64\}$. Second, we group tasks with identical best alphas to form "buckets". For each new task, we pick a best alpha following a truncated discrete Gaussian over the bucket's indexes, centered in the bucket corresponding to $\alpha = 5$ with standard deviation σ_α. Third, we sample one task uniformly at random from that bucket. After dropping some outliers (e.g. curves with $\epsilon_{\text{min}} < 0.05$), we rescale the curves to fit any desired value of the average and the standard deviation of ϵ_{min} for each best alpha, by shifting the curves up or down. This scaling lets us change the distribution in best alphas while controlling for the average size of the workload (in a real workload, the value of ϵ_{min} might be correlated with best alpha and other parameters). Increasing σ_α increases workload heterogeneity in best alphas.

We explore each heterogeneity knob separately. First, we vary σ_{blocks} while keeping $\sigma_\alpha = 0$ (i.e. all the tasks have best alpha equal to 5) and $\mu_{blocks} = 10$. Second, we vary σ_α while keeping $\sigma_{blocks} = 0$, $\mu_{blocks} = 1$ (i.e. all the tasks request the same single block). In both cases, we keep ϵ_{min} constant for all tasks. We set $\epsilon_{\text{min}} = 0.1$ for the σ_{blocks} experiment (to keep the number of



(a) Block heterogeneity



(b) Best alpha heterogeneity

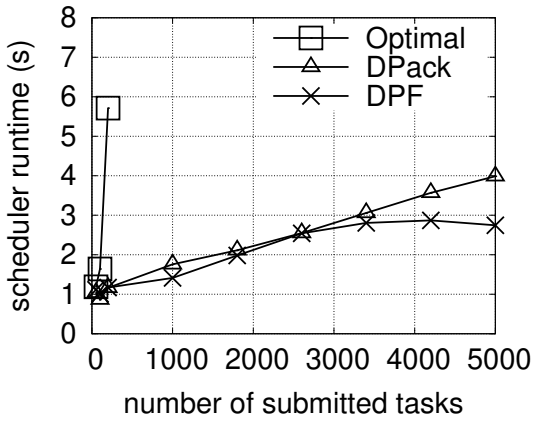
Fig. 2.4: **(Q1) DPack under workloads with variable heterogeneity using our microbenchmark.** Global efficiency of the algorithms (y axes) in the offline setting, as heterogeneity increases on the x axes: (a) variation in number of blocks requested, (b) variation in best alphas for the tasks' RDP curves. *Q1 Answer: DPack tracks Optimal closely and significantly outperforms DPF on workloads with high heterogeneity: 0–161% improvement for Fig. 2.4a and 0–67% for Fig. 2.4b.*

tasks small enough to be tractable for Optimal) and $\epsilon_{\min} = 0.005$ for the σ_α experiment (to have a large number of tasks with high diversity in $\epsilon(\alpha)$).

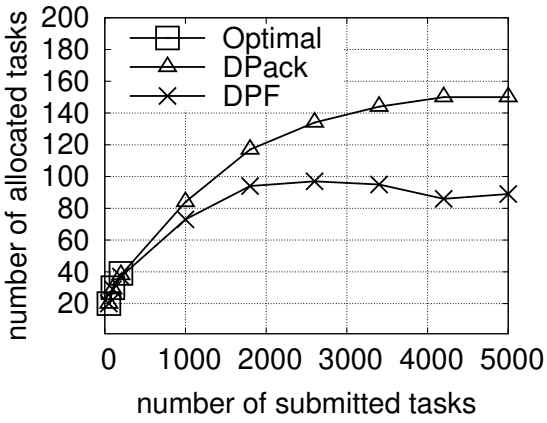
Q1: On what types of workloads does DPack improve over DPF, and how close is DPack to Optimal? Fig. 2.4 compares the schedulers' global efficiency in the offline setting, as the heterogeneity of the workload increases in the two preceding dimensions: the number of requested blocks (Fig. 2.4a) and the best alphas of the tasks' RDP curves (Fig. 2.4b). Across the entire spectrum of heterogeneity, DPack closely tracks the optimal solution, staying within 23% of it. For workloads with low heterogeneity (up to 0.5 stdev in blocks and 1 stdev in best alphas), there is not much to optimize. DPF itself therefore performs close to Optimal and hence DPack does not provide significant improvement. As heterogeneity in either dimension increases, DPack starts to outperform DPF, presenting significant improvement in the number of allocated tasks for over 3 stdev in blocks and 2 stdev in best alphas: 161% and 67% improvement, respectively.

As all three schedulers try to schedule as many tasks as they can with a finite privacy budget, these 1.0–2.6× additional tasks that DPack is able to schedule are tasks that *DPF would never be able to schedule*, because the requested blocks' budget has been depleted for posterity.

Q2: How do the algorithms scale with increasing load? Fig. 2.5a shows the runtime of our



(a) Scheduler runtime as a function of offered load.



(b) Number of allocated tasks as a function of offered load.

Fig. 2.5: **(Q2) Scalability under increasing load from the microbenchmark.** (a) Scheduler runtime and (b) number of allocated tasks, as a function of offered load (x axes). *Q2 Answer: Optimal becomes intractable quickly while DPack and DPF remain practical even at high load.*

simulator on a single thread. We use a single thread for a fair comparison, but some schedulers can be parallelized (our Kubernetes implementation is indeed parallelized). We use the microbenchmark with heterogeneity knobs $\sigma_\alpha = 4, \sigma_{blocks} = 10, \mu_{blocks} = 1, \epsilon_{\min} = 0.01$ and 7 available blocks. Optimal's line stops at $x = 200$ tasks because after that its execution never finishes. DPack takes slightly longer than DPF to run because it needs to solve multiple single-block knapsacks. Fig. 2.5b shows scheduler efficiency in number of allocated tasks as a function of the number of tasks in the system. DPF performs the worst, unable to efficiently schedule tasks across multiple blocks and varying alpha order demands. DPack matches Optimal (up to Optimal's 200 task limit) and schedules more tasks when it has a larger pool of tasks to choose from, since it can pick the most efficient tasks. Since the workload has a finite number of different tasks, as we increase the load, both schedulers reach a plateau where they allocate only one type of task.

2.7.3 Online Plausible Workload (Q3)

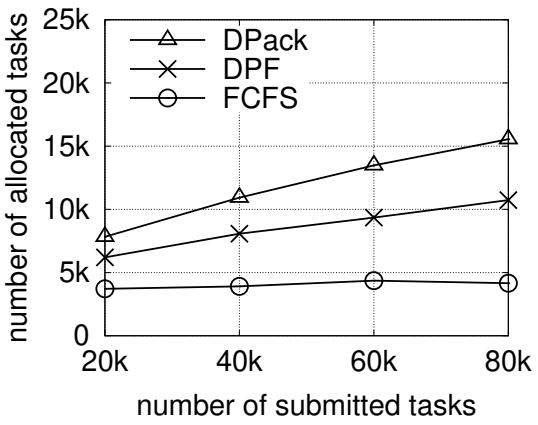
We now evaluate online scenarios where tasks and blocks arrive dynamically, and budget is unlocked over time. The simulator uses a virtual unit of time, where one block arrives each time unit. Tasks always request the m most recent blocks. For all the evaluated policies we run a batch

scheduler on the available unlocked budget, every T blocks.

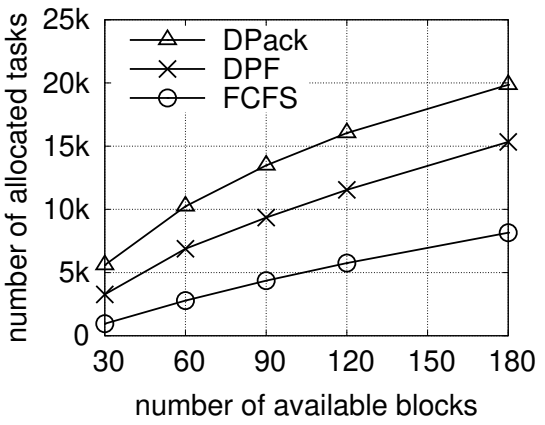
The Alibaba-DP Workload. We create a macrobenchmark based on Alibaba's GPU cluster trace [17]. The trace includes 1.1 million tasks submitted by 1,300 users over 3 months, and contains each task's resource demands and the resource allocation over time. We use these metrics as proxies for task DP budget demands, which do not exist in this trace.

We use machine type (CPU/GPU) as a proxy for DP mechanism type. We assume CPU-based tasks correspond to mechanisms used for statistics, analytics, or lightweight ML (e.g. XGBoost or decision trees [40, 41]), while GPU-based tasks correspond to deep learning mechanisms (DP-SGD or DP-FTRL [42, 43]). We map each CPU-based task to one of the {*Laplace, Gaussian, Subsampled Laplace*} curves and each GPU-based task to one of the {*composition of Subsampled Gaussians, composition of Gaussians*} curves, at random. We use memory usage as a proxy for privacy usage by setting traditional DP ϵ as an affine transformation of memory usage (in GB hours). We don't claim that memory will be correlated with privacy in a realistic DP workload, but that the privacy budget might follow a similar distribution (e.g. a power law with many tasks having small requests and a long tail of tasks with large requests). We compute the number of blocks required by each task as an affine function of the bytes read through the network. Unlike the privacy budget proxy, we expect this proxy to have at least some degree of realism when data is stored remotely: tasks that don't communicate much over the network are probably not using large portions of the dataset. Finally, all tasks request the most recent blocks that arrived in the system and are assigned a weight of 1. We truncate the workload by sampling one month of the total trace and cutting off tasks that request more than 100 blocks or whose smallest normalized RDP ϵ is not in $[0.001, 1]$. The resulting workload, called *Alibaba-DP*, is an objectively derived version of the Alibaba trace. We use it to evaluate DPack under a more complex workload than our synthetic microbenchmark or PrivateKube's also synthetic workload. We open-source Alibaba-DP at `https://github.com/columbia/alibaba-dp-workload`.

Q3: Does DPack present an efficiency improvement for plausible workloads? How much does it trade fairness? Fig. 2.6a shows the number of allocated tasks as a function of the number of



(a) Allocated tasks as a function of submitted tasks

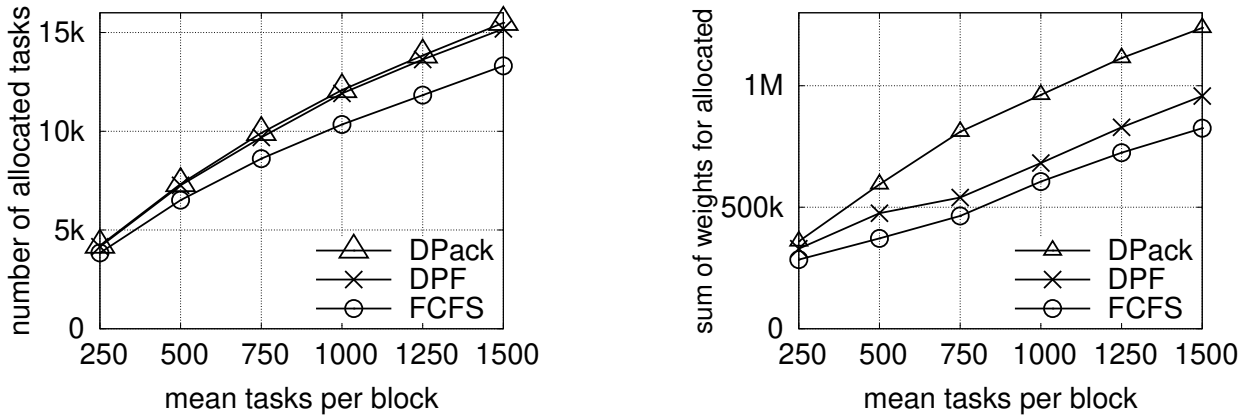


(b) Allocated tasks as a function of number of available blocks

Fig. 2.6: **(Q3) Efficiency evaluation on the online Alibaba-DP workload.** Number of allocated tasks as a function of (a) offered load for 90 blocks and (b) available blocks for 60k tasks. *Q3 Answer: Alibaba-DP exhibits sufficient heterogeneity for DPack to present a significant improvement (1.3–1.7×) over DPF.*

submitted ones from the Alibaba-DP workload. The results show that as the number of submitted tasks increases, both DPF and DPack can allocate more tasks, because they have a larger pool of low-demand submitted tasks to choose from. This is not the case with FCFS, which does not prioritize low-demand tasks. DPack allocates 22–43% more tasks than DPF, since it packs the tasks more efficiently. Similarly, Fig. 2.6b shows the number of allocated tasks as a function of the number of available blocks. As expected, all algorithms can schedule more tasks when they have more available budget. DPack consistently outperforms DPF, scheduling 30–71% more tasks. Across all the configurations evaluated in Fig. 2.6a and 2.6b, DPack outperforms DPF by 1.3–1.7×. The results confirm that Alibaba-DP, a workload derived objectively from a real trace, exhibits sufficient heterogeneity for DPack to show significant benefit.

Efficiency–Fairness Trade-off. While DPack schedules significantly more tasks than DPF on the Alibaba workload, this increased efficiency comes at the cost of fairness, when we use DPF's definition of fairness. To demonstrate this, we run the Alibaba workload with 90 blocks and 60k tasks, and set the DPF "fair share" of tasks to be $\frac{1}{50}$. This means that DPF will always prioritize tasks that request $\frac{1}{50}$ or less of the epsilon-normalized global budget. In the Alibaba trace, using this definition, 41% of tasks would qualify as demanding less or equal budget than their fair share.



(a) Original workload

(b) Workload with task weights

Fig. 2.7: **Evaluation on Amazon Reviews workload from [8].** (a) The original synthetic workload exhibits limited heterogeneity, so there is no room for DPack to improve over DPF. (b) Adding randomly selected weights to the tasks creates sufficient heterogeneity for DPack to show an improvement. Global efficiency is measured as the sum of weights of allocated tasks (y axis).

With DPack, 60% of the allocated tasks are fair-share tasks; with DPF 90% are. However, DPack can allocate 45% more tasks than DPF. As expected, this shows that optimizing for efficiency comes at the expense of fairness. In the case of privacy scheduling, however, due to the finite nature of the privacy budget, DPF's fairness guarantees are limited only to the first N fair share tasks (in the experiment, $N = 50$); the guarantees do *not* hold for later-arriving tasks. This makes the overall notion of fairness as defined by DPF somewhat arbitrary and underscores the merit of efficiency-oriented algorithms.

Another workload: Amazon Reviews [8]. We also evaluate on the macrobenchmark workload from the PrivateKube paper [8], which consists of several DP models trained on the Amazon Reviews dataset [44]. Unlike our Alibaba-DP, which is rooted in a real ML workload trace, this workload is completely synthetic and very small, and as a result, its characteristics may be very different from real workloads. Yet, for completeness, we evaluate it here, too. The workload consists two categories of tasks: 24 tasks to train neural networks with a composition of subsampled Gaussians, and 18 tasks to compute summary statistics with Laplace mechanisms. Unlike for our Alibaba-DP workload, task arrival needs to be configured for this workload; tasks arrive with a Poisson process and request the latest blocks. The Amazon Reviews workload has low hetero-

Scheduler	Number of allocated tasks
DPack	1269
DPF	1100

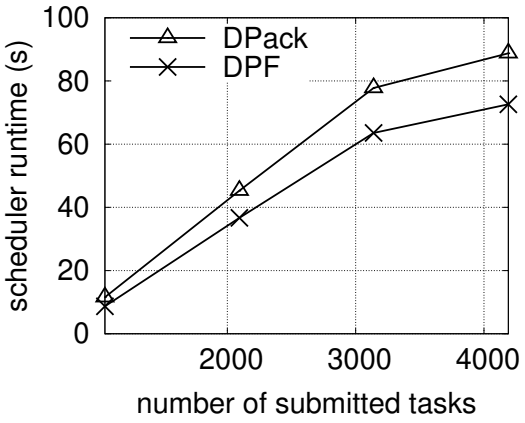
Tab. 2.2: **Efficiency on Kubernetes prototype with Alibaba-DP.**

geneity both in terms of block and the best-alpha variance. Although tasks request up to 50 blocks, 95% of the tasks in this workload request 5 blocks or fewer, and 63% of the tasks request only 1 block. Moreover, tasks have only 2 possible best alphas (4 or 5), with 81% of the tasks with a best alpha of 5. Hence, per our Q1 results in §2.7.2, we expect DPF to already perform well and leave no room for improvement for DPack. Fig. 2.7a confirms this: all schedulers perform largely the same on this workload.

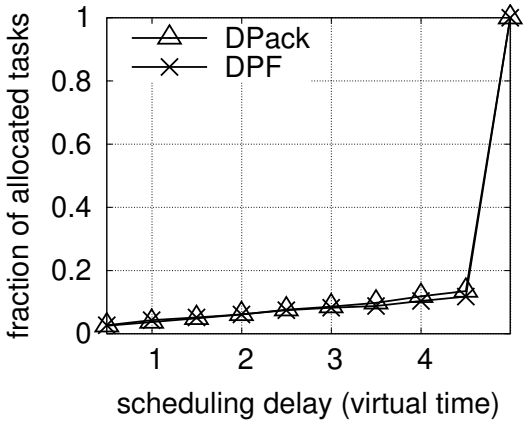
Next, without modifying the privacy budget or the blocks they request, we configure different weights for submitted tasks, corresponding to different profits the company might get if a task gets to run. We assume that large tasks (neural networks) are more important than small tasks. Then, we pick an arbitrary grid of weights while still allowing some small tasks to be more profitable than some large tasks. Weights are chosen uniformly at random from $\{10, 50, 100, 500\}$ for large tasks and $\{1, 5, 10, 50\}$ for small tasks. This change implicitly scales the number of requested blocks and increases heterogeneity. In terms of global efficiency, a task with weight k demanding m blocks is roughly similar to a task with weight 1 demanding m/k blocks. Instead of having most tasks request 1 block, tasks now demand a higher-variance weighted number of blocks (the variation coefficient is 1.9 instead of 1.3). Fig. 2.7b shows the global efficiency, measured as the sum of weights of allocated tasks, as a function of the number of submitted tasks. Recall that we also incorporate task weights in DPF (§2.4.1). Still, DPack now outperforms DPF by 9–50%.

2.7.4 Kubernetes Implementation Evaluation (Q4)

Q4: How does our implementation perform in a realistic setting? We evaluate the Alibaba-DP workload on our Kubernetes system. *Scheduler runtime:* We first estimate the scheduler's overhead by emulating an offline scenario, where all the tasks and blocks are available. To do so, we



(a) Scheduler runtime as a function of submitted tasks in the offline experiment ($T = 25$).



(b) CDF of scheduling delay (excluding scheduler runtime) for allocated tasks in the online experiment ($T = 5$).

Fig. 2.8: **(Q4) Evaluation on Kubernetes with Alibaba-DP.** DPack has only a modestly higher runtime than DPF, as system-related overheads dominate. In the online setting, scheduling delays are nearly identical across schedulers.

use a large $T = 25$. For this experiment, we generate a total of 4,190 tasks by sampling 2 days of the Alibaba cluster trace. The experiment shows the runtime as a function of the number of submitted tasks. It uses 10 offline and 20 online blocks. Fig. 2.8a shows the total time spent in the scheduling procedure, which includes Kubernetes-related overheads (e.g. inter-process communication and synchronization). As noted in §2.5a, DPack has a higher overhead since it solves knapsack subproblems. DPack has a higher runtime overhead than DPF since it has to recompute the efficiency of each task when the global state changes after a scheduling cycle, while DPF computes the dominant share of each task only once. Nevertheless, the overhead is modest, because: (a) the Kubernetes overheads dominate, and (b) the DPack (and DPF) algorithms are parallelized. In addition, since DP tasks are often long-running (e.g. distributed training of deep neural networks), the scheduling delay of DPack in many cases is insignificant compared to the total task completion time.

Scheduling delays and efficiency: We run an experiment to measure the scheduling delays (Fig. 2.8b) and efficiency (Table 2.2) in an online scenario on Kubernetes. We use the same workload and number of blocks as in Fig. 2.8a, with $T = 5$. As before, DPack is more efficient than DPF. Scheduling delay, measured in virtual time, excluding scheduler runtime, shows no signifi-

cant difference between the two policies.

2.8 Related Work

We have already covered the details of the most closely related works: DPF and related systems for privacy scheduling [9, 8, 10] (background in §2.3.3, efficiency limitations in §2.4.1 and §2.4.2, and experimental evaluation in §5.6). To summarize, we adopt the same threat and system models, but instead of focusing on fairness, we focus on efficiency because we believe that the biggest pressure in globally-DP ML systems will ultimately be how to fit as many models as possible under a meaningful privacy guarantee. The authors of Cohere [10] concurrently developed a privacy management system with novel partitioning and accounting features. They also investigate efficiency-oriented privacy resource allocation, but they rely on an ILP solver for scheduling, which is similar to our *Optimal* baseline (§2.7.1). Their optimal solver faces the same scalability issues we identified, unless tasks query non-overlapping block ranges, thus reducing the number of constraints in the privacy knapsack. Cohere supports DPack as an approximate scheduler, and the authors observe that "the DPK heuristic[2] achieves within 96% and 98% of optimal request volume and utility, respectively" on their workload. This further validates DPack.

Bin packing for data-intensive tasks. Multidimensional knapsack and bin packing are classic NP-hard problems [45, 46, 47]. In recent years, several heuristics for these problems have been proposed to increase resource utilization in big data and ML clusters [16, 11, 48, 49]. Some of these heuristics assign a weight to each dimension and reduce to a scalar problem with a dot product [31, 33]. We show that the Rényi formulation of differential privacy generates a new variation of the multidimensional knapsack problem, making standard approximations and heuristics unsuitable.

Scheduling trade-offs. Fairness and performance is a classic tradeoff in scheduling even in single-resource scenarios. Shortest-remaining-time-first (SRTF) is optimal for minimizing the average completion time, but it can be unfair to long-running tasks and cause starvation. Recent works have shown a similar fairness and efficiency tradeoff in the multi-resource setting [11]. Although

[2]DPack was known as DPK in a previous preprint of our paper.

max-min fairness can provide both fairness and efficiency for a single resource, its extension to multi-resource fairness [50] can have arbitrarily low efficiency in the worst case [12]. In this paper, we highlight the fairness-efficiency tradeoff when allocating privacy blocks among multiple tasks with RDP.

Differential privacy. The literature on *DP algorithms* is extensive, including theory for most popular ML algorithms (e.g. SGD [42, 51], federated learning [52]) and statistics (e.g. contingency tables [53], histograms [54]), and their open source implementations [55, 56, 57, 25, 26]. These lower-level algorithms run as tasks in our workloads. Some algorithms focus on workloads [58], including on a data streams [59], but they remain limited to linear queries. Some *DP systems* also exist, but most do not handle ML workloads, instead providing DP SQL-like [60, 61, 29] and MapReduce interfaces [62], or support for summary statistics [63]. Sage [9], PrivateKube [8] and Cohere [10], previously discussed, handle ML workloads.

2.9 Conclusions

This paper for the first time explores how data privacy should be scheduled efficiently as a computing resource. It formulates the scheduling problem as a new type of multidimensional knapsack optimization, and proposes and evaluates an approximate algorithm, DPack, that is able to schedule significantly more tasks than the state-of-the-art. By taking the first step of building an efficient scheduler for DP, we believe this work builds a foundation for tackling several important open challenges for managing access to DP in real-world settings, such as supporting tasks with different utility functions, investigating job-level scheduling, and better scheduling of traditional computing resources alongside privacy blocks.

Chapter 3: Turbo: Effective Caching for Differentially-Private Databases

3.1 Overview

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 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 Covid 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 provides a theoretical foundation and general system design for effective caching in DP databases.

3.2 Introduction

ABC 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 [64] and Aircloak [65], 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.

The preceding scenario, while fictitious, is representative of what often occurs 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 [66]. 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 [67]. In 2021, New York City excluded demographic information from datasets released from their CitiBike bike rental service, which could reveal sensitive user data [68]. The city's new, more restrained data release not only remains susceptible to privacy attacks 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 [69], including publishing trends in Google searches related to Covid [70], sharing LinkedIn user engagement statistics with outside marketers [71], enabling analyst access to Uber mobility data while protecting against insider attacks [72], and releasing the US Census' 2020 redistricting data [73]. To facilitate the application of DP, industry has developed a suite of systems, ranging from specialized designs like the US Census TopDown [73] and LinkedIn Audience Engagements [71] to more general DP SQL systems, like GoogleDP [29], Uber Chorus [72], and Tumult Analytics [30].

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 [74] 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 [64] and Aircloak's data [65] 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 [8]. When computation is expensive, *caching* is a go-to solution: it uses past results to save CPU on future computations. Caches are ubiquitous in all computing systems – from the processor to operating systems and databases – enabling scaling to much larger workloads than would otherwise be afforded with fixed resources. 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?* While DP theory has explored algorithms to reuse past query results to save privacy budget in future queries, there is no general DP caching system that is effective in common practical settings.

We propose *Turbo*, the first general and effective caching layer for DP SQL databases that boosts the number of linear queries (such as sums, averages, 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)* [75], 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 has compelling convergence guarantees in theory, we find it ineffective in practice, being overrun even by

an 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 both the exact-match cache and vanilla PMW 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 contribution on its own and can be used separately, or, as we do in Turbo, as part of the first *general, effective, and accurate DP-SQL caching design*. We prototype Turbo on TimescaleDB, a timeseries database, and use Redis to store caching state. We evaluate Turbo on workloads based on Covid 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 ABC that seek an effective DP SQL system to address their user data analysis and sharing concerns. We make Turbo available open-source at `https://github.com/columbia/turbo`, part of a broader set of infrastructure systems we are developing for DP, all

described here: https://systems.cs.columbia.edu/dp-infrastructure/.

3.3 Background

Threat model. We consider a threat model known as *centralized differential privacy*: one or more untrusted analysts 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 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. 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. Previous research demonstrated attacks under this threat model [66, 76, 77, 65, 64, 20].

Differential privacy (DP). DP [7] randomizes aggregate queries over a dataset to prevent membership inference and data reconstruction [78, 79]. 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 subset 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 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. As an example, for counting queries 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. The accuracy for such queries can be controlled probabilistically by converting it into the (ϵ, δ) parameters.

Answering multiple queries on the same data fundamentally degrades privacy [74]. 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 data subsets, their composition is $(\max(\epsilon_1, \epsilon_2), \max(\delta_1, \delta_2))$-DP and is called *parallel composition*. Using composition, one can enforce a global (ϵ_G, δ_G)-DP guarantee over a workload, with each DP query "consuming" part of a *global privacy budget* that is defined upfront as a system parameter [35].

Good values of the global privacy budget in interactive DP SQL systems remain subject for debate [80], but generally, an ideal value for strong theoretical guarantees is $\epsilon_G = 0.1$, while $\epsilon_G = 1$ are considered acceptable. Larger values are often considered vacuous semantically, since individuals' privacy risk grows with e^{ϵ_G}. In this paper, we aim to achieve values of $\epsilon_G = 1$ or smaller over a query workload.

Private multiplicative weights (PMW). PMW is a DP mechanism to answer online linear queries with bounded error [75]. We defer detailed description of PMW, plus an example illustrating its functioning, to §3.5 and only give here an overview. PMW maintains an approximation of the dataset in the form of a *histogram*: estimated counts of how many times any possible data point appears in the dataset. When a query arrives, PMW estimates an answer using the histogram and computes the *error of this estimate* against the real data in a DP way, using a DP mechanism called *sparse vector (SV)* [81] (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/Gaussian mechanism, consuming privacy budget as needed. It returns the DP result and also uses it to update the histogram for more accurate estimates to future queries.

An additional cost in using PMW comes from the SV, a well-known DP mechanism that can be used to test the error of a sequence of query estimates against the ground truth with DP guarantees

and limited privacy budget consumption [81]. We refer the reader to textbook descriptions of SV for detailed functioning [81] and provide here only an overview of its semantics. SV is a stateful mechanism that receives queries and estimates for their results one by one, and assesses the error between these estimates and the ground-truth query results. While the estimates have error below a preset threshold with high probability, SV returns success and consumes *zero privacy*. However, as soon as SV detects a large-error estimate, it requires a *reset*, which is a privacy-expensive operation that re-initializes state within the SV to continue the assessments. In common SV implementations, a reset costs as much as 3× the privacy budget of executing one DP query on the data.

The theoretical vision of PMW 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 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 [58]. However, no one has examined whether this worst-case bound is practical and if PMW outperforms natural baselines, such as an exact-match cache.

3.4 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 §3.5), which are widely used in interactive analytics and constitute the class of queries supported by approximate databases such as BlinkDB [82]. 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.

3.4.1 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) / (G4) *Provide worst-case convergence guarantees but optimize for empirical convergence:* We aim to maintain PMW's theoretical convergence (G3), but we prioritize for *empirical convergence* speed, a new metric that measures, on a workload, the number of updates needed to answer most queries for free (G4).

(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 benefit multiple important workload types, including static and streaming databases, and queries that arrive over time.

(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 §3.5 and empirical evaluations for (G4-G7) in §5.6.

3.4.2 Use Cases

The DP literature is fragmented, with different algorithms developed for different use cases. We seek to create a *general 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 larger workload of queries compared to existing approaches.

(2) and (3) Partitioned databases are less frequently investigated in DP theory literature, but important to distinguish in practice [83, 84]. 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.

(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.

For all three use cases, we aim to support *online workloads* of queries that are not all known upfront. As §4.8 reviews, most works on optimizing global privacy budget consumption operate in the *offline setting*, where all queries are known upfront. For that setting, algorithms are known

to answer all queries simultaneously with optimal use of privacy budget. However, this setting is unrealistic for real use cases, where analysts adapt their queries based on previous results, or issue new queries for different analyses. In such cases, which correspond to the *online setting*, we require adaptive algorithms that accurately answer queries on-the-fly. Turbo does this by making effective use of PMW, as we next describe.

3.4.3 Turbo Architecture

Fu4.3 shows the Turbo architecture. It is a caching layer that can be added to a DP SQL engine, like GoogleDP [29], Uber Chorus [72], or Tumult Analytics [30], to boost the number of linear queries that can be answered accurately under a fixed global DP guarantee. The filled components indicate our additions to the DP SQL engine, while the transparent components are standard in DP SQL engines. 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 (as do Chorus and Tumult Analytics) or sacrifices privacy by "resetting" the budget (as does LinkedIn Audience Insights). We assume 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. 4.3) 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

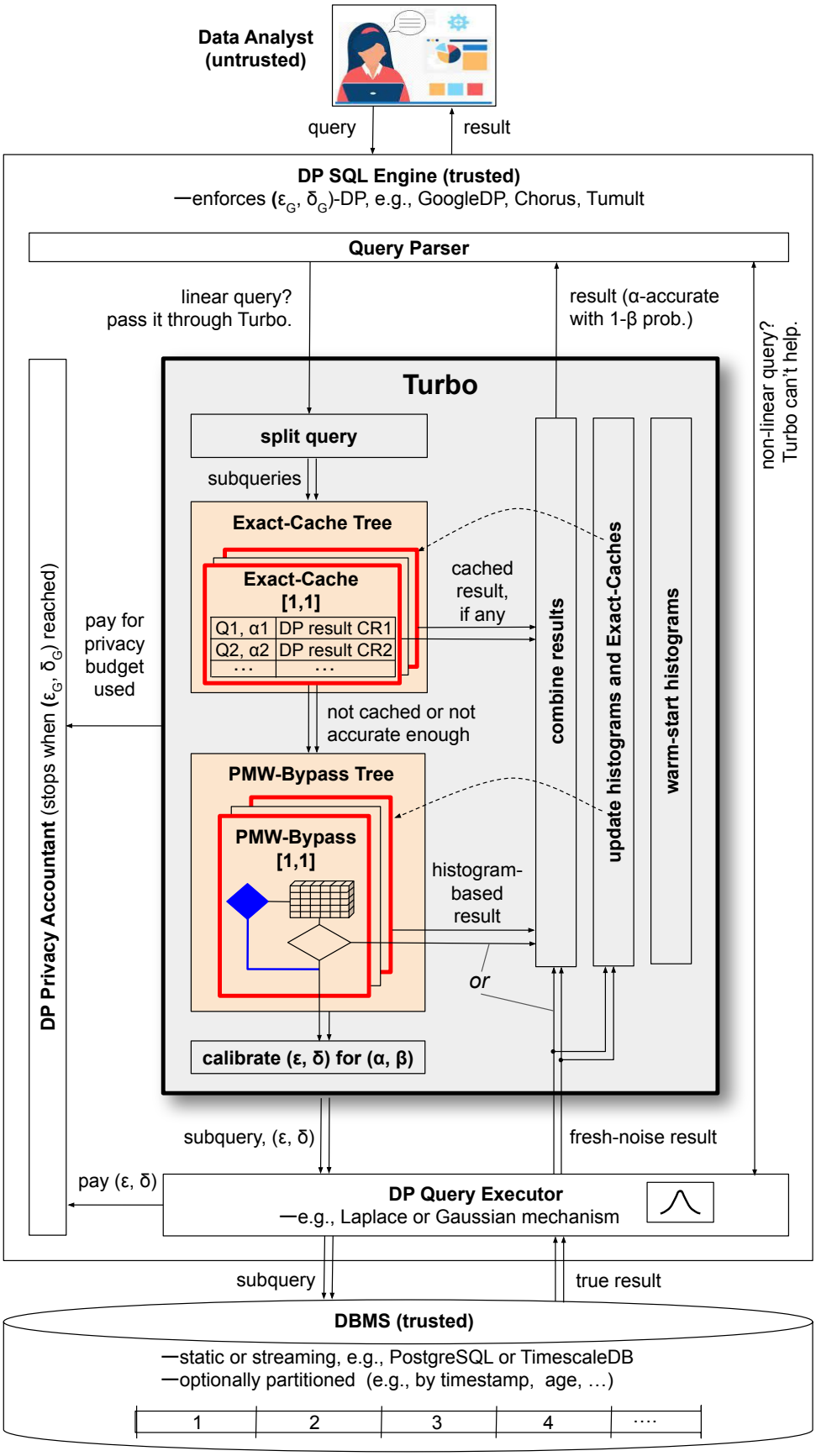


Fig. 3.1: **Turbo architecture.**

training phase of its histogram (§3.5.3). 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 time-series or streaming workloads, by supporting database partitioning. In theory, partitions could be defined by attributes with public values that are typically queried by range, such as time, age, or geo-location. In this paper, we will focus on partitioning by time. 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 (§3.5.4). 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 (α, β) accuracy parameters, using the "calibrate (ϵ, δ) for (α, β)" functional component in Fig. 4.3. 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 (§3.5.5). 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-starting give convergence properties similar to, but slightly slower than, vanilla PMW.

3.5 Detailed Design

We next detail the novel caching objects and mechanisms in Turbo, using different use cases from §3.4.2 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.

3.5.1 Notation

Our algorithms require some notation. Given a data domain $\mathcal{X}$, a database x with n rows can be represented as a histogram $h \in \mathbb{N}^{\mathcal{X}}$ as follows: for any data point $v \in \mathcal{X}$, $h(v)$ denotes the number of rows in x whose value is v. $h(v)$ is the *bin* corresponding to value v 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 that are equal to $(0,0,1)$. §4.3.1 exemplifies a database, its dimensions, and its histogram.

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

```
SELECT AVG(*) FROM ( SELECT q(A, B, C, ...) FROM Table ),
```

where `q` takes d arguments (one for each attribute of `Table`, denoted $A, B, C, ...$) and outputs a value in $[0,1]$. When `q` has values in $\{0,1\}$, a query returns the *fraction of rows satisfying*

(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 executing Q1, then Q2:

	A = 0	A = 1	A = 2	A = 3
P = 0	h(0,0): 12.5->18.3->**8** (real: 13) c: 1	h(0,1): 12.5->18.3->**21.7** (real: 27) c: 0	h(0,2): 12.5->18.3->**21.7** (real: 15) c: 0	h(0,3): 12.5->18.3->**21.7** (real: 25) c: 0
P = 1	h(1,0): 12.5->6.7->**2.9** (real: 3) c: 2	h(1,1): 12.5->6.7->**8** (real: 5) c: 1	h(1,2): 12.5->6.7->**8** (real: 8) c: 1	h(1,3): 12.5->6.7->**8** (real: 4) c: 1

Format: h(p,a): default-bin-value->value-after-Q1->**value-after-Q2 (current value)**
real: real value of the histogram bin (no DP, included as reference for h(v))
c: number of purposeful updates to the histogram bin

(d) Next query to execute:

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

Fig. 3.2: **Running example**. (a) Simplified Covid tests dataset with $n = 100$ rows and data domain size $N = 8$ for the two non-time attributes, test outcome P and subject's age bracket A. (b) Two queries that were previously run. (c) State of the histogram as queries are executed. (d) Next query to run.

predicate $\mathtt{q}$. To get raw counts, we multiply by n, which we assume is public information. PMW (and hence Turbo) is designed to support only linear queries. Examples of *non-linear* queries are: maximum, minimum, percentiles, top-k.

3.5.2 Running Example

Fig. 3.2 gives a running example inspired by our evaluation Covid dataset. Analysts run queries against a database consisting of Covid test results over time. Fig. 3.2(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. 3.2(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. 3.2(c) illustrates the histogram representation corresponding to the dataset, as estimated by the PMW algorithm, whose execution we discuss shortly. Fig. 3.2(d) shows the next query that will be executed, $Q3$, requesting the fraction of positive minors. $Q3$ is not identical to either $Q1$ or $Q2$, but it is *correlated* with both, as it accesses data that overlaps with both queries. Thus, while neither $Q1$'s nor $Q2$'s DP results can be used to directly answer $Q3$, intuitively, they both should help. That is the insight that PMW (and PMW-Bypass) exploits through its query-by-query build-up of a DP histogram representation of the database that becomes increasingly accurate in bins that are accessed by more queries.

Fig. 3.2(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 $h(p, a)$ field in the figure, for which we show the sequence of values it has taken following updates due to $Q1$ and $Q2$. Initially, $h(p, a)$ 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, $h(p, a)$ values are updated with DP results, depending 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 DP results to specific bins (we specify the update algorithm later when discussing Alg. 2). $Q1$ 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, t hese 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 $(P = 1, A = 2)$, updated only once, is even more accurate 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 is within α w.h.p. Assessing that the error is within α – privately, and 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.

3.5.3 PMW-Bypass

PMW-Bypass addresses practical inefficiencies of PMW, which we illustrate with simple demonstration.

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 with replacement 35K queries to form a workload; there is therefore some identical repetition of queries but not much. This workload is not necessarily realistic, but it 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 .3.3 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 more slowly compared to PMW's beginning. The PMW eventually becomes better than

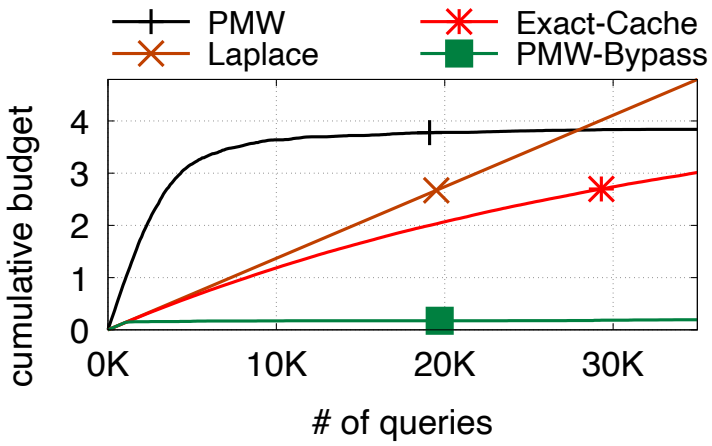


Fig. 3.3: **Demo experiment.**

Laplace, but only after $\approx 27K$ queries.

Moreover, if instead of always executing with 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 notably better than this simple baseline! This happens for a workload that should be ideal for PMW. §5.6 shows that for other workloads, less favorable for PMW but more realistic, the outcome persists: *PMWs underperform even the simplest baselines in practice*.

We propose *PMW-Bypass*, a re-design for PMWs that releases their power and makes them *very effective*. We make multiple changes 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 .3.3 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 others in §5.6, the outcome is the same: *our changes make PMWs very effective*. We thus believe that PMW-Bypass should replace PMW in most settings where the latter is studied, not just in our system's design.

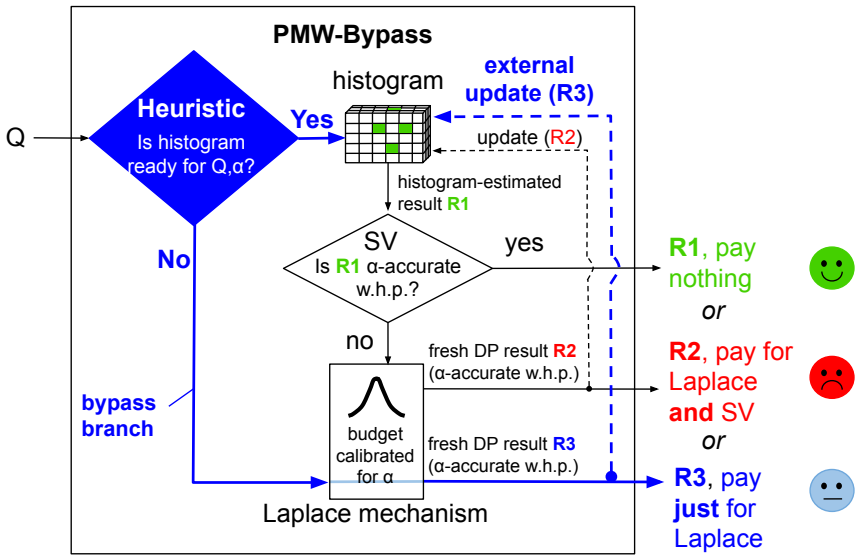


Fig. 3.4: **PMW-Bypass.** New components over vanilla PMW are in blue/bold.

PMW-Bypass. Fig .3.4 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 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 costs privacy. But beyond that, the SV must be *reset*, which consumes privacy. In total, if the 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 be used to update the histogram (the arrow denoted "update (R2)" in Fig .3.4), in hopes that future correlated queries "hit" in the histogram.

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 well, enabling the significant privacy budget savings in Fig .3.3.

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. We prove that performing the same kind of update as the PMW would do, from outside the protocol, would break its theoretical convergence guarantee. Thus, for PMW-Bypass, we design an *external update* procedure that *can* be used to update the histogram with $R3$ while preserving the PMW's worst-case convergence, albeit at slower speed.

Heuristic IsHistogramReady. One option to assess if a histogram is ready to answer a query accurately is to check if it has received at least C updates, for some global threshold C. However, this approach is often imprecise as it fails to detect histogram regions that might still be untrained. Thus, we 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 errs (i.e., predicts it is ready when it is in fact not ready for that query). While the threshold is too small, the heuristic gets penalized until it reaches a threshold high enough to avoid

mistakes. For queries that span multiple bins, we only penalize the least-updated bins to prevent a single, inaccurate bin from setting back the histogram from queries using accurate bins only. With these thresholds, we only configure initial parameters C_0 and S_0, which we find experimentally easy to do (§3.7.2).

Algorithm 2 **PMW-Bypass algorithm.**

Cfg.: PRIVACYACCOUNTANT, HEURISTIC, accuracy params (α, β), histogram convergence params lr, τ, database DATA with n rows.
function UPDATE(h, q, s)
    Update estimated values: $\forall v \in \mathcal{X}, g(v) \leftarrow h(v)e^{s*q(v)}$
    Renormalize: $\forall v \in \mathcal{X}, h(v) \leftarrow g(v)/\sum_{w\in\mathcal{X}} g(w)$
    return h
function CALIBRATEBUDGET(α, β)
    return $\frac{4\ln(1/\beta)}{n\alpha}$
Initialize histogram h to uniform distribution on $\mathcal{X}$
$\epsilon \leftarrow$ CALIBRATEBUDGET(α, β)
PRIVACYACCOUNTANT.PAY$(3 \cdot \epsilon)$ // Pay to initialize first SV
while PRIVACYACCOUNTANT.HASBUDGET() **do**
    $\hat{\alpha} \leftarrow \alpha/2 + \text{Lap } 1/\epsilon n$ // SV reset
    $SV \leftarrow$ NOTCONSUMED
    while $SV ==$ NOTCONSUMED **do**
        Receive next query q
        if HEURISTIC.ISHISTOGRAMREADY(h, q, α, β) **then**
            // Regular PMW branch:
            if $|q(\text{DATA}) - q(h)| + \text{Lap } 1/\epsilon n < \hat{\alpha}$ **then** // SV test
                Output $R1 = q(h)$ →**R1**, pay nothing
            else
                PRIVACYACCOUNTANT.PAY$(4 * \epsilon)$ → **R2**, pay for
                Output $R2 = q(\text{DATA}) + \text{Lap } 1/\epsilon n$ Laplace, SV
                // Update histogram (R2):
                $s \leftarrow \begin{cases} lr & \text{if } R2 > q(h) \\ -lr & \text{if } R2 < q(h) \end{cases}$
                $h \leftarrow$ UPDATE(h, q, s)
                $SV \leftarrow$ CONSUMED // force SV reset
                HEURISTIC.PENALIZE(q, h)
        else
            // Bypass branch:
            PRIVACYACCOUNTANT.PAY(ϵ) → **R3**, pay for
            Output $R3 = q(\text{DATA}) + \text{Lap } 1/\epsilon n$ Laplace
            // External update of histogram (R3):
            $s \leftarrow \begin{cases} lr & \text{if } R3 > q(h) + \tau\alpha \\ -lr & \text{if } R3 < q(h) - \tau\alpha \\ 0 & \text{otherwise // no updates if we're not confident!} \end{cases}$
            $h \leftarrow$ UPDATE(h, q, s)

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. 2). They follow a similar structure as a regular PMW update (lines 24-25 in Alg. 2), 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. 2).

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 even though it 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. Finally, like regular PMW updates, external updates reuse the already DP result $R3$, hence they do not consume any additional privacy budget beyond what was already consumed to generate $R3$.

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. 2). Prior PMW works fix learning rates that minimize theoretical convergence time, typically $\alpha/8$ [85]. 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. **(G2)**

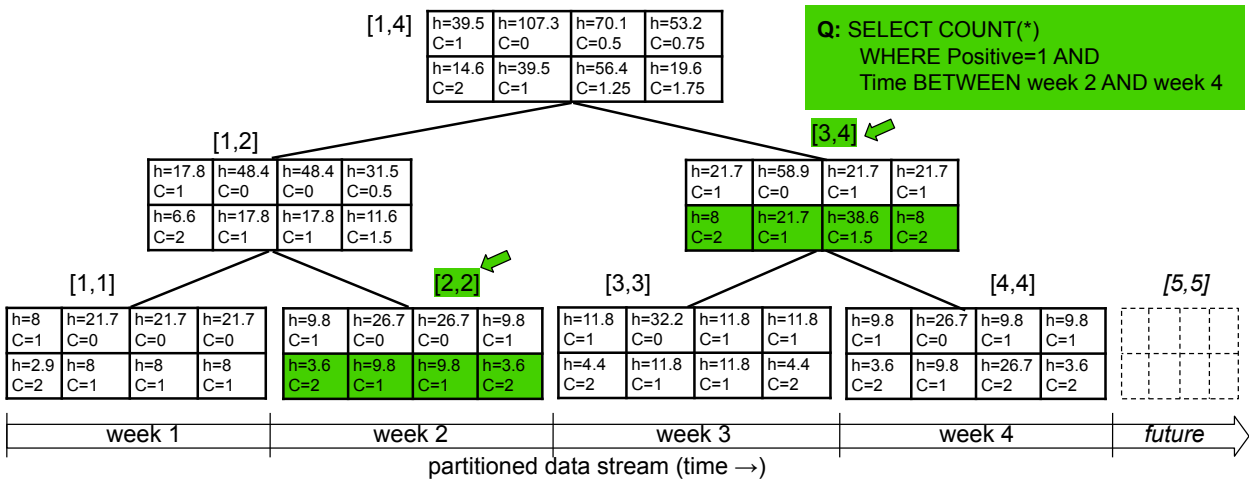


Fig. 3.5: **Example of tree-structured histograms.**

Accuracy: PMW-Bypass is α-accurate with $1-\beta$ probability for each query. This property stems from how we calibrate Laplace budget ϵ to α and β. This is function CALIBRATEBUDGET in Alg. 2 (lines 6-7). **(G3)** *Worst-case convergence:* 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. PMW-Bypass's worst-case convergence is thus similar to PMW's, but roughly $1/2\tau$ times slower. §3.7.2 confirms this empirically.

3.5.4 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. With this approach, a system can answer more queries under a fixed global (ϵ_G, δ_G)-DP guarantee compared to not partitioning [8, 9, 83, 62]. We implement support for partitioning and parallel composition in Turbo through a new caching object called a *tree-structured PMW-Bypass*.

Example. Fig .3.5 shows an extension of the running example in §4.3.1, 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 Laplace$(1/n\epsilon)$ has standard deviation $\sqrt{2}/n\epsilon$, 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. Another option would be to use *one histogram per range* (i.e. set of contiguous partitions), but that involves maintaining a large state that grows quadratically in the number of partitions.

Instead, our approach is to maintain a *binary-tree-structured set of histograms*, as shown in Fig .3.5. 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 . 3.6 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 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

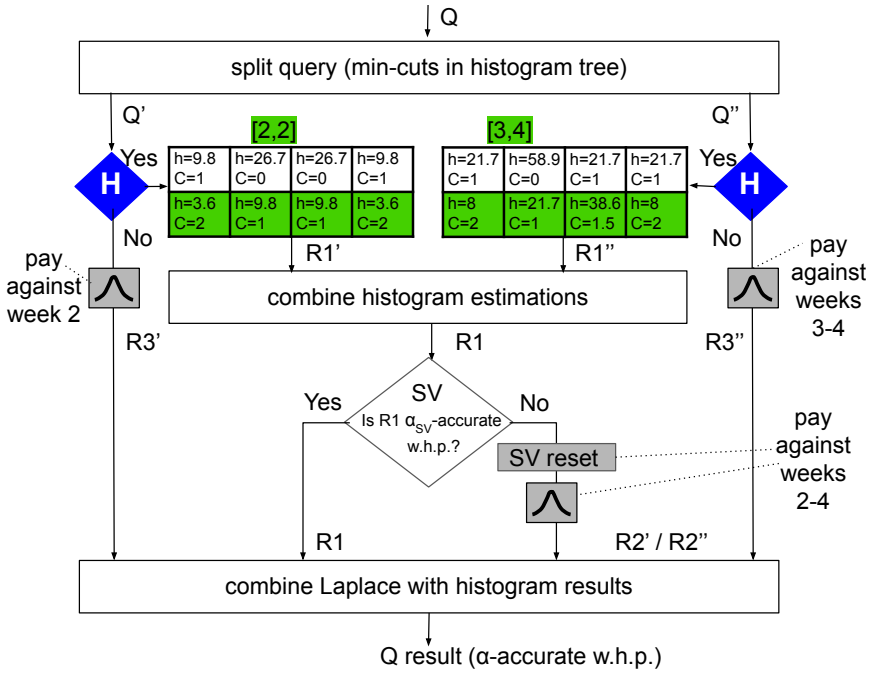


Fig. 3.6: **Tree-structured PMW-Bypass.**

mechanisms. We pay for any Laplace's and SV resets against the queried data partitions and finally combine Laplace results with histogram-based results. Each subquery updates the corresponding histograms of the tree and increments c for updated nodes.

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

3.5.5 Histogram Warm-Start

An opportunity exists in streams to warm-start histograms from previously trained ones to converge faster. Prior work on PMW initialization [86] 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. We also initialize the per-bin thresholds and update counters of each node.

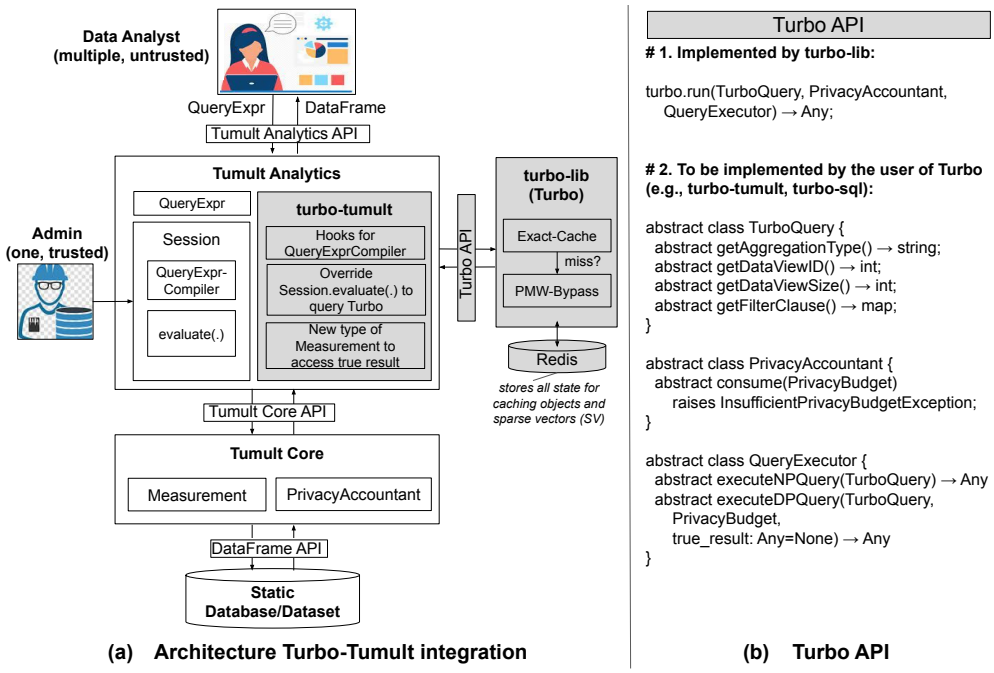


Fig. 3.7: **(a) Turbo integration into Tumult. (b) Turbo API.**

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. 2 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. **??**). The same properties hold for the tree.

3.6 Prototype Implementations

We prototype Turbo in three components that we release open-source: (1) *turbo-lib*, a library that contains Turbo-specific functionality, notably the caching objects and functional components in the Turbo architecture (Fig. 4.3); (2) *turbo-tumult*, a library that connects turbo-lib with Tumult Analytics, to add caching functionality into that existing DP system; and (3) *turbo-sql*, a basic standalone library to run a select subset of DP SQL queries through turbo-lib directly against a traditional, non-DP database, such as TimescaleDB or PostgreSQL. The reason for both (2) and (3) is that Tumult provides a more complete database query engine, supporting a wide variety of Spark-SQL-like queries while having significant limitations with respect to parallel composition on partitioned databases. Our integration with Tumult (2) shows that Turbo can be integrated with

a real, existing DP system, while our standalone querying library (3) can let us experiment with both non-partitioned and partitioned databases, in both static- and streaming-DB settings. We use a version of (3) (released through the SOSP'23 artifact) throughout our evaluation, but describe here predominantly our integration with Tumult, which can serve as a blueprint for integration with other existing DP systems in the future. Finally, we separately release the artifact that we used in our evaluation and which was evaluated by the SOSP'23 artifact evaluation committee. All are available from the repository: `https://github.com/columbia/turbo`.

Fig. 3.7(a) shows the architecture of our Turbo-Tumult integration. The grey boxes are Turbo-specific while the clear boxes are unchanged Tumult components.

Tumult overview. Without Turbo, Tumult functions as follows. It consists of two main components: (1) Tumult Core, a library that implements primitive DP mechanisms and privacy accounting; and (2) Tumult Analytics, a layer on top of Tumult Core that exports a higher-level, Spark-SQL-like query interface on top of one or more static datasets or databases. Tumult Core exports a low-level API consisting of a privacy accountant and a *measurement* abstraction, which is the Tumult terminology for a DP computation. It implements the necessary methods to "evaluate" a measurement on top of a dataset and deduct its privacy budget against the accountant. Tumult Analytics implements two main abstractions: (1) a *session*, which represents the context against which Tumult will enforce a global privacy budget across all queries issued against this session and (2) a Spark-SQL-like interface for analysts to construct queries that consists of multiple *transformations* chained one after another (such as filters, projections, joins, etc.) against one or more datasets, followed by a single *aggregate function* (such as an average, count, sum, median, percentiles, stdev, etc.), with potential for splitting and grouping the results by one or more attributes. Compared to other DP SQL databases, it is our impression that Tumult supports a fairly wide range of SQL that can be handled with DP.

For the purposes of this paper, we will assume that an administrator creates a session upfront, specifying a global privacy budget to be enforced and hosts this session as a service to guard analysts' accesses to a sensitive dataset (or datasets) underneath. Analysts, which can be many and

are untrusted, send their query expressions for execution against the session. The session is then responsible for executing each query by first compiling it into a measurement and then evaluating it through the Tumult Core, which will deduct the necessary privacy budget. While the measurement abstraction is a quite general representation of a DP computation, Tumult Core and Analytics assume a Spark DataFrame-based API for interacting with the dataset(s) underneath. Thus, measurements compiled through Tumult Analytics, will be Spark DataFrame queries – to be executed through Spark – in which Tumult Analytics transparently includes an additional operation that adds an appropriately scaled amount of noise to the result of the aggregation. A Tumult measurement encapsulates this compiled Spark DataFrame query, along with information regarding the privacy budget it is programmed to expend upon its execution. Tumult Analytics hands over this measurement for Tumult Core, which executes the DataFrame query through Spark and deducts the measurement's reported privacy consumption through its privacy accountant.

Turbo-Tumult. The preceding describes Tumult and its main abstractions (relevant for this paper) *without Turbo*. Tumult itself has no caching capabilities, so our integration aims to add Turbo as a caching layer in Tumult. The integration consists of two components, denoted in grey in Fig. 3.7(a). First is turbo-lib, which contains the core Turbo functionality we described in this paper. Turbo-lib exposes an API, Turbo API, consisting of the functions Turbo exports to and requires from any user of Turbo, such as turbo-tumult and turbo-sql. Second is turbo-tumult, a small library that incorporates Turbo into Tumult by invoking and implementing different parts of the Turbo API.

Turbo-tumult takes a *light-touch approach* to incorporating Turbo into Tumult, which ensures that our system is easily adoptable. It manifests in two ways. First, we only extend, but do not modify, certain classes within Tumult Analytics and implement new types of measurements to extend, but not change, Tumult Core functionality. Specifically, turbo-tumult provides one type of externally visible abstraction: a new type of session, called *TurboSession*, which overrides the original's query evaluation function to: (1) incorporate a set of hooks into the query compiler such that certain information necessary for Turbo is extracted from the query, such as the dataset ID,

the type of aggregate function, and the filtering conditions; and (2) if the query can be handled by Turbo, TurboSession passes it through turbo-lib instead of executing it directly on Tumult Core. Turbo-lib then checks its own caching objects for an answer, but resorts to Tumult Core – which it accesses back through the Turbo API, discussed shortly – for execution of the query and for privacy budget deduction in the Tumult Core accountant.

Second, we take a fail-to-Tumult approach for all queries. Turbo supports a small subset of all queries supported by Tumult: e.g., we do not support joins, medians, percentiles, and a number of transformation functions allowed in Tumult. Moreover, Turbo aims to control accuracy of the queries, and presently that accuracy must be specified upfront, when the cache (e.g., through TurboSession) service is created. Yet, analysts may wish to vary their accuracy targets per query, and in some cases may wish to specify privacy budgets rather than accuracies for a query. Finally, we support only certain types of DP mechanisms and definitions in our prototype, specifically those relying on Laplace, whereas Tumult supports more. Our approach to address these limitations without restricting analysts' interaction with Tumult is to consult the Turbo caches only when the queries exhibit properties we can handle, while resorting to Tumult-based execution when they do not. As a result, an analyst interacting with a TurboSession will not be restricted in terms of their queries or accuracy demands compared to interacting with a vanilla Tumult session, but Turbo will only conserve privacy budget for those queries that it can handle.

The preceding two approaches for light-touch integration of Turbo into Tumult ensure that our system can be easily adopted.

Turbo API. The Turbo API is the central component for integrating Turbo into real DP systems. Shown in Fig. 3.7(b), it consists of two components: functionality that Turbo-lib implements and users invoke to take advantage of its caches (specifically, the `run` function) and (2) several classes that users implement to provide Turbo with services it needs from the DP system with which it is integrated. Turbo needs three types of services from the DP system. First, it needs the ability to extract certain information about the query, such as: the type of aggregation and filter chain; a unique ID for the dataset (or partition or view over the dataset or partition) on which the query is run, as

Turbo's state is tied to a dataset/partition/view; and the number of records in that dataset (recall that our design assumes that the dataset size is public information). This information is supplied by implementing the `TurboQuery` interface, which wraps the original, DP-system-specific query structure into one that supplies the necessary information. For example, our turbo-tumult library wraps query expressions into a TurboQuery with this enhanced functionality.

Second, Turbo is *not* a query engine, so it needs the ability to execute a query through the original DP system. This is provided by implementing the QueryExecutor interface. A peculiarity of Turbo in this context, which was easy to implement in Tumult but which we anticipate may be non-trivial to implement in other DP systems, is that Turbo needs not only the ability to execute the query in a DP way, but also the ability to execute it without DP. Recall that Turbo's SV checks compares the histogram-based result to the *true result* of invoking the same query on the data without DP. Turbo thus needs access to this true result, a piece of functionality that typically DP databases do not offer publicly, for good, safety-related reasons. Still, in Tumult, due to its highly modular structure, we find that this functionality can be implemented without having to modify its code base. Specifically, turbo-tumult implements `QueryExecutor.executeNPQuery(.)` by defining a special type of measurement that does not, in fact, incorporate randomness into its aggregate and which reports as zero the privacy budget being used. This measurement is executed against the Tumult Core and returns the true result of the query. In turbo-lib, we take care to only leverage this sensitive result internally during the SV check in a DP way. Moreover, to optimize query execution in the case that the SV fails, turbo-tumult implements `QueryExecutor.executeDPQuery(.)` with the optional ability to reuse a non-private, true result previously obtained for the SV check. This is achieved by implementing another type of measurement, which, when executed by Tumult Core, will only apply the randomness operation to the given `true_result` and report the appropriate amount of privacy budget to be deducted by the Tumult Core's accountant.

Third, Turbo needs the ability to deduct the privacy budget consumed by the SV reset. This is supplied by implementing the Turbo PrivacyAccountant interface, with one function: `consume`. In turbo-tumult, we implement this interface by defining a third type of measurement, which does

not perform any computation but just consumes privacy. We believe that DP systems should export this kind of functionality to more naturally support extensions.

Turbo-lib. Turbo-lib implements the Turbo design described in this paper, with some notable restrictions. First, we do not yet support partitioning in the turbo-lib implementation, though that support exists in our SOSP artifact release, as used in our evaluation. Second, our implementation only supports count queries presently, although our histograms and exact-match caches can be extended to support other types of linear aggregations, such as sums, averages, standard deviation. Third, we use Redis to store all state in Turbo, including the exact-match caches, PMW histograms, and SV state. Redis can be replaced with a persistent, consistent and durable storage service for production use.

Turbo-sql. In addition to incorporating Turbo into the Tumult Analytics engine, we are also creating a basic, standalone, SQL DP database ourselves, which only supports the types of queries that Turbo supports, but which can support streaming and partitioning. At the time of this writing, the most mature version of this library can be found in our SOSP artifact release, but we are working on a more modular version of this library that presently lacks support for streaming and partitioning. The full-featured version of this library, which is what we use in our evaluation, receives simple linear SQL queries as strings, parses them, implements the Turbo API to first check for answers to them in the Turbo cache, and execute the queries – DP through Laplace or non-DP (as needed by Turbo) – using TimescaleDB, a streaming version of PostgreSQL.

3.7 Evaluation

We evaluate Turbo using the SOSP artifact version of our own, dedicated DP SQL database with Turbo support incorporated in it. We use two public timeseries datasets – Covid and CitiBike – to evaluate Turbo in the three use cases from §3.4.2. 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 §3.4.1 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.

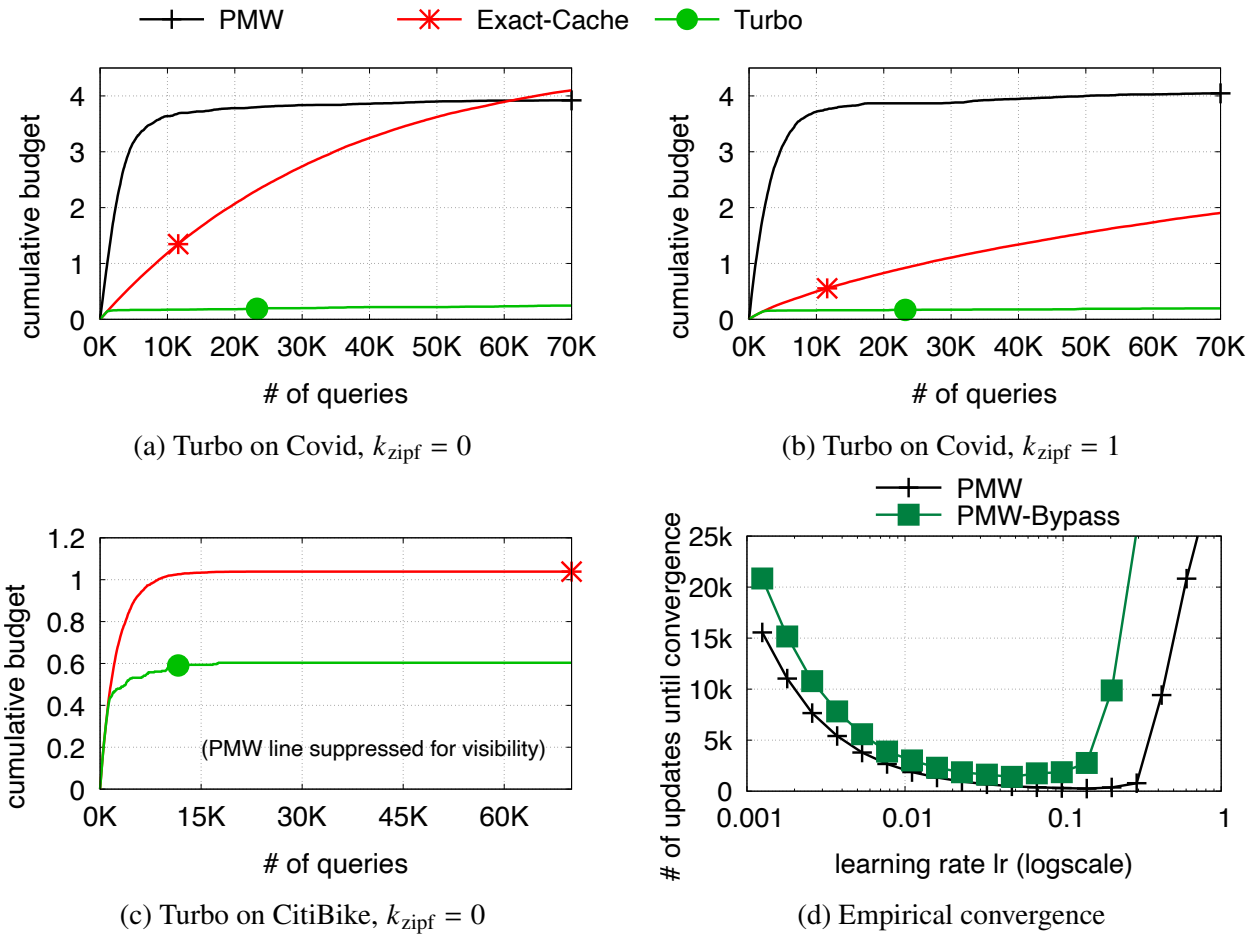
(3) Partitioned streaming database: We configure Turbo with the tree-structured PMW-Bypass, Exact-Cache, and histogram warm-up, letting us evaluate warm-up.

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 for time-series workloads where 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)**). Finally, we provide a basic runtime and memory evaluation, which shows that while Turbo performs reasonably for our datasets, further research is needed for larger-domain data.

3.7.1 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 Zipfian distribution. Covid uses a completely synthetic query pool. CitiBike uses a pool based on real-user queries from prior CitiBike analyses. We use the former as a microbenchmark, the latter as a macrobenchmark.

Covid. *Dataset:* We take a California dataset of Covid-19 tests from 2020 that provides daily *aggregate information* of the number of Covid tests and their positivity rates for various demographic



(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. 3.8: **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.

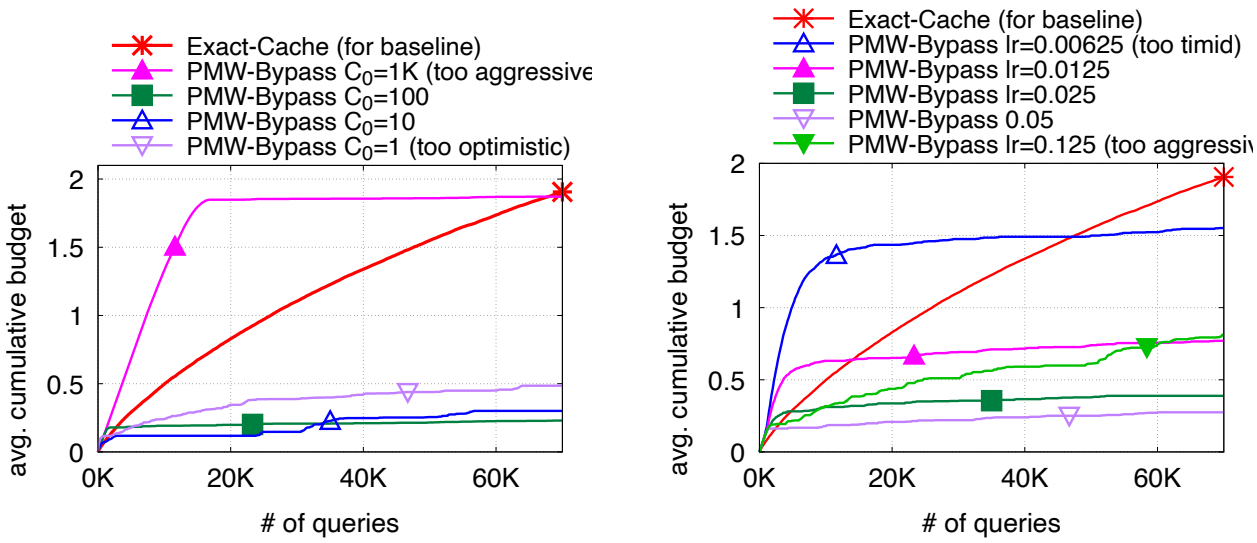
groups defined by age × gender × ethnicity. We combine this data with US Census data to generate a synthetic dataset that contains $n = 50,426,600$ per-person test records, each with the date and four attributes: positivity, age, gender, and ethnicity. These attributes have domain sizes of 2, 4, 2 and 8, respectively, so the dataset domain size is $N = 128$. The dataset spans 50 weeks, so in partitioned use cases we have up to 50 partitions. *Query pool:* We create a synthetic and 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.

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 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 [87]. An example is here [88]. We extract 30 distinct queries, most containing 'GROUP BY' statements that we decompose into multiple *primitive queries* that can interact with Turbo histograms. This gives us a pool of $2,485$ queries, which is smaller than Covid's but more realistic and suitable as a macrobenchmark.

Workload generation. As is customary in caching literature [89, 90, 91], we use a Zipfian distribution to control the skewness of query distribution, which affects hit rates in the exact-match cache. From a pool of Q queries, a query of type $x \in [1, Q]$ is sampled with probability $\propto x^{-k_{\text{zipf}}}$, 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. For streaming, queries arrive online with arrival times following a Poisson process; they request a window of certain size over recent timestamps.

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



(a) Impact of heuristic C_0 ($S_0 = 1$)

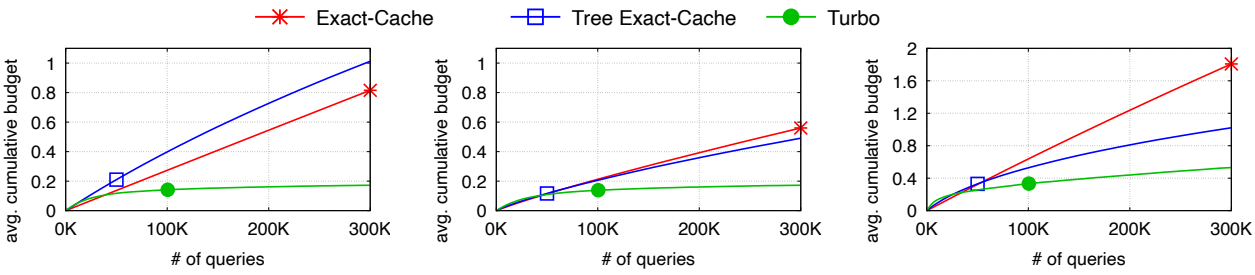
(b) Impact of learning rate lr

Fig. 3.9: **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), gives poor performance.

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 $\tau = 0.01$}.

3.7.2 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. 3.8a-3.8c 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×! 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× (Fig. 3.8a). Here, queries are relatively unique, resulting in low hit rate for the Exact-Cache. That hit rate is higher for the skewed workload (Fig. 3.8b), leaving less room for



(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. 3.10: **Partitioned static database: system-wide evaluation (Question 5).** 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.

improvement for Turbo: 9.7× better than Exact-Cache. For CitiBike (Fig. 3.8c), 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× improvement over Exact-Cache. And in this workload, Turbo outperforms PMW by 37.4× (omitted from figure for visualization reasons). Overall, then, Turbo significantly reduces privacy budget consumption in non-partitioned databases, achieving 1.7 – 15.9× improvement over the best baseline for each workload (goal **(G5)**).

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 at slower speed (§3.5.3). Fig. 3.8d compares the *empirical convergence* (defined in §4.7.1) of PMW-Bypass vs. PMW, as a function of the learning rate lr. 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 [85], we find that empirically, larger values of lr (e.g., $lr = 0.05$, which is 8× larger) give much 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. 3.9 shows their effects. *Heuristic C_0 (Fig. 3.9a):* 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. 3.9b):* 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)**).

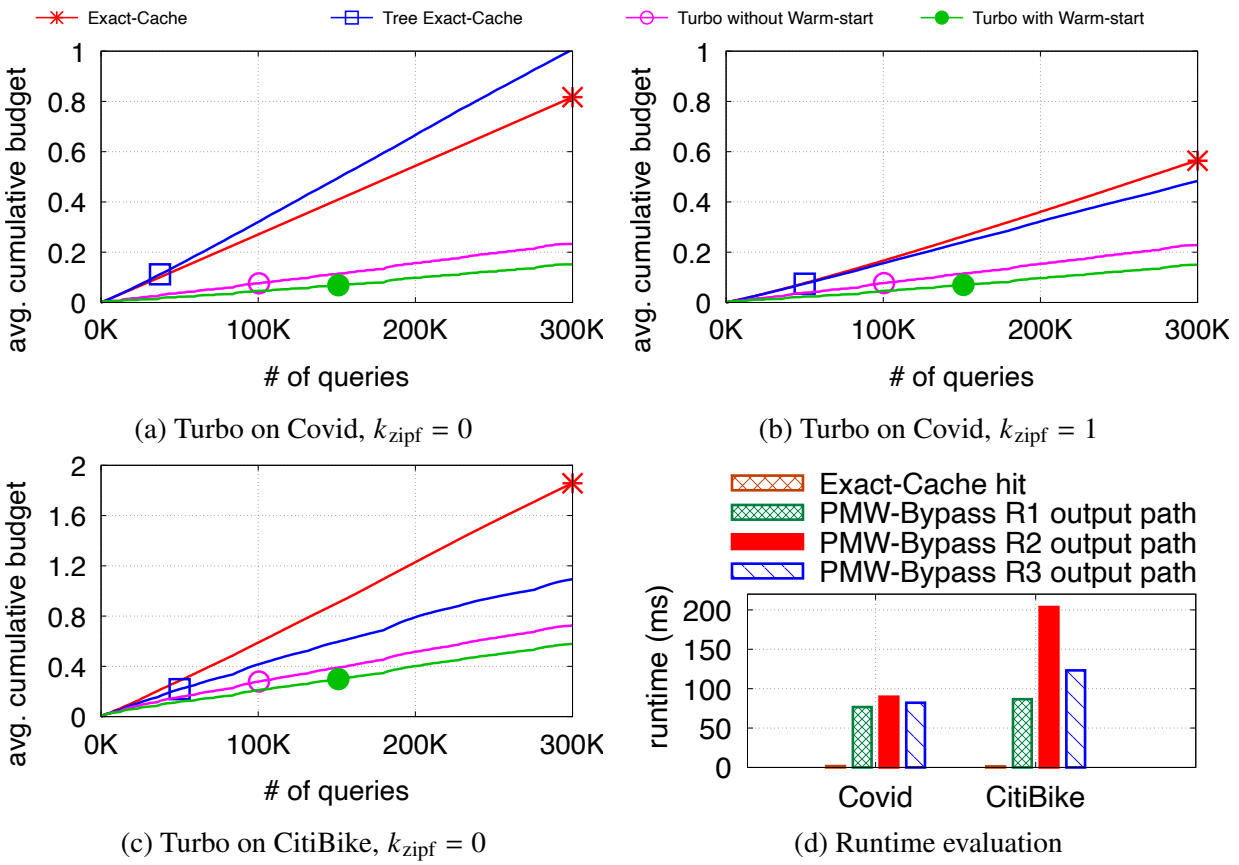
Question 4: How does Turbo's adaptive, per-bin heuristic compare to alternatives? We experimented with three alternative IsHistogramReady designs that forgo either (1) the per-bin granular thresholds, or (2) the adaptivity property, or (3) both. We make two observations. First, the *coarse-grained heuristics* consume more privacy budget than the fine-grained heuristics, especially on more skewed workloads, such as $k_{\text{zipf}} = 1.5$, which have less diversity so they tend to train histogram bins less uniformly. For example, a coarse-grained heuristic that uses a histogram-level count of the number of updates, with a threshold C_0 to determine when the histogram is ready to receive *any* query, consumes at best 0.7 global privacy budget on a Covid workload with $k_{\text{zipf}} = 1.5$; this is achieved when C_0 is optimally configured to a value of 2070 updates. In contrast, a fine-grained heuristic, which uses a per-bin update count with a threshold C_0 for each bin, consumes at best 0.44 global privacy budget, achieved when C_0 is set to 160 updates. Second, the *adaptive heuristics* consume similar budget as the optimally-configured, non-adaptive ones, but the former are much easier to configure, as they offer stable performance around wide ranges

of the C_0 parameter. For example, when C_0 varies in range $[20, 200]$, the non-adaptive per-bin heuristic's budget consumption varies in range $[0.44, 0.81]$ for the $k_{\text{zipf}} = 1.5$ workload, and in range $[0.31, 0.76]$ for $k_{\text{zipf}} = 1$ workload. In contrast, Turbo's adaptive, per-bin heuristic's budget consumption varies in much tighter ranges under the same circumstances: $[0.44, 0.52]$ and $[0.28, 0.48]$ for the $k_{\text{zipf}} = 1.5$ and $k_{\text{zipf}} = 1$ workload, respectively. Thus, Turbo's heuristic is the best of these options.

3.7.3 Use Case (2): Partitioned Static Database

System-wide evaluation. *Question 5: 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. We adjust the (C_0, S_0) heuristic parameters to $(50, 1)$ for Covid and $(1, 1)$ for CitiBike. Fig. 3.10a-3.10c show the average budget consumed per partition up to 300K queries. Compared to the static case, Turbo can now support more queries under $\epsilon_G = 10$ thanks to parallel composition: each query only consumes privacy from the accessed partitions. Turbo further divides privacy budget consumption by $1.9 - 4.7\times$ 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 6: When 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



(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. 3.11: **(a-c) Partitioned streaming database: system-wide consumed budget (Question 7); (d) PMW-Bypass runtime in non-partitioned setting (Question 8).** (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.

of partitions.

3.7.4 Use Case (3): Partitioned Streaming Database

System-wide evaluation. *Question 7: In streaming databases partitioned by time, does Turbo significantly improve privacy budget consumption compared to baselines? Does warm-start help?* Fig. 3.11a-3.11c show Turbo's 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 $1.5 - 3.5\times$ at the end of the workload. With warm-start, Turbo gives $1.9 - 5.4\times$ improvement over the best baseline for each workload, showing its effectiveness for the streaming use case. When there is a large variety of unique queries the tree-structured Exact-Cache has a significantly better hit-rate than the Exact-Cache baseline and performs better (Fig. 3.11a). In Fig. 3.11b and 3.11c the query pool is considerably smaller. Both baselines have a good enough hit-rate while the tree-structured Exact-Cache needs to consume more privacy budget to compensate for the aggregation error which makes it perform worse. This concludes our evaluation across use cases (goal **(G6)**).

3.7.5 Runtime and Memory Evaluation

Question 8: What are Turbo's runtime and memory bottlenecks? We evaluate Turbo's runtime and memory consumption to identify areas of improvement. Fig. 3.11d shows the average runtime of Turbo's main execution paths in a non-partitioned database. 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 experiment in the partitioned streaming case and find the same bottlenecks: TimescaleDB for Covid, histogram operations for CitiBike. Finally, we report Turbo's memory consumption in the streaming case with 50 partitions: 5.21MB for Covid and 1.43GB for CitiBike. For context, the raw datasets occupy on disk 600MB and 795MB, respectively. Thus, Turbo's memory overhead is significant and it is caused by the PMWs. The next section discusses this limitation and proposes potential directions to address it.

3.8 Discussion

We discuss several of Turbo's strengths and weaknesses. Turbo provides benefits when queries overlap in the data they access, i.e., new queries access histogram bins that have been accessed by past queries. The functions computed atop these bins can differ among queries (e.g., the new query can compute an average while all the past ones computed count fractions). If there is no data overlap in the queries, then Turbo does not give any benefit and comes with memory/computational costs. This is typical for caching systems: they only help if the workload has some level of locality.

A key strength in Turbo is its support for dynamic workloads, both new queries and new data arriving in the system. First, Turbo adapts seamlessly to changing queries. In the worst case, the new queries will access completely "untrained" regions within a histogram. Our heuristic will detect this and trigger a new cycle of external updates. In more moderate cases, the workload will touch a mix of "trained" and "untrained" regions. This will yield a mix of hits and misses in the heuristic, and Turbo will use just the right amount of privacy budget to adapt to these slower workload changes. Second, thanks to histogram warm-start, Turbo adapts to new data partitions arriving into the system with minimal privacy budget consumption: as new partitions arrive, their histograms are initialized from past ones and then fine-tuned for the new data by a few external updates. This way, the new histograms will quickly start serving query answers for free, conserving privacy budget. Still, there is a limitation: while we support new data arriving in the system, we do not support updates on past data; such updates would result in our heuristics predicting less accurately when the histogram can answer a query, and thus in more expensive SV failures.

By far, Turbo's biggest limitation is the memory consumed to maintain the PMW histograms. Each histogram is a RedisAI vector whose size grows with data domain size N, i.e., exponentially in data domain dimension d (N and d are defined in Section 3.5.1). With T partitions and k queries, Turbo maintains a binary tree of such histograms, which means it stores $\approx 2TN$ scalar values. By comparison, the Tree Exact-Cache baseline stores at most $\log(T)k$ scalar values, a much lower memory consumption. This impacts not only the scale of the datasets that can be

handled with Turbo, but also the runtime performance of Turbo-mediated queries. Indeed, as shown in the preceding section, histogram operations for CitiBike are the bottleneck in runtime due to the relatively high domain size. Some techniques have previously been proposed to address this rather fundamental challenge for PMW [92]. However, for even larger-scale deployments, we believe that it will be worth considering PMW alternatives that may not offer as compelling convergence guarantees as PMW but which are much more lightweight. One example may be the relaxed adaptive projection (RAP) [93], which builds a lightweight representation of the dataset by learning a small subset of representative data points using gradient-descent. One would have to be willing to forfeit the theoretical convergence guarantees to use this mechanism, and to develop an adaptive version of RAP to support realistic systems settings involving dynamic workloads and data. Even so, some of the core concepts we have proposed in this paper may transfer to this new design, including passing RAP-based estimations through an SV to ensure result accuracy while incorporating a heuristic-based bypass to avoid expensive failures in the SV.

We also touch on several potential vulnerabilities. First, an adversary may craft queries that consume budget by generating cache misses. The convergence proofs in §**??** provide a bound on how much such queries can affect budget consumption when a straightforward cutoff parameter is configured upfront. Second, response time can be a side-channel, which we leave out of scope but should be addressed in the future. Third, n, the number of elements in the database (or in each partition), is considered public knowledge. This can leak information and should be addressed by consuming some of the budget to compute n privately, as done in [9].

Regarding integration of Turbo with a real system, Tumult, we find that it can be done with ease, thanks to Tumult Core's extensible measurement API. We anticipate that such integration will not be as easy or "light touch" in other DP systems we have seen, and in general we see a gap in the core primitives that DP systems (SQL or not) should implement to support extensions such as Turbo; these might include providing direct access to the privacy accountant, decoupling the accountant from the query executor, and others. We encourage the community to work to articulate this set of key primitives, which we suspect will be useful in other extensions beyond

Turbo.

3.9 Related Work

This paper presents the first design, implementation, and evaluation for a *general, effective, and accurate DP-caching system for interactive DP-SQL systems*. In computer systems, caching is a heavily-explored topic, with numerous algorithms and implementations [94, 95, 96], some pervasively used in processors, operating systems, databases, and more. However, traditional forms of caching differ significantly from DP caching, justifying the need for a specialized approach for DP. The primary purposes of traditional caching are to conserve CPU and to improve throughput and latency; for these purposes, existing caches can be readily reused in DP systems. However, DP caching aims to conserve privacy budget, which requires a new design to be truly effective. For example, layering Redis on a DP database to cache query results would save CPU, but for privacy it would be equivalent to the "Exact-Cache" baseline that our evaluation shows is less effective than Turbo. This paper thus builds upon general traditional caching concepts – such as the two-layer design, the principle of generality in supporting multiple workloads – but develops a cache specialized in conserving DP budget.

To our knowledge, no existing DP system incorporates such a specialized caching system. Most DP systems do not incorporate caching capabilities at all [83, 97, 30, 71, 98, 99]; [29] explicitly leaves the design of an effective DP cache for future work. Some DP systems incorporate what amounts to an Exact-Cache by deterministically generating the same noise upon the arrival of the same query. Three systems consider more sophisticated mechanisms for DP result reuse: PrivateSQL [100], Chorus [72], and CacheDP [101]. But the result reuse components in these systems suffer from such significant limitations that they cannot be considered general and effective caching designs. **PrivateSQL** [100] takes a batch of "representative" offline queries and precomputes a private synopsis that answers them all. If new queries arrive (online), PrivateSQL uses the synopsis to answer them in a best-effort way, without accuracy guarantees. It does not learn on-the-fly from them, so it is unsuited for online workloads and does not support data streams.

Chorus [72] provides a trivialized implementation of MWEM, a variant of PMW, however the implementation only works for databases with a *single attribute*. The paper does not evaluate the MWEM-based implementation, nor integrates it as a caching layer. **CacheDP** [101] is an interactive DP query engine and has a built-in DP cache that answers queries using the Matrix Mechanism [102]. Our experience with the CacheDP code suggests that it is not a general, effective, or accurate caching layer for DP databases. First, CacheDP's implementation only scales to a few attributes and does not support parallel composition on data partitions; this suggests that it is not general enough to support a variety of workloads. Second, the "Tree Exact-Cache" baseline with which we compare in evaluation matches, to our understanding, the CacheDP design while scaling to the higher-dimension datasets and streaming workloads we evaluate against. Our evaluation shows Turbo more effective than Tree Exact-Cache.

While DP caching are under-explored in systems, the topic of optimizing global privacy budget for a query workload is heavily explored in theory. Approaches include generating synthetic datasets or histograms that can answer certain classes of queries, such as linear queries, with accuracy guarantees and no further privacy consumption [75, 103, 104, 93, 92, 86]; and optimizing privacy consumption over a batch of queries by adapting the noise distribution to properties of the queries [102, 105, 106]. Apart from PMW [75], all these methods operate in the offline setting, where queries are known upfront. This setting is unrealistic, as discussed in §3.4.2.

All of the theory works cited above, including PMW, suffer from another limitation: they operate on static datasets and do not support new data arriving into the system. PMWG [107] is an extension of PMW for dynamic "growing" databases, but operates in a setting where all queries request the *entire database*. This precludes the use of parallel composition for queries that access less than the entire database, such as queries over windows of time. Other algorithms focus on continuously releasing specific statistics over a stream, such as the streaming counter [108] that inspired our tree structure, and extensions to top-k and histogram queries [109]. These works do not support arbitrary linear queries, and they answer all predefined queries at every time step while we only pay budget for queries that are actually posed by analysts.

3.10 Conclusion

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 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 increase the number of answered queries.

Chapter 4: Cookie Monster: Efficient On-device Budgeting for Differentially-Private Ad-Measurement Systems

4.1 Overview

With the impending removal of third-party cookies from major browsers and the introduction of new privacy-preserving advertising APIs, the research community has a timely opportunity to assist industry in qualitatively improving the Web's privacy. This paper discusses our efforts, within a W3C community group, to enhance existing privacy-preserving advertising measurement APIs. We analyze designs from Google, Apple, Meta and Mozilla, and augment them with a more rigorous and efficient differential privacy (DP) budgeting component. Our approach, called *Cookie Monster*, enforces well-defined DP guarantees and enables advertisers to conduct more private measurement queries accurately. By framing the privacy guarantee in terms of an individual form of DP, we can make DP budgeting more efficient than in current systems that use a traditional DP definition. We incorporate Cookie Monster into Chrome and evaluate it on microbenchmarks and advertising datasets. Across workloads, Cookie Monster significantly outperforms baselines in enabling more advertising measurements under comparable DP protection.

4.2 Introduction

Web advertising is undergoing significant changes, presenting a major opportunity to enhance online privacy. For years, numerous entities, often without users' knowledge, have exploited Web protocol vulnerabilities, such as third-party cookies and remote fingerprinting, to track user activity across the Web. This data has been used to target individuals with ads and assess ad campaign performance. Two key shifts are reshaping this landscape. First, major browsers are making it more difficult to track users across websites. Apple's Safari and Mozilla's Firefox blocked third-

party cookies in 2019 [110] and 2021 [111], respectively, while Google Chrome will soon facilitate users' choice of disabling these cookies [112]. Additionally, browsers are strengthening defenses against IP tracking [113] and remote fingerprinting [111, 114, 115].

Second, acknowledging the critical role online advertising plays in the Web economy – and the impossibility of perfect tracking protection – browsers are introducing explicit APIs to measure ad effectiveness and enhance ad delivery while protecting individual privacy. Early designs, like Apple's PCM [116] and Google's FLoC [117], focused on intuitive but not rigorous privacy methods, resulting in limited adoption due to poor utility [118] or privacy [119]. Recently, browsers have shifted to theoretically-sound privacy technologies – such as differential privacy (DP), secure multi-party computation (MPC), and trusted execution environments (TEEs) – in the hope of achieving better privacy-utility tradeoffs.

However, substantial challenges remain in implementing these privacy technologies at Web scale. The research community now has a timely opportunity – and responsibility – to assist industry in refining these technologies to deliver both strong privacy protections and meet advertising needs. Only by addressing these challenges can we hope to drive adoption of privacy-preserving APIs, remove incentives for individual tracking, and meaningfully improve Web privacy.

This paper focuses on our efforts to analyze and enhance current *ad-measurement APIs* (a.k.a., attribution-measurement APIs), which enable advertisers to measure and optimize the effectiveness of their ad campaigns based on how often people who view or click certain ads go on to purchase the advertised product. While separate *ad-targeting APIs* are also under development [120], we concentrate on *ad-measurement APIs*.

The W3C's Private Advertising Technology Community Group (PATCG) [121] is working towards an interoperable standard for private ad-measurement APIs. Leading proposals include Google's Attribution Reporting API (ARA) [122], Meta and Mozilla's Interoperable Private Attribution (IPA) [123], Apple's Private Ad Measurement (PAM) [124], and a hybrid proposal [125]. Our first contribution is a systematization of these proposals into abstract models, followed by a comparative analysis to identify opportunities for improving their privacy-utility tradeoffs (§4.3).

We focus on the differential privacy (DP) component, present in all four systems. DP is used to ensure advertisers cannot learn too much about any single user through measurement queries. Each system employs a *privacy loss budget*, accounting for the privacy loss incurred by each query. Once the budget is exhausted, further queries are blocked. This process, called *DP budgeting*, is handled centrally in IPA, but in the other systems, DP budgeting is done separately by each device. We observe that this *on-device budgeting* cannot be formalized under standard DP and instead requires a variant, *individual DP* (IDP) or personalized DP [126], for proper formalization. Our formal modeling and analysis of on-device budgeting under IDP form our second contribution (§4.5).

Through our IDP formalization, we uncover optimizations that enhance utility in on-device budgeting systems, allowing advertisers to execute more accurate queries under the same DP budget. IDP enables devices to maintain their own, separate DP guarantees and to account for privacy loss based on the device's data. This lets a device deduct zero privacy loss if it lacks relevant data for a query. Notably, one such optimization is already used in ARA, though without formal justification. Our third contribution is providing formal proof for this optimization as well as other, novel optimizations that can further improve the privacy-utility tradeoff.

Our final contribution is a prototype implementation of our optimized DP budgeting system, called *Cookie Monster*, integrated into ARA within Chrome (§4.4, §4.6). Cookie Monster is the first ad-measurement system to enforce a fixed, user-time DP guarantee [127], improving on the event-level guarantees of ARA. We evaluate Cookie Monster on microbenchmarks and advertising datasets (§5.6), showing that it delivers $\times$1.16–2.88 better query accuracy compared to a user-time version of ARA and substantially outperforms IPA, which exhausts its budget very early. Our prototype is available at `https://github.com/columbia/cookiemonster` and has been incorporated into a W3C draft report on privacy-preserving attribution from Mozilla [128].

4.3 Review of Ad-Measurement APIs

We review the designs of privacy-preserving ad-measurement systems considered for a potential interoperable standard at PATCG: Meta and Mozilla's IPA, Google's ARA, Apple's PAM, and Meta and Mozilla's Hybrid. ARA and IPA are implemented; PAM and Hybrid exist only as design docs. We abstract their functionality for comparison and articulate the improvement opportunity addressed in this paper.

4.3.1 Example Scenario

We use a fictitious scenario to illustrate the motivation and requirements of ad-measurement systems from two key perspectives: Ann, a web user, and Nike, an advertiser measuring ad campaign effectiveness. While real-world players like first-party ad platforms (e.g., Meta) and ad-techs (e.g., Criteo) typically run measurement queries on behalf of advertisers, for simplicity, we assume the advertiser performs its own measurements.

User perspective. Ann visits various *publisher* sites, such as nytimes.com and facebook.com, where she sees ads. She understands that ads fund the free content she enjoys and occasionally finds them useful, like when she clicked on a Nike ad for running shoes on nytimes.com and later purchased a pair. However, Ann values her privacy and expects *no cross-site tracking*, meaning no site should track her across different websites. She also expects *limited within-site linkability*, preventing even a single site from linking her activities across cookie-clearing browsing sessions (e.g., incognito sessions). Ann accepts that some privacy loss is necessary for effective advertising but expects it to be *explicitly bounded* and *transparently reported* by her browser.

Fig. 4.1 shows a screenshot of the privacy loss dashboard we developed for Cookie Monster in Chrome, where Ann can monitor the privacy loss resulting from various sites and intermediaries querying her ad interactions, including *impressions* (e.g. ad views and clicks) and *conversions* (e.g. purchases, cart additions). While Ann may not grasp the concept of differential privacy that underpins the reported privacy loss, she trusts her browser to enforce protective bounds on it.

Fig. 4.1: **Privacy loss dashboard.** Screenshot from our Chrome implementation of Cookie Monster (minimally edited for visibility).

Advertiser perspective. Nike runs multiple ad campaigns for its running shoes, some emphasizing shock-absorbing technology, others focusing on aesthetics. Nike seeks to understand which campaigns perform best across different demographics and contexts (e.g., publisher sites, content types). In the past, Nike used third-party cookies and device fingerprinting[1] to track individuals from ad impressions to purchases, attributing purchase value using an *attribution function*, such as last-touch (giving all credit to the last impression) or equal credit (splitting value among recent impressions). Using such *attribution reports* from many users, Nike measured the purchase value attributed to different campaigns and optimized future ad targeting.

Now that third-party cookies are disabled on multiple browsers and fingerprinting is harder, Nike is transitioning to ad-measurement APIs, expecting similar attribution measurements with comparable accuracy. Nike understands that ad measurement has always involved some imprecision (e.g., due to cookie clearing or fraud), so its expectation of accuracy from these APIs is not stringent. Nike plans to conduct numerous attribution measurements over time to adjust to chang-

[1]The example is fictitious, as are claims regarding the companies mentioned.

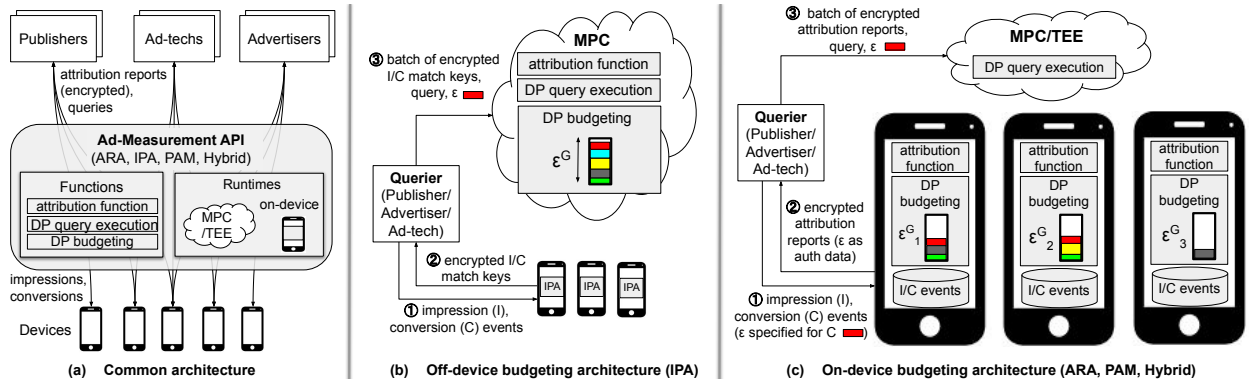


Fig. 4.2: **Architectures of ad-measurement systems.** Common structure, with a key difference in where attribution and DP budgeting occur: off-device (IPA) vs. on-device (ARA, PAM, Hybrid).

ing user preferences and product offerings. These measurements are single-advertiser summation queries, a key query type that ad-measurement systems aim to support.

4.3.2 Ad-Measurement Systems

IPA, ARA, PAM, and Hybrid aim to balance user privacy with utility for advertisers and other Web-advertising parties (referred to as *queriers*). Utility is defined as the number of accurate measurement queries a querier can execute under a privacy constraint. Despite variations in terminology, privacy properties, and mechanisms, these systems share key similarities. A commonality is the use of DP techniques, with ARA focusing on event-level DP, while IPA, PAM, and Hybrid emphasize user-time DP. This paper focuses on user-time DP, applied per querier site, as defined in §4.5.2.

Common architecture. The high-level architecture of all four systems is similar (see Fig. 4.2a). All systems act as intermediaries between user devices and sites. Previously, these parties collected impression and conversion events directly, matched them through third-party cookies, performed attribution, and aggregated reports. To break these privacy-infringing direct data flows, ad-measurement systems interpose a DP querying interface over impression and conversion data.

All systems include three core components: (1) the *attribution function*, which matches conversions to relevant impressions on the same device and assigns conversion value to impressions based on an attribution logic like last-touch; (2) *DP query execution*, which aggregates reports and

adds noise for DP guarantees; and (3) *DP budgeting*, which tracks privacy loss from each query using DP composition and enforces a maximum on total privacy loss, called a *DP budget*.

A key difference is where these components are executed. In IPA, all components run off-device within an MPC involving multiple helper servers. In ARA, PAM, and Hybrid, attribution and DP budgeting occur on-device, while DP query execution is off-device, in an MPC (PAM, Hybrid) or TEE (ARA). The MPC/TEE is trusted not to leak inputs, and the devices are trusted to safeguard their own data. The placement of attribution and DP budgeting is crucial for this paper.

Off-device budgeting (IPA). Fig. 4.2b illustrates IPA, which operates in a standard centralized-DP setting. The MPC handles all three functions, while the device's role is limited to generating a *match key* to link impressions and conversions. For example, when nytimes.com sends an ad for Nike shoes to Ann's device ①, the device responds with a match key, secret-shared and encrypted toward the MPC helper servers ②. When Ann later purchases the shoes on nike.com, her device sends the same key to the MPC, also secret shared and encrypted toward the helpers. Periodically, NYtimes sends batches of encrypted impression match keys to Nike, who cannot directly match these with its conversion match keys due to the encryption and secret sharing. Instead, Nike collects its conversion match keys and NYtimes' impression match keys into batches and submits them to the MPC, specifying the privacy budget ϵ to spend on the query ③. The MPC checks the budget, matches impressions to conversions, applies the attribution function with an L^1 cap for sensitivity control, aggregates the data, and adds DP noise to enforce ϵ-DP. The MPC tracks and deducts Nike's privacy budget, refusing further queries once the budget is exhausted until the per-site budget is "refreshed" (e.g., daily).

On-device budgeting (ARA, PAM, Hybrid). Fig. 4.2c shows the on-device architecture, which operates in a rather non-standard DP setting. While DP query execution occurs centrally on the MPC or TEE, attribution and DP budgeting are done *separately on each device*. Every device maintains a timeseries database of impression and conversion events. When Ann sees an ad for Nike on nytimes.com, her device records it locally ①. Later, when she buys shoes on nike.com, Nike requests an attribution report from her device. Ann's device checks its database for relevant

impressions, applies the attribution function with an L^1 cap, and sends an *attribution report* ②, either secret-shared and encrypted toward the helper parties (for MPC) or directly encrypted to a TEE. Nike aggregates attribution reports from multiple users, submits them to the MPC or TEE, which performs DP aggregation, adding noise based on Nike's ϵ parameter ③. The MPC/TEE ensures each report is used only once for sensitivity control.

DP budgeting in on-device systems differs from centralized DP by accounting for privacy loss when the advertiser requests a conversion report, prior to query execution. When Nike requests a report, it specifies the ϵ parameter for the future query. The device checks Nike's budget locally, generates and encrypts the report (with secret sharing if MPC is used), includes ϵ as authenticated data, and deducts ϵ from Nike's local budget. Since the budget is spent at the device, each report can only be used once, so the device includes a unique nonce with every report in authenticated data and the MPC/TEE tracks report nonces to prevent reuse.

Threat models. The threat models differ based on whether an MPC or TEE is used. In all cases, MPC/TEE systems are trusted to protect inputs and intermediate states. For MPC, the deployment models assume either a three-party, malicious, honest-majority MPC protocol (IPA, Hybrid) [123] or a two-party malicious protocol (PAM). The querier selects MPC parties from a browser-configured list, typically relatively trusted Web organizations like Cloudflare. The device secret shares the report and encrypts it toward the chosen parties after report generation.

4.3.3 Improvement Opportunity

On-device budgeting systems offer certain advantages over off-device systems but also present a key challenge, which we aim to address. First, on-device systems can enhance user transparency by putting the user's device in control of per-site budgets and the tracking of privacy losses incurred by the user due to specific attribution reports the device releases to various querier sites, as seen in the Cookie Monster privacy loss dashboard (Fig. 4.1). In contrast, in IPA, the device can only track the encrypted match keys returned by the device, not the specific privacy losses users incur through subsequent matching and aggregation in the MPC.

Second, on-device systems allow for finer-grained budgeting. While off-device systems enforce a global site-wide budget ϵ^G, on-device systems maintain a per-device budget ϵ^G_d, which is only consumed for queries involving that device. This granularity enables Nike, for instance, to continue querying other users' reports even if it exhausts Ann's budget. However, this behavior requires formalization under the less standard (but equally protective) privacy definition known as individual DP (IDP) [126], which allows enforcement of a separate privacy guarantee for each device.

The challenge lies in formalizing the data, query, and system model that capture the behavior of on-device ad-measurement systems, and in proving its IDP properties. This formalization then opens opportunities for further optimizing DP budgeting in on-device systems by deducting privacy loss based on the device's data. However, it also requires keeping the remaining privacy budgets on each device private, as revealing these budgets leaks data. This paper presents a formally-justified, practical and efficient DP budgeting module, *Cookie Monster*, designed for on-device systems like ARA, PAM, and Hybrid, which maximizes utility while maintaining DP guarantees.

4.4 Cookie Monster Overview

The design of Cookie Monster is guided by three principles. First, it must enforce well-defined DP guarantees at an industry-accepted granularity. We adopt a fixed "user-time" DP guarantee for each querier, supported by IPA, PAM, and Hybrid, and recognized by Apple, Meta, and Mozilla as the minimum acceptable. Second, Cookie Monster must support similar use cases and queries as existing systems.

Finally, Cookie Monster must not introduce new vectors for illicit tracking, given increasing browser efforts to prevent tracking both across sites and within-site across cookie refreshes.

Fig. 4.3 presents Cookie Monster's architecture with an example execution overlaid. We describe each aspect below.

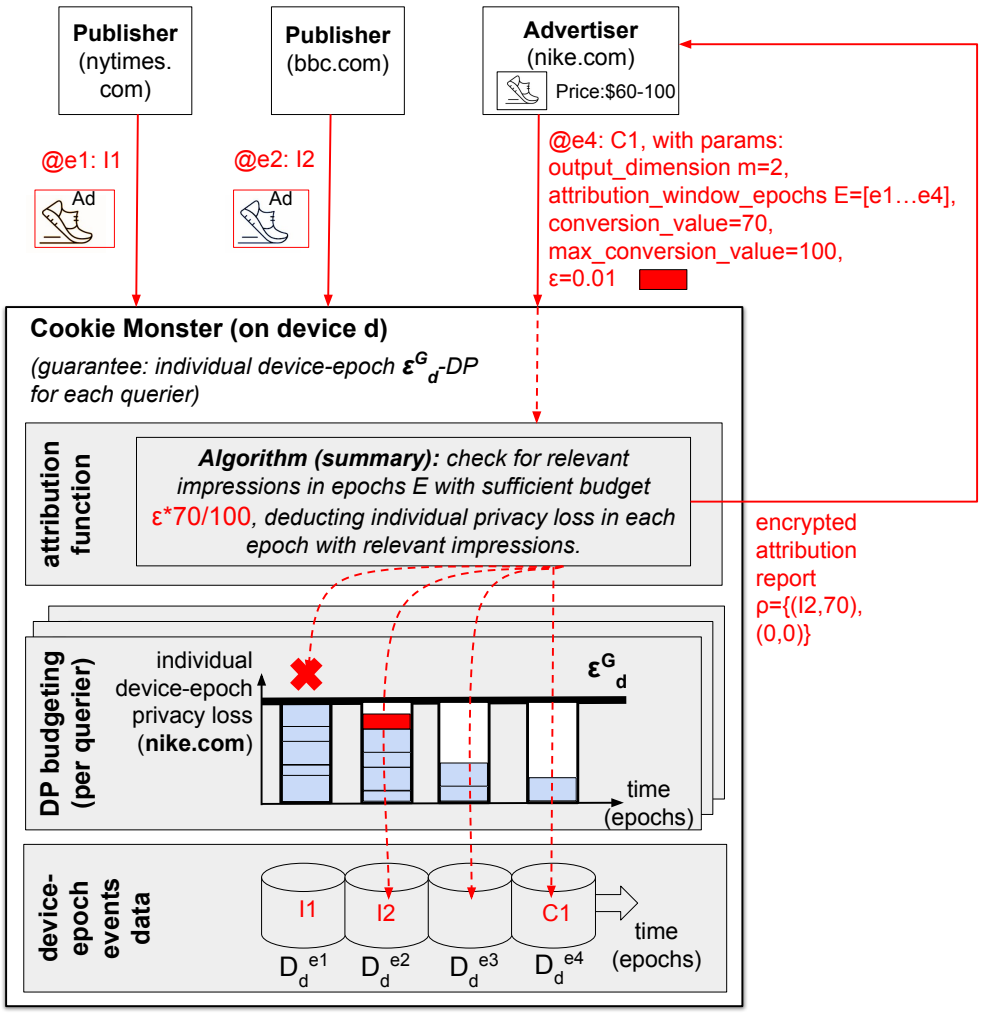


Fig. 4.3: **Cookie Monster architecture and example execution (red overlay).** §4.4.1 describes the architecture and §4.4.2 the example execution. Notation: $@e_1 : I_1$ indicates that Ann's device receives an impression I_1 of a Nike shoe ad from nytimes.com in epoch e_1. Red dotted arrows show the attribution function's search for impressions over epochs $e_1 - e_4$.

4.4.1 Architecture

Cookie Monster adopts on-device budgeting, similar to ARA, PAM, and Hybrid. DP query execution occurs off device, in an MPC or TEE, trusted not to leak inputs or intermediate states. Since Cookie Monster does not modify this component, it is omitted from Fig. 4.3; we think of it as a trusted *aggregation service*. Cookie Monster modifies the on-device component, based on ARA in our prototype. While the external APIs remain unchanged, we modify: (1) the on-device events database to support a "user-time" guarantee, and (2) the internals of the attribution function

and DP budgeting to enforce this guarantee efficiently.

Cookie Monster enforces *individual device-epoch* ϵ_d^G*-DP* for each querier site, formally defined in §4.5.2. This device-epoch granularity aligns with traditional "user-time" from DP literature [127, 9, 8], though we rename it to reflect that a user's complete activity is not directly observable by a device or browser, the scope in which Cookie Monster operates. We partition the on-device events database into time-based *epochs*, such as weeks or months. In each epoch e, device d collects impression and conversion events into a *device-epoch database* D_d^e. Queriers submit multiple queries over time, accessing data from one or more epochs. For each epoch e, Cookie Monster ensures that no querier learns more about device d's data in e than permitted by an ϵ_d^G-DP guarantee.

The *DP budgeting* in Cookie Monster is implemented using privacy filters [129], which ensure that the cumulative privacy loss from a series of queries does not exceed a pre-specified budget. For each querier, Cookie Monster maintains multiple filters – one for each device-epoch database. Fig. 4.3 shows these filters for nike.com. Each filter is initialized with a privacy budget ϵ_d^G and monitors cumulative privacy loss for queries involving data from that epoch.

In on-device systems, privacy loss is accounted for when the attribution report is generated, not when the query is executed. The *attribution function* is responsible for generating these reports. Upon a conversion, the function checks for relevant impressions in the device-epoch databases within a specified attribution window. Privacy filters prevent use of impression data from epochs with insufficient budget.

For epochs with sufficient budget, the filter allows access to the device-epoch data and deducts privacy loss. Under standard centralized DP, this loss would be ϵ, the DP parameter enforced later by the MPC or TEE during aggregation. However, our theoretical analysis of on-device budgeting reveals that viewing the system under an individual-DP lens opens opportunities to optimize privacy accounting, often allowing deductions of "less than ϵ." §4.5 outlines our theoretical analysis, a major contribution in this paper. We dedicate the remainder of this section to providing the systems view of our theory, including an execution example (§4.4.2), Cookie Monster's algorithm,

which is backed by our theory (§4.4.3), and a discussion on mitigating IDP-induced bias (§4.4.4).

4.4.2 Execution Example

The red overlay in Fig. 4.3 illustrates the attribution function's operation for the example from §4.3.1. Ann receives two impressions of Nike shoe ads: one in epoch e_1 and another in e_2, with no impressions in e_3. Later, in epoch e_4, Ann buys the shoes, and nike.com registers a conversion $C1$. It requests an attribution report with parameters: the set of epochs E to search for impressions, the maximum number of impressions m to attribute value to, the conversion value (\$70), and ϵ, the privacy parameter enforced by the MPC or TEE when executing the aggregation query.

The shoes' price ranges by color, with a maximum of \$100. While Ann's conversion is \$70, Nike's query will include conversions up to \$100. Thus, for a summation query with the Laplace mechanism, the noise added to the aggregate depends on $100/\epsilon$, where 100 is the *global sensitivity* of the summation (i.e., the largest change *any* device-epoch can contribute). Ann, with a purchase of \$70, can only contribute up to \$70 across her device-epochs.

Here, IDP lets us optimize privacy loss based on *individual sensitivity*, the maximum change that *a specific* device-epoch can make on the query output. In this case, Ann's device only deducts $\epsilon' = \$70/\$100 * \epsilon$ from the privacy filters of the epochs in the attribution window E. This is one optimization enabled by IDP. Another is that if no relevant impressions exist in an epoch (e.g., e_3 in Fig. 4.3), we need not deduct anything, since the individual sensitivity for that epoch is 0 and thus its privacy loss is also 0. §4.5.3 formalizes global and individual sensitivities and details further optimizations.

In Fig. 4.3, Cookie Monster's attribution function checks epochs $e_1 - e_4$ for relevant impressions. In e_1, access to data $D_d^{e_1}$ is denied because the filter has exhausted nike.com's budget. In e_2, the filter allows access, and a relevant impression I_2 is found, deducting ϵ' (shown as a red square in the e_2 filter). In e_3, there is budget, but no relevant impression is found, so no deduction occurs. Finally, in e_4, where the conversion happened but no impression occurred, then through a formalization of publicly available information that we support (§4.5.1), we can justify that no

privacy loss occurs in e_4.

The final attribution report assigns the \$70 value to the single impression I_2 and includes a null value for the second attribution, as Nike requested two. If no impressions were found, or Nike also ran out of budget in e_2, the attribution function would return a report with two null values to avoid leaking information about ad presence.

4.4.3 Algorithm

Listing 4.1 shows how Cookie Monster computes an attribution report.
The `compute_attribution_report` function receives an `attribution_request`, which encapsulates all querier-provided parameters, sanitized by the device. Key parameters include:

1. the window of epochs to search for relevant events (`epochs` parameter);
2. the requested privacy budget (`requested_epsilon`);
3. logic for selecting relevant events (`select_relevant_events`);
4. the attribution policy, such as last-touch or equal-credit (`compute_attribution`);
5. two global sensitivity parameters: `report_global_sensitivity`, the maximum change a device-epoch can make to the output of the report generation function, and `query_global_sensitivity`, the maximum across all devices and reports;
6. p-norm, based on the DP mechanism in MPC/TEE, e.g., 1-norm for Laplace and 2-norm for Gaussian.

All parameters follow a predefined protocol, and while the algorithm is general enough to handle different mechanisms and p-norm sensitivities, our DP result (Thm. 7) focuses on pure DP, assuming the Laplace mechanism and L_1 sensitivity.

Computing an attribution report consists of four steps.

Step 1: Cookie Monster invokes the querier-provided `select_relevant_events` to select relevant events from each separate epoch in the attribution window, such as impressions with a specific campaign ID.

Step 2: For each epoch, Cookie Monster computes the individual privacy loss resulting from the querier's query, following the IDP optimizations in Thm. 10. Three cases:

1. if the epoch has no relevant events, privacy loss is zero;
2. if a single epoch is considered, privacy loss is proportional to the L_p-norm of the attribution function output;
3. if multiple epochs are considered, privacy loss is proportional to the report's global sensitivity.

The privacy loss is scaled by `requested_epsilon` and the query's global sensitivity. In §4.4.2, the report's global sensitivity is 70, and the query's global sensitivity is 100.

Step 3: For each epoch, we attempt to deduct the computed privacy loss from the querier's budget for that epoch, ensuring atomic, thread-safe checks. If the filter has sufficient budget, the epoch's events are used for attribution; otherwise, they are dropped. The justification for dropping contributions is provided in Theorem 7.

Step 4: The attribution function is applied across events from all epochs, following the querier's policy. The device ensures that the attribution computation: (1) respects the querier's specified `report_global_sensitivity` by clipping the attribution histogram to ensure its L_p-norm is $\leq$ `report_global_sensitivity`, and (2) produces encrypted outputs indistinguishable from others. For (2), the device ensures a fixed dimension for the attribution report by padding or dropping elements. For instance, if only one relevant impression is found but two are requested, the output vector is padded with a null entry.

For the example in §4.4.2, this algorithm is invoked with an `attribution_request` where `querier_site` = "nike.com," `epochs` = $[e_1 - e_4]$, `report_global_sensitivity` = 70, `query_global_sensitivity` = 100. Function `select`

`_relevant_events` filters impressions by campaign ID, `pnorm` returns the L1-norm of the attribution histogram, and `compute_attribution` divides the conversion value of 70 across at most two impressions, padding with nulls as needed. This attribution function has sensitivity 70.

```
# Global variables: events_database, privacy_filters.
def compute_attribution_report(attribution_request):
  relevant_events_per_epoch = {}
  for epoch in attribution_request.epochs:
    relevant_events = attribution_request.select_relevant_events(events_database[epoch])  # Step 1
    individual_privacy_loss = compute_individual_privacy_loss(relevant_events, attribution_request)  # Step 2
    filter_status = privacy_filters[attribution_request.querier_site][epoch].check_and_consume(individual_privacy_loss)  #
        Step 3
    if filter_status == "out_of_budget":
      relevant_events = {}
    relevant_events_per_epoch[epoch] = relevant_events
  return attribution_request.compute_attribution(relevant_events_per_epoch)  # Step 4

def compute_individual_privacy_loss(epoch_events, attribution_request):
  if epoch_events == {}:  # Case 1 in Theorem 4
    return 0
  if len(attribution_request.epochs) == 1:  # Case 2 in Theorem 4
    individual_sensitivity = attribution_request.pnorm(attribution_request.compute_attribution(relevant_events))
  else:  # Case 3 in Theorem 4
    individual_sensitivity = attribution_request.report_global_sensitivity
  return attribution_request.requested_epsilon * individual_sensitivity / attribution_request.query_global_sensitivity
```

Code Listing 4.1: **Cookie Monster Algorithm**

4.4.4 Bias Implications of IDP

The execution example and algorithm demonstrate Cookie Monster's budget savings, confirmed in Section 5.6, where we show that these savings allow more accurate queries than ARA and IPA under the same privacy guarantees. However, IDP can introduce bias into query results. Since privacy loss and remaining budgets depend on data, they must remain hidden from advertisers. When a device exhausts its budget for an epoch, it continues participating in queries with "null" data, protecting privacy but potentially introducing bias. For example, Nike's report should have included two impressions, but running out of budget in epoch e_1 meant I_1 wasn't returned, altering the report undetectably.

This bias is a general challenge for all systems operating on IDP, including all existing ad-measurement systems with on-device budgeting – although this challenge is not always acknowl-

edged or handled. Indeed, ARA incorporates code to send null reports when budgets are exhausted and its documentation states that these nulls must be sent to preserve privacy [130]. Such nulls would add bias to query results. In absence of proper IDP formulation, a rudimentary justification we have seen for sending nulls in on-device systems is to prevent revealing budget exhaustion, which could facilitate remote fingerprinting, a concern actively addressed by browsers. Our paper reveals a deeper issue: these systems inherently operate under IDP, and IDP systems must keep budgets hidden, which can lead to bias. Acknowledging this bias opens pathways to mitigate it.

Any (DP or IDP) system must tolerate some error. In ad measurement, high error tolerance is common due to factors like tracking inaccuracies and fraud. The goal is to equip queriers with tools that rigorously bound errors from both DP noise and IDP bias, allowing for informed decision-making. Previous work on centralized-budgeting IDP has developed methods to bound bias using global sensitivity [131] and periodic DP counting queries [132, 131]. These approaches require adaptation to on-device budgeting, given the lack of centralized privacy-loss tracking and non-i.i.d. report sampling. We leave it for future work to develop advanced bias-management tools and here only present a rudimentary approach, which we implement in Cookie Monster and evaluate in §4.7.5 as a proof-of-concept that bias can be effectively managed in on-device budgeting systems.

Our approach adds a *side query* to each attribution query, which bounds potential error from out-of-budget epochs. With each report, the querier requests a boolean flag indicating whether the report could be affected by an out-of-budget epoch. This flag is bundled with the attribution report, secret-shared, and encrypted toward the MPC/TEE. The querier receives a DP-aggregated count of how many reports could be erroneous out of its total batch. With the count, the querier computes a high-probability upper bound on the error from both DP noise and IDP bias. The querier can then filter the results of its queries based on this error bound, ignoring those with unacceptable error.

Consider last-touch attribution. If no epoch in the attribution window is out of budget or an impression is found in a later epoch, the device returns a 0-valued error assessment, indicating no bias. If no impression is found in epochs later than the out-of-budget epoch, the device returns a 1-valued error assessment, signaling potential bias. This information is encrypted and only accessible

to the querier after DP aggregation by the MPC/TEE.

This mechanism lets queriers manage IDP-induced error rigorously, though it consumes additional privacy budget. In Steps 3 and 4 of Listing 4.1, each epoch that is not out of budget must deduct privacy loss for the side query. Fortunately, since the side query is a count query with lower sensitivity than the main query, Cookie Monster's optimizations still provide benefits.

Our evaluation shows that even with bias detection, Cookie Monster consumes less privacy and incurs lower errors compared to ARA and IPA (§4.7.5).

4.5 Formal Modeling and Analysis

This section outlines the theoretical analysis behind Cookie Monster's design, divided into three parts: §4.5.1 introduces a formal model that captures the behavior of on-device budgeting systems, including Cookie Monster but also ARA and PAM. §4.5.2 analyzes this model under IDP, proving that Cookie Monster bounds cross-site leakage and within-site linkability. Finally, §4.5.3 details and justifies the optimizations enabled by IDP, both ones inherently employed in ARA and new ones that our theory uncovers.

4.5.1 Formal System Model

To rigorously analyze privacy properties and identify optimization opportunities in on-device budgeting systems for ad measurement, we must establish a formal model of their behavior. Current ad-measurement systems lack such models, preventing formal analysis or justification of optimizations. Although our model is tailored to Cookie Monster, it can also serve as a foundation for analyzing other systems.

We define the data and queries Cookie Monster operates on, from the perspective of a fixed querier (e.g., advertiser, publisher, or ad-tech).

Data Model

Our data model is based on conversion and impression events collected by user devices and grouped by the time epoch in which they occurred. We view the data available to queriers as a database of such device-epoch groups of events, coming from many devices and defined formally as follows.

Conversion and impression events (F). Consider a domain of impression events $\mathcal{I}$ and a domain of conversion events $\mathcal{C}$. A set of impression and conversion events F is a subset of $\mathcal{I} \cup \mathcal{C}$. The powerset of events is $\mathcal{P}(\mathcal{I} \cup \mathcal{C}) := \{F : F \subset \mathcal{I} \cup \mathcal{C}\}$.

Device-epoch record (x). Consider a set of epochs $\mathcal{E}$ and a set of devices $\mathcal{D}$. We define the domain for device-epoch records $\mathcal{X} := \mathcal{D} \times \mathcal{E} \times \mathcal{P}(\mathcal{I} \cup \mathcal{C})$. That is, a *device-epoch record* $x = (d, e, F)$ contains a device identifier d, an epoch identifier e, and a set of impression and conversion events F.

Database (D). A *database* is a set of device-epoch records, $D \subset \mathcal{X}$, where a device-epoch appears at most once. That is, $\forall d, e \in \mathcal{D} \times \mathcal{E}, |\{F \subset \mathcal{I} \cup \mathcal{C} : (d, e, F) \in D\}| \leq 1$. We denote the set of all possible databases by $\mathbb{D}$. This will be the domain of queries in Cookie Monster. Given a database $D \in \mathbb{D}$ and $x \in \mathcal{X}$, $D + x$ denotes that device-epoch record x is added to database D that initially did not include it.

Device-epoch events data ($\mathbf{D_d^e}$, $\mathbf{D_d^E}$). Given a database $D \in \mathbb{D}$, we define $D_d^e \subset \mathcal{I} \cup \mathcal{C}$ as $D_d^e = F$ if there exist (a unique) F such that $(d, e, F) \in D$, and $D_d^e = \emptyset$ otherwise. Think of this as the event data of device d at epoch e. We also define $D_d^E := (D_d^e)_{e \in E} \in \mathcal{P}(\mathcal{I} \cup \mathcal{C})^{|E|}$ the events of device d over a set of epochs E (typically a contiguous window of epochs).

Public events (P). A key innovation in Cookie Monster's data model is to support incorporation of side information that can be reliably assumed as available to the querier. For example, an advertiser such as Nike can reliably know when someone places a product into a cart (i.e, a conversion occurred), though depending on whether the user is logged in or not, Nike may or may not know who did that conversion.

We model such side information as a domain of *public events* for a querier, denoted $P \subseteq \mathcal{I} \cup C$. P is a subset of all possible events, that will be disclosed to the querier if they occur in the system. We do *not* assume that the querier knows the devices on which events in P occur, and different queriers can have knowledge about different subsets of events. Such side information is typically not modeled explicitly in DP systems, as DP is robust to side information. Cookie Monster also offers such robustness to generic side information. However, we find that additionally modeling the "public" events known to the querier has two key benefits. First, it opens DP optimizations that leverage this known information to consume less privacy budget. Second, it lets us formally define within-site linkability and adapt our design to provide a DP guarantee against such linkability.

Query Model

In on-device systems, queries follow a specific format: first the attribution function runs locally to generate an attribution report, on a set of devices with certain conversions; then, the MPC sums the reports together and returns the result with DP noise. Formally, we define three concepts: attribution function, attribution report, and query.

Attribution function, *a.k.a.* attribution (A). Fix a set of events relevant to the query $F_A \in \mathcal{P}(\mathcal{I} \cup C)$, and $k, m \in \mathbb{N}^*$ where k is a number of epochs. An *attribution function* is a function $A : \mathcal{P}(\mathcal{I} \cup C)^k \to \mathbb{R}^m$ that takes k event sets $F_1, \ldots, F_k$ from k epochs and outputs an m-dimensional vector $A(F_1, \ldots, F_k)$, such that only *relevant events* contribute to A. That is, for all $(F_1, \ldots, F_k) \in \mathcal{P}(\mathcal{I} \cup C)^k$, we have:

$$A(F_1, \ldots, F_k) = A(F_1 \cap F_A, \ldots, F_k \cap F_A).$$

Attribution report, *a.k.a.* report (ρ). This is where the non-standard behavior of on-device budgeting systems, which deduct budget only for devices with specific conversions, becomes apparent. Intuitively, we might consider attribution reports as the "outputs" of an attribution function. However, in the formal privacy analysis, we must account for the fact that only certain devices self-select to run the attribution function (and thus deduct budget). We model this in two steps. First,

we introduce a conceptual *report identifier*, r, a unique random number that the device producing this report generates and shares with the querier at report time.

Second, we define an *attribution report* as a function over the whole database D, that returns the result of an attribution function A for a set of epochs E *only for one specific device* d *as uniquely identified by a report identifier* r. Formally, $\rho_r : D \in \mathbb{D} \mapsto A(D_d^E)$. At query time, the querier selects the report identifiers it wants to include in the query (such as those associated with a type of conversion the querier wants to measure), and devices *self-select* whether to deduct budget based on whether they recognize themselves as the generator of any selected report identifiers. Defining attribution reports on D lets us account for this self-selection in the analysis.

Query (Q). Consider a set of report identifiers $R \subset \mathbb{Z}$, and a set of attribution reports $(\rho_r)_{r\in R}$ each with output in $\mathbb{R}^m$. The *query* for $(\rho_r)_{r\in R}$ is the function $Q : \mathbb{D} \to \mathbb{R}^m$ is defined as $Q(D) := \sum_{r\in R} \rho_r(D)$ for $D \in \mathbb{D}$.

Instantiation in Example Scenario

To make our data and query models concrete, we instantiate the scenarios from §4.3.1.

User Ann's data, together with that of other users, populates dataset D. Each device Ann owns has an identifier d, and events logged from epoch e go into observation $x = (d, e, F)$. $F = I \cup C$ is the set of all events logged on that device during that epoch, including impressions (I) shown to Ann by various publishers, and conversions (C) with various advertisers. Other devices of Ann, other epochs, and other users' device-epochs, constitute other records in the database.

The advertiser, Nike, can observe some of Ann's behavior on its site. As a result, any such behavior logged in C on nike.com constitutes public information for querier Nike. This might include purchases, putting an item in the basket, as well as associated user demographics (e.g. when Ann is logged-in). However, Nike cannot observe impression or conversion events on other websites. As a result, for this querier $P = C_{\text{Nike}}$, which denotes all possible events that can be logged on nike.com. Each actual event in this set (e.g., $F \cap C_{\text{Nike}}$, including Ann's purchase) is associated with an identifier r in Cookie Monster. Using these identifiers, Nike can analyze the

relative effectiveness of two ad campaigns a_1 and a_2 on a given demographics for a product p, such as the shoes Ann bought. First, Nike defines the set of relevant events for the shoe-buying conversion; these are any impressions of a_1 and a_2. Nike uses these relevant events in an attribution function $A : \mathcal{P}(\mathcal{I} \cup \mathcal{C})^{|E|} \to \mathbb{R}^2$ that looks at epochs in E and returns, for example, the count (or value) of impression events corresponding to ads a_1 and a_2. Third, using the set of report identifiers r from purchases of p from users in the target demographic, Nike constructs a query Q that will let it directly compare the proportion of purchases associated with ad campaign a_1 versus campaign a_2.

4.5.2 IDP Formulation and Guarantees

With Cookie Monster's data and query models defined, we now formalize and prove its privacy guarantees using individual DP. After introducing our neighboring relation in §4.5.2, we briefly define traditional DP for reference in §4.5.2, followed by individual DP in §4.5.2. In §4.5.2, we state the IDP guarantees for Cookie Monster, which imply protection against both cross-site tracking and within-site linkability.

Neighboring Databases

A DP guarantee establishes the neighboring database relation, determining the unit of protection. In our case, this unit is the device-epoch record. To account for the existence of public event data (§4.5.1), we constrain neighboring databases to differ by one device-epoch record *while preserving public information*. This ensures that a database containing an arbitrary device-epoch record is indistinguishable from a database containing a device-epoch record with the same public information but no additional data.

Neighboring databases under public information ($D \sim_x^P D'$). Given $D, D' \in \mathbb{D}, x = (e, d, F) \in \mathcal{X}$ and $P \subset \mathcal{I} \cup \mathcal{C}$, we write $D \sim_x^P D'$ if there exists $D_0 \in \mathbb{D}$ such that $\{D, D'\} = \{D_0 + (e, d, F), D_0 + (e, d, F \cap P)\}$. This definition corresponds to a replace-with-default definition [131] combined with Label DP [133].

DP Formulation (for Reference)

In DP, noise must be applied to query results based on the query's *sensitivity*–the worst-case difference between two neighboring databases. Traditional DP mechanisms rely on global sensitivity.

Global sensitivity. Fix a query $q : \mathbb{D} \to \mathbb{R}^m$ for some m (so q could be either a query or an individual report in our formulation). We define the *global* L_1 *sensitivity of* q as follows:

$$\Delta(q) := \max_{D,D' \in \mathbb{D}: \exists x \in \mathcal{X}, D' = D + x} \|q(D) - q(D')\|_1. \tag{4.1}$$

Device-epoch DP. When scaling DP noise to the global sensitivity under our neighboring definition, we can provide device-epoch DP. Fix $\epsilon > 0$ and $P \subset \mathcal{I} \cup \mathcal{C}$. A randomized computation $\mathcal{M} : \mathbb{D} \to \mathbb{R}^m$ satisfies *device-epoch* ϵ*-DP* if for all databases $D, D' \in \mathbb{D}$ such that $D \sim_x^P D'$ for some $x \in \mathcal{X}$, for any set of outputs $S \subseteq \mathbb{R}^m$ we have $\Pr[\mathcal{M}(D) \in S] \leq e^\epsilon \Pr[\mathcal{M}(D') \in S]$. This is the traditional DP definition, instantiated for our neighboring relation.

IDP Formulation

Since queries are aggregated from reports computed on-device with known data, we would prefer to scale the DP noise to the individual sensitivity, which is the worst case change in a query result triggered by the specific data for which we are computing a report.

Individual sensitivity. Fix a function $q : \mathbb{D} \to \mathbb{R}^m$ for some m (so q could be either a query or an individual report in our formulation) and $P \subset \mathcal{I} \cup \mathcal{C}$. Fix $x \in \mathcal{X}$. We define the *individual* L^1 *sensitivity of* q *for* x as follows:

$$\Delta_x(q) := \max_{D,D' \in \mathbb{D}: D' = D + x} \|q(D) - q(D')\|_1. \tag{4.2}$$

While we cannot directly scale the noise to individual sensitivity, we can scale the on-device budget consumption using this notion of sensitivity. That is, for a fixed and known amount of noise

that will be added to the query, a lower individual sensitivity means that less budget is consumed from a device-epoch. This approach provides a guarantee of individual [2] DP [126, 131] for a device-epoch, defined as follows.

Individual device-epoch DP. Fix $\epsilon > 0$, $P \subset \mathcal{I} \cup \mathcal{C}$, and $x \in \mathcal{X}$. A randomized computation $\mathcal{M} : \mathbb{D} \to \mathbb{R}^m$ satisfies *individual device-epoch ϵ-DP for x* if for all databases $D, D' \in \mathbb{D}$ such that $D \sim_x^P D'$, for any set of outputs $S \subseteq \mathbb{R}^m$ we have $\Pr[\mathcal{M}(D) \in S] \leq e^\epsilon \Pr[\mathcal{M}(D') \in S]$.

Intuitively, IDP ensures that, from the point of view of a fixed device-epoch x, the associated data F is as hard to recover from query results as it would be under DP.

IDP Guarantees

Through IDP, we prove two main properties of Cookie Monster: (1) **Individual DP guarantee**, which implies bounds on *cross-site leakage*, demonstrating that the API cannot be used to reveal cross-site activity; and (2) **Unlinkability guarantee**, which implies bounds on *within-site linkability*, demonstrating that the API cannot be used even by a first-party site to distinguish whether a set of events is all on one device vs. spread across two devices.

For the IDP guarantee, we give two versions. First, a stronger version under a mild constraint on the class of allowed queries, specifically that $\forall i, \forall F,\ A(F_1, ..., F_{i-1}, F_i \cap P, F_{i+1}, ..., F_k) = A(F_1, ..., F_{i-1}, \emptyset, F_i, ..., F_k)$. A sufficient condition is to ensure that queries leverage public events only through their report identifier, i.e. $F_A \cap P = \emptyset$. The queries from the scenarios we consider (§4.3.1) satisfy this property. Second, a slightly weaker version of the DP guarantee with increased privacy loss, but with no constraints on the query class, which is useful when considering colluding queriers.

Theorem 7 (Individual DP guarantee). *Fix a set of public events $P \subset \mathcal{I} \cup \mathcal{C}$, and budget capacities $(\epsilon_d^G)_{d \in \mathcal{D}}$.* **Case 1:** *If all the queries use attribution functions A satisfying $\forall i, \forall F,\ A(F_1, ..., F_{i-1}, F_i \cap P, F_{i+1}, ..., F_k) = A(F_1, ..., F_{i-1}, \emptyset, F_i,$*

[2] While referred to as Personalized Differential Privacy (PDP) in some papers [126], we use the term Individual Differential Privacy (IDP), as it better reflects the concept and aligns with individual sensitivity, the basis of the definition. This recent paper [131] also uses IDP terminology.

..., F_k), then for $x \in \mathcal{X}$ on device d, Cookie Monster satisfies individual device-epoch ϵ_d^G-DP for x under public information P. **Case 2:** *For general attribution functions, Cookie Monster satisfies individual device-epoch $2\epsilon_d^G$-DP for x under public information P.*

Intuitively, the information gained on cross-site (private to the querier) events in device-epoch x under the querier's queries is bounded by ϵ_x^G (or $2\epsilon_x^G$ without query constraints).

Theorem 8 (**Unlinkability guarantee).** *Fix budget capacities $(\epsilon_d^G)_{d\in\mathcal{D}}$. Take any $d_0, d_1 \in \mathcal{D}$, $e \in \mathcal{E}$, and $F_1 \subset F_0$. Denote $x_0 := (d_0, e, F_0), x_1 := (d_1, e, F_1), x_2 := (d_0, e, F_0 \setminus F_1) \in \mathcal{X}$. For any $D, D' \in \mathbb{D}$ such that $\{D, D'\} = \{D_0 + x_0, D_0 + x_1 + x_2\}$ for some $D_0 \in \mathbb{D}$, instantiation $\mathcal{M}$ of Cookie Monster, and $S \subset Range(\mathcal{M})$ we have:* $\Pr[\mathcal{M}(D) \in S] \leq e^{2\epsilon_{d_0}^G + \epsilon_{d_1}^G} \Pr[\mathcal{M}(D') \in S]$.

Intuitively, linking a set of events across two devices—compared to detecting these events on one device—is only made easier by the amount of budget on the second device; Cookie Monster does not introduce additional privacy loss for linkability, above what is revealed through DP queries.

4.5.3 IDP Optimizations

IDP allows discounting the DP budget based on individual sensitivity, which is never greater but often smaller than global sensitivity. The easiest way to grasp this opportunity is to visualize and compare the definitions of global and individual sensitivities for reports and queries. Recall that Cookie Monster enforces a bound on reports by capping each coordinate in the attribution function's output to a querier-provided maximum. Given this cap, we prove the following formulas for both sensitivities:

Theorem 9 (**Global sensitivity of reports and queries).** *Fix a report identifier r, a device d_r, a set of epochs E_r, an attribution function A and the corresponding report $\rho : D \mapsto A(D_{d_r}^{E_r})$. We have:*

$$\Delta(\rho) = \max_{i\in[k], F_1,\ldots,F_k\in\mathcal{P}(\mathcal{I}\cup C)} \|A(F_1, ..., F_k) - A(F_1, ..., F_{i-1}, \emptyset, F_{i+1}, ..., F_k)\|_1$$

Next, fix a query Q with reports $(\rho_r)_{r\in R}$ such that each device-epoch participates in at most one report. We have $\Delta(Q) = \max_{r\in R} \Delta(\rho_r)$.

Theorem 10 (Individual sensitivity of reports and queries). *Fix a device-epoch record $x = (d, e, F) \in \mathcal{X}$. Fix a report identifier r, a device d_r, a set of epochs $E_r = \{e_1, \ldots, e_k\}$, an attribution function A with relevant events F_A, and the corresponding report $\rho : D \mapsto A(D_{d_r}^{E_r})$.*

We have: $\Delta_x(\rho) = \max_{F_1,\ldots,F_{i-1},F_{i+1},\ldots,F_k \in \mathcal{P}(\mathcal{I}\cup\mathcal{C})} \|A(F_1, ..., F_{i-1}, F, F_{i+1}, ..., F_k) - A(F_1, ..., F_{i-1}, \emptyset, F_{i+1}, ..., F_k)\|_1$ if $d = d_r$ and $e = e_i \in E_r$, and $\Delta_x(\rho) = 0$ otherwise.

In particular,

$$\Delta_x(\rho) \leq \begin{cases} 0 & \text{if } d \neq d_r, e \notin E_r \text{ or } F \cap F_A = \emptyset \\ \|A(F) - A(\emptyset)\|_1 & \text{if } d = d_r \text{ and } E_r = \{e\} \\ \Delta(\rho) & \text{if } d = d_r, e \in E_r \text{ and } F \cap F_A \neq \emptyset \end{cases}$$

Next, fix a query Q with reports $(\rho_r)_{r\in R}$. Then we have: $\Delta_x(Q) \leq \sum_{r\in R} \Delta_x(\rho_r)$. In particular, if x participates in at most one report ρ_r, then: $\Delta_x(Q) = \Delta_x(\rho_r)$.

This theorem justifies both the inherent optimization used by all on-device systems and the new optimizations added in Cookie Monster.

Inherent on-device optimization. The condition $d = d_r$ in Thm. 10 explains why, under IDP, on-device budgeting systems deduct privacy loss only for devices that participate in a query. This is more efficient than off-device systems like IPA, which, under traditional DP, must deduct budget based on $\Delta(Q)$ from *all devices*, regardless of their participation (Thm. 9).

New optimization examples. First, devices that participate in a query but have no relevant data (i.e. $F \cap F_A = \emptyset$ or $A(F) = A(\emptyset)$ in Thm. 10) do not incur budget loss. This is why, in the example from § 4.4.2, we don't deduct from epoch e_3, which has no Nike impressions. Second, a device's individual sensitivity depends only on reports it participates in ($\Delta_x(Q) = \Delta_x(\rho_r)$), whereas global sensitivity depends on all reports in the query ($\Delta(Q) = \max_{r\in R} \Delta(\rho_r)$). For instance, since

the report ρ typically depends on the public information $F \cap P$ of a record (d, e, F), we use a \$70 cap instead of \$100 in the Nike example. Third, if an attribution spans only one epoch (or is broken into single-epoch reports), individual sensitivity can be further reduced based on the private information F. For example, if Nike measures the average impression-to-conversion delay (0 to 7 days) in a single epoch and a record x has one impression only 1 day before the conversion, its individual budget will be 1/7th of the global budget.

4.6 Chrome Prototype

We integrated Cookie Monster into Google Chrome by modifying ARA. We disabled ARA's impression-level budgeting, added epoch support, and extended ARA's database to include a table for privacy filters for each epoch-querier pair. Unlike ARA, which supports only last-touch attribution and fetches only the latest impression, our implementation retrieves all impressions related to the conversion, groups them by epoch, and identifies epochs with no relevant data to avoid unnecessary budget consumption.

4.7 Evaluation

We seek to answer three key questions:

Q1: How do optimizations impact budget consumption?

Q2: How do optimizations impact query accuracy?

Q3: How effective is bias measurement?

4.7.1 Methodology

We evaluate Cookie Monster on three datasets—a microbenchmark and two realistic advertising datasets from PATCG and Criteo—and compare its privacy budget consumption and query accuracy against two baselines. The first baseline is **IPA-like**, our own prototype implementing IPA's centralized budgeting and query execution. The second is **ARA-like**, a version of ARA

providing device-epoch-level guarantees. ARA-like includes the inherent optimization of all on-device systems but excludes the new optimizations in §4.5.3.

Scenario-driven methodology. We conduct our evaluation by enacting the scenario from §4.3.1. An advertiser (Nike) runs ad campaigns and repeatedly measures their efficacy. Each time a customer purchases quantity C of a product, Nike requests an attribution report, specifying the relevant ad campaigns. Nike requests reports over some attribution window and uses last-touch attribution. If no relevant impression is found, the report value is 0; otherwise, it is C. Nike batches reports and submits them to the aggregation service for a DP summation query using the Laplace mechanism. In our experiments, Nike repeatedly performs queries on report batches of size B, which varies by dataset. Once B reports are gathered, Nike runs its query. This is repeated over time as more batches of B reports are gathered. This is also repeated for each product, e.g., 10 in the microbenchmark/PATCG and a variable number in Criteo.

When requesting an attribution report for a conversion, Nike must specify the requested privacy budget, ϵ – the same value for all reports in a batch. Since the MPC uses the Laplace mechanism to ensure ϵ-DP, Nike selects ϵ to achieve acceptable accuracy. We assume Nike chooses ϵ in an attempt to keep query error within 5% ($\alpha = 0.05$) of the true value with 99% probability ($\beta = 0.01$), which corresponds to roughly 0.02 RMSRE. The formula for ϵ is: $\epsilon = \Delta \ln(1/\beta)/(\alpha \cdot B \cdot \tilde{c})$, where Δ is the maximum value for C and $\tilde{c}$ is Nike's rough estimate of the average C.

Our specific method is: we run repeated, single-advertiser summation queries on fixed-size batches of attribution reports, using last-touch attribution and a privacy budget calibrated as described above. Default parameters include: a 7-day epoch size, a 30-day attribution window, and a global privacy budget per epoch of $\epsilon_G = 1$.

Microbenchmark dataset. To methodically evaluate Cookie Monster, under a range of conditions, more or less favorable to our optimizations, we create a synthetic dataset with 40,000 conversions across 10 products over 120 days. We expose two knobs: **Knob1**, the user participation rate per query, determines the fraction of users who are assigned conversions relevant for a particular query; **Knob2**, the number of impressions per user per day. These knobs impact budget

allocation across IPA-like, ARA-like, and Cookie Monster. Lower Knob1 increases opportunities for fine-grained accounting in ARA-like and Cookie Monster. Lower Knob2 allows Cookie Monster to conserve privacy by not deducting from epochs with no relevant impressions, a key optimization over ARA-like.

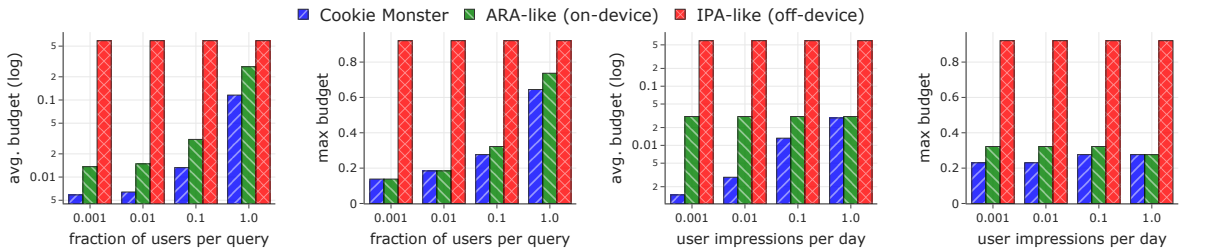
PATCG dataset. To evaluate Cookie Monster under more realistic conditions, we resort to the PATCG and Criteo datasets. PATCG is a synthetic dataset released by the namesake W3C community group [134], which contains 24M conversions from a single advertiser over 30 days. This dataset represents a large advertiser, with only 1% of conversions attributed to impressions. There are 16M distinct users, and each user sees an average of 3.2 impressions. Users who convert take part in 1.5 conversions on average.

Criteo dataset. The Criteo dataset [135] is sampled from a 90-day log of live ad impressions and conversions recorded by the Criteo ad-tech. The dataset includes data from 292 advertisers with 12M impression records and 1.3M conversion records. There are 10M unique users. The dataset provides opportunities for evaluating Cookie Monster in some additional dimensions compared to PATCG and the microbenchmark. In particular, the Criteo dataset contains data from multiple advertisers of widely distinct sizes, i.e., having a wide range in terms of number of impressions (1–2.6M impressions) and conversions per advertiser (0–478k conversions). However, since the dataset is heavily subsampled, missing many impressions, we also evaluate Cookie Monster on augmented versions of this dataset, in which we add synthetic impressions to compensate for the missing impressions that might otherwise favor Cookie Monster's optimizations.

4.7.2 Microbenchmark Evaluation (Q1)

We use the microbenchmark to evaluate the impact of individual-sensitivity optimizations on privacy budget consumption across a range of controlled workloads (question Q1).

Varying user participation rate per query (knob1). We first vary the user participation rate per query. With a default batch size of 2,000 reports and 10 products (queried twice, totaling 20 queries), we create 40,000 conversions. Knob1 controls how these conversions are assigned to



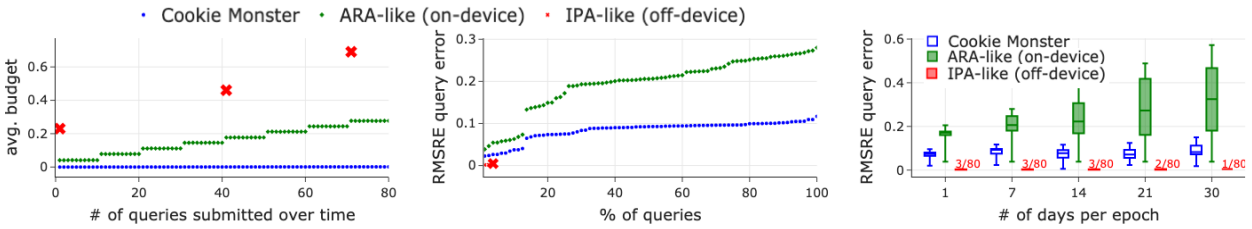
(a) Avg. budget varying knob1 (b) Max. budget varying knob1 (c) Avg. budget varying knob2 (d) Max. budget varying knob2

Fig. 4.4: **Budget consumption on the microbenchmark.** (a) and (b) show average and maximum budget consumption across all device-epochs, respectively, as a function of the fraction of users that participate per query (knob1); value of knob2 is constant 0.1. (c) and (d) show the same metrics as a function of user impressions per day (knob2); value of knob1 is constant 0.1.

users, indirectly determining the total number of users. A lower knob1 favors on-device budgeting, as it spreads the 40,000 conversions across more users, creating more privacy filters for the advertiser. For example, with knob1 = 1, each user participates in all 20 query batches, requiring a minimum of 2,000 users, while knob1 = 0.001 generates 2M users. In the PATCG dataset, users convert with a 0.05 daily rate, corresponding to knob1 = 0.1, which we use as default in other experiments.

Fig. 4.4a and 4.4b show the average and maximum budget consumption across all device-epochs requested through the 20 queries. Qualitatively, the average budget consumption is a much more useful metric to assess the efficiency of the three systems, but we include the maximum because it reduces IDP guarantees to standard DP guarantees, thereby providing a more apples-to-apples comparison between on-device and off-device budgeting. Recall that IPA-like does not distribute budget consumption across devices but has a centralized privacy filter for each epoch, from which it deducts budget upon executing each query. As a result, increasing user participation per query (knob1) does not impact its budget consumption, which is always higher than the other methods'. Cookie Monster consistently consumes the least budget due to its optimizations, with greater improvements as user participation increases (lower knob1), since more device-epochs lack relevant impressions and don't deduct budget. Even under the max budget metric, on-device systems outperform IPA-like, with Cookie Monster being the most efficient.

Varying the number of impressions per user per day (knob2). We now fix knob1 at 0.1 and



(a) Avg. budget consumed across all device-epochs

(b) CDF of RMSRE

(c) RMSRE as a function of epoch length

Fig. 4.5: **Budget consumption and query accuracy on the PATCG dataset.** (a) Average budget consumption across all device-epochs as a function of the number of queries submitted by the advertiser. (b) CDF of RMSRE with a 7-day epoch. (c) RMSRE median (horizontal lines), first and third quartiles (boxes), and max/min (top/bottom range markers) as epoch length increases.

vary the number of impressions per user per day (knob2). In PATCG, users see an average of 3.22 ads over 30 days, giving knob2 a value of 0.1. Fig. 4.4c and 4.4d confirm that Cookie Monster's optimizations are most effective when users have fewer impressions.

Thus, Cookie Monster reduces budget consumption compared to baselines, especially when budget is spread across many users and when users have fewer impressions.

4.7.3 PATCG Evaluation (Q1, Q2)

We use the PATCG dataset to evaluate Cookie Monster's impact on budget consumption (Q1) and query accuracy (Q2). This dataset links impressions and conversions to attributes, with values uniformly sampled from 0 to 9, representing 10 potential products. Nike queries each product eight times over the four months spanning the dataset, totaling 80 queries with batch sizes between 280,000 and 303,009 reports. Large batch sizes accommodate the low attribution rate (1% of impressions relevant to conversions), assuming Nike adjusts batch sizes accordingly.

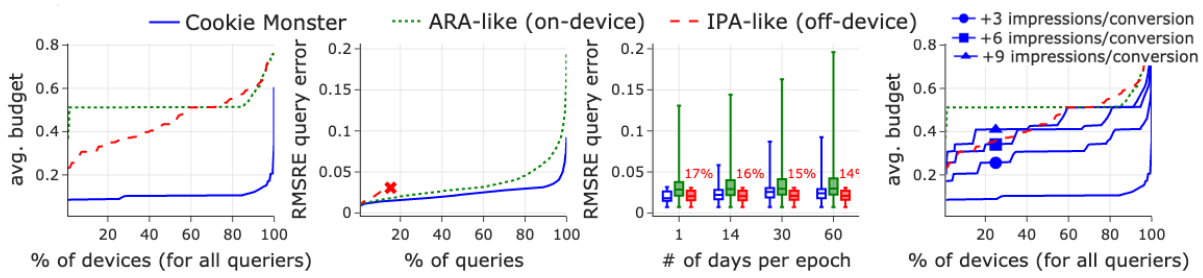
Fig. 4.5a illustrates the average privacy budget consumed by each system as 80 queries are submitted for execution by the advertiser. The x-axis represents the order of queries, with points indicating budget consumption. IPA-like executes only a small fraction of queries (3.75%) due to its coarse-grained, population-level accounting, leading to early budget depletion. ARA-like and Cookie Monster, with finer-grained, individual-level accounting, execute all queries and resulting in smoother and lower average budget consumption. Cookie Monster shows up to 206 times lower

average budget consumption compared to ARA-like, highlighting the benefits of its individual-sensitivity optimizations.

Next, we assess query accuracy (Q2). On-device systems (ARA-like and Cookie Monster) hide budgets when depleted, which can affect query accuracy, while IPA-like explicitly rejects queries with exhausted budgets. As in our experiments, privacy budgets are set to aim for high accuracy in the Laplace mechanism, we expect IPA's executed queries to have errors within the 0.02 mark. In contrast, ARA and Cookie Monster may incur additional errors when epochs run out of budget, leading to nullified or incomplete reports.

Fig. 4.5b shows the CDF of root mean square relative error (RMSRE), defined as $\sqrt{\mathbb{E}[(\mathcal{M}(D) - Q(D))^2/Q(D)^2]}$ for an estimate $\mathcal{M}(D)$ of the query output $Q(D)$. This metric captures both Laplace-induced and IDP-bias-induced errors. The CDF shows query errors for each system. IPA-like's line ends at 3.75% of queries, aligning with its budget constraints but maintaining within the 5% error mark. Cookie Monster consistently exhibits lower errors than ARA-like due to its budget conservation, resulting in fewer nullified reports and reduced bias. This is true without any bias mitigation strategies. In §4.7.5, we show that even with bias measurement running alongside every query, Cookie Monster still outperforms ARA-like (which has no bias measurement) in terms of budget consumption and query accuracy.

Finally, we explore how epoch length affects performance. Longer epochs strengthen device-epoch privacy guarantees but slow budget refreshing, leading to more query rejections in IPA and increased bias in on-device systems without mitigation. Fig. 4.5c evaluates RMSRE measures (median, first and third quartiles, and range) as epoch length varies. IPA-like's query execution drops to 1.25% at one-month epochs, while Cookie Monster and ARA-like complete all queries but with increasing errors. Cookie Monster's budget conservation results in fewer altered or nullified reports, maintaining lower error degradation compared to ARA-like as epochs grow.



(a) CDF of budget on Criteo.

(b) CDF of RMSRE.

(c) RMSRE as function of epoch length.

(d) CDF of budget on Criteo++.

Fig. 4.6: **Budget consumption and query accuracy on Criteo.** (a) CDF of per-device average budget consumption across epochs for all devices and advertisers. (b) CDF of RMSREs for a 7-day epoch. (c) RMSRE metrics with varying epoch length (see Fig. 4.5c for format). (d) The same CDF as in (a), but for Criteo++, showing the impact of synthetic impression augmentation on Cookie Monster's performance.

4.7.4 Criteo Evaluation (Q1, Q2)

The Criteo dataset enables evaluation across diverse advertisers. It includes 1.3M conversions from 292 advertisers, with conversions ranging from 0 to 478k per advertiser. To achieve meaningful accuracy under DP, an advertiser needs a minimum number of reports. We set this minimum to 350, allowing us to formulate at least one query for 109 advertisers. Advertisers with more than 350 conversions wait to accumulate 350 reports per batch for each query, resulting in 898 queries across these advertisers using the attribute "product-category-3" as a product ID.

Fig. 4.6a shows a CDF of per-device average budget consumption across epochs, where the distribution covers all devices and all advertisers; that is, there is a single data point corresponding to each device and advertiser pair, which indicates the average consumption across epochs within an advertiser's filters on a given device by the end of the workload. Lower values indicate better performance. Cookie Monster conserves the most privacy budget, with 95% of device-advertiser pairs having more capacity left compared to both baselines.

Fig. 4.6b presents the CDF of RMSREs for all 898 queries. IPA-like completes only a small fraction of queries but with good accuracy. ARA-like and Cookie Monster accept all queries, potentially at the expense of higher error; however, Cookie Monster's error distribution remains better than ARA-like's, with errors within IPA-like's range for up to 96% of queries. This results

(a) Avg. budget consumed (b) CDF of RMSRE (c) RMSRE as function of error estimation cutoff.

Fig. 4.7: **Budget consumption and query accuracy with bias measurement on the microbenchmark.** (a) Average budget consumed across all device-epochs. (b) CDF of true RMSRE for executed queries, alongside Cookie Monster's RMSRE estimation from bias measurement (light-purple line). (c) Quartiles of true RMSRE, where queries with error estimate above a given cutoff are rejected by Cookie Monster with bias measurement.

from Cookie Monster's optimizations that conserve budget and avoid introducing bias.

Fig. 4.6c examines how RMSRE varies with epoch length. Longer epochs increase contention on per-epoch filters. Despite this, Cookie Monster's optimizations show substantial benefits, with minimal RMSRE increase (25% increase from 1-day to 60-day epoch for median RMSRE). Although maximum RMSRE increases with epoch length, Cookie Monster's performance remains superior to ARA-like.

Recall that the Criteo dataset is heavily subsampled, so there is the possibility that missing impressions may amplify the benefit of our optimizations. To assess Cookie Monster's performance in scenarios with more relevant impressions, we augment the Criteo dataset with synthetic impressions for each conversion. The results, shown in Fig. 4.6d, compare the CDFs of budget consumption with varying augmentation levels. The behavior of IPA-like and ARA-like remains unchanged by augmentation, as they do not optimize for missing relevant impressions. For Cookie Monster, budget efficiency decreases as more synthetic impressions are added, approaching ARA-like's performance at 9 extra impressions per conversion. The impressions are uniformly distributed across the attribution window, ensuring that most epochs have relevant impressions for most conversions, so Cookie Monster's optimization is eliminated and its behavior follows ARA-like's.

4.7.5 Bias Measurement (Q3)

We evaluate Cookie Monster's bias measurement technique using our microbenchmark with default knob settings (0.1) and an increased query load to measure significant bias. Specifically, we use 60 days and repeat each query 40 times.

Fig. 4.7a shows the budget overhead incurred by bias measurement. The bias measurement's counts are scaled to have 10% the sensitivity of the original query, so the overall sensitivity of the query/side-query combination increases by 10%. The average consumed budget goes from 0.36 without bias measurement to 0.43 with bias measurement; this is more than a 10% increase since some epochs that originally paid zero budget through our IDP optimization, now pay for bias counts.

Fig. 4.7b shows the CDF of RMSREs across all 400 queries, with a log scale on the y-axis to highlight smaller differences among Cookie Monster variants compared to ARA. Due to the heavy query load, IPA executes only 5% of the queries and ARA ultimately returns empty reports, resulting in a relative error of 1. Cookie Monster without bias measurement plateaus at 0.2 error. Cookie Monster with bias measurement shows a similar trend to Cookie Monster without it, albeit with increased error, because the higher sensitivity of the query leads additional epochs to run out of budget. However, the bias measurements let queriers compute an estimate of the error, which, although noisy (as it is also differentially private), generally serves as an upper bound on true RMSRE. Queriers can compare this estimate to a predetermined cutoff and reject queries exceeding it. Fig. 4.7c displays the quartiles of true RMSREs after rejecting queries based on estimated RMSRE cutoffs. For instance, using a cutoff of 0.05 enables queriers to limit bias, achieving a maximum error of 0.04 (down from 0.21), but only accepting 30% of the queries. Rejected queries still consume budget, as rejection is a post-processing step.

Thus, even with rudimentary bias measurement, Cookie Monster offers substantial benefits over IPA while maintaining lower real error than ARA. While we validated our technique on a microbenchmark with increased query load, applying it to real-life datasets remains an open challenge. Future work could enhance our technique by scheduling bias measurements or using DP

threshold comparison mechanisms.

4.8 Related Work

DP systems. Most DP systems operate in the centralized-DP model, where a trusted curator runs queries using global sensitivity [136]. Some implement fine-grained accounting through parallel composition [137, 9, 8, 10], a coarse form of individual DP (IDP) that lacks optimizations like those in Cookie Monster. Others function in the local-DP model, where devices randomize their data locally [138], and therefore inherently do on-device budgeting but have higher utility costs. Distributed systems like [98, 139] emulate the central model with cryptographic constructions; like IPA, they maintain a single privacy filter, not leveraging IDP to conserve budget. [140] uses the shuffle model [141] to combine local randomization with a minimal trusted party. Cookie Monster operates in the central model with on-device budgeting and uses an IDP formalization to enable new optimizations.

Private ads measurement. Several proposals exist for private ad measurement systems. Apple's PCM [142] relies on entropy limits for privacy. Meta and Mozilla's IPA [143] uses centralized budgeting, while Google's ARA [122] and Apple's PAM [124] utilize on-device budgeting. ARA has primarily focused on optimizing in-query budget and utility. [144] optimizes a single vector-valued hierarchical query, whereas [145] assumes a simplified ARA with off-device impression-level DP guarantees, efficiently bounding each impression's contribution for queries known upfront. [146] offers a framework for attribution logic and DP neighborhood relations, proposing clipping strategies for bounding global sensitivity. Our work optimizes on-device budgeting across queries, using tighter individual sensitivity bounds. Our method is agnostic to how these bounds are enforced, potentially benefiting from clipping algorithms [144, 145, 146].

IDP was introduced in the centralized-DP setting, where a trusted curator manages individual budgets and leverages individual sensitivity to optimize privacy accounting [126, 131]. IDP is used for SQL-like queries and gradient descent. The literature emphasizes the need to keep individual budgets private. [132] studies the release of DP aggregates over these budgets while [126] notes

that out-of-budget records must be dropped silently, leaving bias analysis for future work.

4.9 Conclusion

Web advertising is at a crossroads, with a unique opportunity to enhance online privacy through new, privacy-preserving APIs from major browser vendors. We show that a novel individual DP formulation can significantly improve privacy budgeting in on-device systems. However, further progress is needed in query support, error management, and scalability. Our paper provides foundational insights and formal analysis to guide future research and industry collaboration.

Chapter 5: Dances with Locks: An Adaptive Commit Protocol for Distributed Transactions

5.1 Overview

Strict Two-Phase Locking (2PL) combined with Two-Phase Commit (2PC) remains the standard approach for ensuring strict serializability and atomicity in distributed transactions. However, its conservative strategy of holding locks throughout the full duration of the commit process amplifies the contention footprint of transactions, limiting throughput and latency under high-contention workloads. Relaxed variants, such as Early Lock Release (ELR), improve concurrency through pipelining: by releasing locks earlier in the commit protocol, subsequent transactions can acquire locks and proceed before prior commits complete. However, this introduces commit-time dependencies that require additional coordination and risk cascading aborts. No single protocol excels across all workload conditions and resource availability, yet most distributed systems hardcode one fixed strategy — designed for specific assumptions while sacrificing generality.

This paper introduces Sangria, a novel distributed commit protocol that dynamically adapts its commit strategy to exploit the complementary strengths of both conservative and relaxed approaches. Unlike traditional designs that enforce a single, fixed commit strategy across the entire system, Sangria provides fine-grained adaptability: each participant involved in the transaction — typically corresponding to an accessed data item — independently adjusts its commit behavior based on its local contention and general resource availability. By intelligently balancing between conservative and relaxed commit modes at runtime, Sangria dynamically optimizes performance under varying conditions while preserving the strong consistency guarantees inherent to each mode. This work opens the door to a more flexible, workload-aware approach to distributed transaction management — where transactions no longer commit rigidly, but instead dance to the

rhythm of the workload.

5.2 Introduction

Distributed transactions provide a powerful abstraction for simplifying application logic in distributed systems that operate on shared mutable state. By adhering to the ACID properties — Atomicity, Consistency, Isolation, and Durability — distributed transaction protocols allow applications to reason about distributed operations as if they execute atomically and in isolation, even in the presence of failures and concurrency. Strict serializability [147] is widely regarded as the strongest isolation and consistency guarantee, ensuring that the outcome of concurrent transactions is equivalent to some serial execution that respects real-time order.

Enforcing strict serializability in distributed settings often relies on combining Strict Two-Phase Locking [148] (2PL) for concurrency control with Two-Phase Commit [149] (2PC) for atomic commitment across participants — a combination we hereafter refer to as Strict-2PC. 2PL enforces serializability by holding locks across all accessed data items until the transaction's commit decision is finalized, while 2PC ensures that all participants either commit or abort atomically, preserving atomicity. While Strict-2PC provides strong correctness guarantees, its conservative nature forces transactions to hold locks across the entire commit process, including under network delays and coordination rounds, resulting in increased contention footprints and degraded performance under high-contention workloads. While we focus on 2PL in this paper, optimistic concurrency control (OCC) schemes are also not immune, and often perform even worse under high contention [150, 151, 152].

To address these limitations, various relaxations of Strict-2PC have been explored, aiming to reduce lock contention by decoupling the time spent holding locks from commit coordination. One such approach [153, 154, 155] is to apply Early Lock Release [156, 157] (ELR) to 2PC, which we hereafter refer to as Pipelined-2PC. In Pipelined-2PC, locks are released early in the commit protocol, allowing subsequent transactions to acquire locks earlier and pipeline their execution. This approach takes advantage of the fact that after a transaction T finishes the execution phase, it will

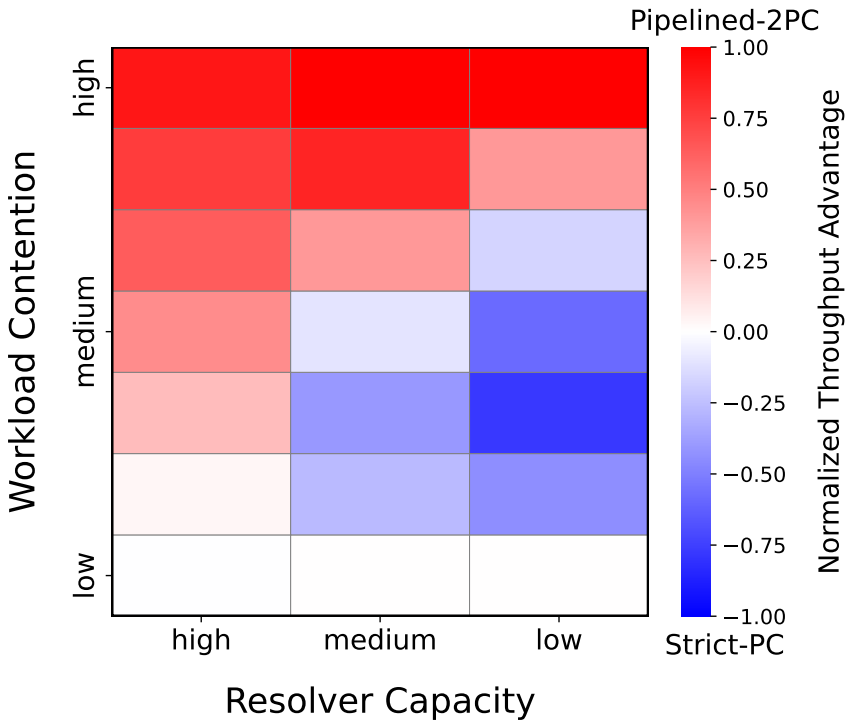


Fig. 5.1: Heatmap illustrating the regimes where each protocol is most effective as a function of workload contention (vertical axis) and Resolver capacity (horizontal axis). Red regions indicate scenarios where pipelining (Pipelined-2PC) outperforms Strict-2PC, while blue regions indicate the opposite. The color intensity reflects the magnitude of the performance advantage.

not acquire more locks, so holding locks after this point has no value from a 2PL perspective. Furthermore, at this point, T is guaranteed to not abort due to concurrency control or application logic, so the only reason it aborts is due to server failure, which should be rare. By releasing its locks at the start of 2PC, T avoids blocking other transactions that access T's writeset while T is running its high-latency commit process. While this approach improves concurrency, it introduces commit-time dependencies between transactions: if a transaction releases its locks early but subsequently aborts, any dependent transactions that acquired conflicting locks before the commit decision will also need to be aborted — a phenomenon known as cascading aborts. These dependencies introduce two fundamental challenges: (1) pipeline stalls, where commit dependencies must be resolved before transactions can safely commit, limiting the benefits of pipelining; and (2) the need for distributed dependency tracking, which requires additional coordination and bookkeeping across participants.

In this paper, we present a novel implementation of Pipelined-2PC that addresses these challenges by introducing a centralized coordination entity called the Resolver. The Resolver maintains a global dependency graph of in-flight transactions and orchestrates commit resolution on behalf of participants. This design simplifies dependency management by offloading tracking to a centralized entity, and enables an important performance optimization: batching commits. As dependent transactions queue up waiting for their predecessors to commit, the Resolver groups multiple ready-to-commit transactions into batches, reducing commit-time overhead by amortizing the cost of durable storage writes across multiple transactions. In effect, this enables a distributed form of group commit [158].

While Pipelined-2PC with the Resolver significantly improves concurrency under many workloads, the Resolver has limited resources so there are scenarios where it can become a bottleneck itself. Always involving the Resolver on every transaction would therefore impose an overall scalability limit on the system, which would not be ideal for low contention workloads. A big challenge is that the level of contention within the same database varies over time and over different records, and is often unpredictable [159, 160, 155]. Figure 5.1 sketches the interaction between two key dimensions that influence commit protocol performance: (i) the level of workload contention, and (ii) the capacity of the centralized Resolver responsible for dependency tracking and commit coordination in Pipelined-2PC. This heatmap visualizes the regimes where each protocol is most effective and is derived from a set of experiments presented in the evaluation section, which systematically explore performance across varying contention levels and Resolver capacities. Red regions indicate scenarios where pipelining is more beneficial, while blue regions highlight where Strict-2PC performs better. The intensity of each color reflects the magnitude of the performance advantage for each protocol. We define it as the normalized throughput advantage i.e. the normalized difference between the throughput of Pipelined-2PC and that of Strict-2PC under the same workload configuration — a value closer to 1 indicates a performance gain relative to Strict-2PC, while a value closer to -1 indicates a loss.

As contention increases, pipelining becomes increasingly advantageous. By enabling early

lock release and overlapping prepare phases across dependent transactions, it improves concurrency and reduces idle time. However, these benefits hinge on the Resolver's capacity to efficiently manage dependency tracking. When the Resolver is heavily loaded — due to background traffic, system resource contention, or long dependency chains (low capacity) — its coordination overhead can offset pipelining's gains.

This observation motivates our main contribution: Sangria, an adaptive commit protocol that dynamically switches between Strict-2PC and Pipelined-2PC based on the workload. In our design, each participant independently chooses whether to release locks early (enabling pipelining) or hold locks conservatively, based on its own observed workload contention and the Resolver's capacity. The Resolver remains responsible for dependency tracking and group commits, but its involvement is reduced when participants opportunistically bypass pipelining under favorable conditions. This design allows the system to automatically balance between aggressive pipelining and conservative commit behavior, achieving consistently high performance across diverse workload patterns even under heavy contention or centralized Resolver pressure.

Enabling this adaptive behavior required extending the 2PC protocol itself. We introduce lightweight coordination enhancements where the coordinator piggybacks information about the current Resolver load and contention onto the Prepare requests. Participants use this information to decide whether to release locks early or continue to hold them. In turn, participants piggyback both their locking decisions and any local dependency information onto the corresponding Prepare responses, informing the coordinator of their chosen commit behavior and dependency state. This information allows the coordinator to make transaction-specific decisions on whether to delegate commit processing to the Resolver, notify it asynchronously of the commit decision, or bypass it altogether. These protocol-level extensions support different policies to guide adaptation decisions, allowing the system to flexibly respond to changing workload and resource dynamics while preserving the correctness guarantees of the underlying protocols.

In summary, this paper makes the following contributions:

- We present a novel Pipelined-2PC protocol with a centralized Resolver that addresses depen-

dency tracking challenges and enables efficient commit batching.

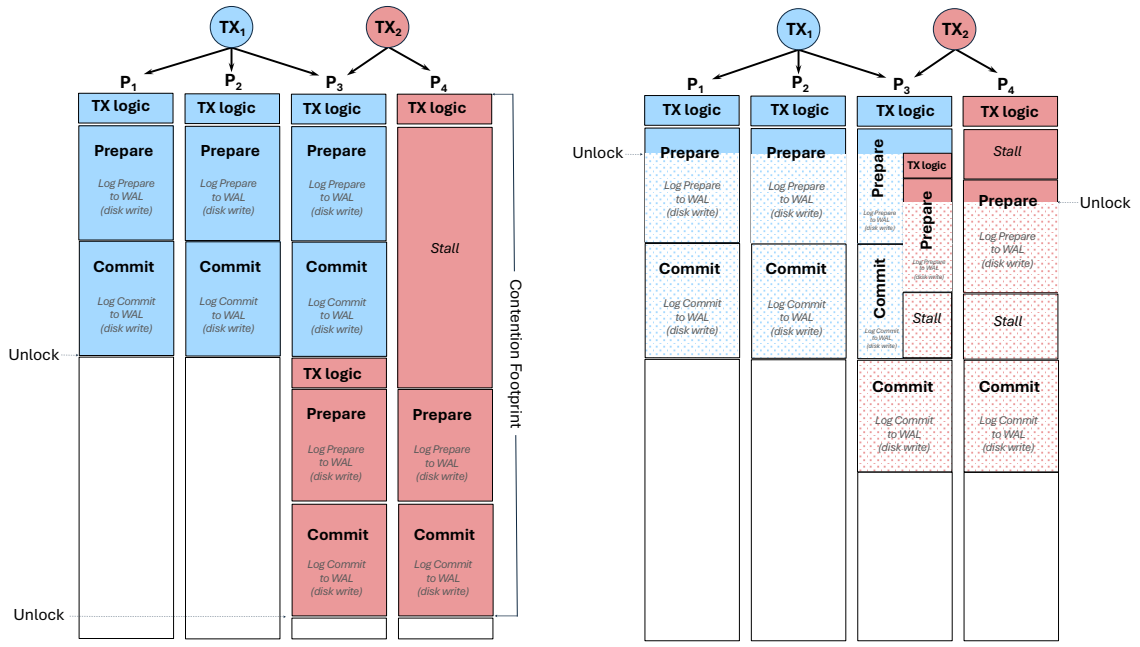
- We introduce an adaptive commit protocol that allows each participant to independently make early lock release decisions based on its local contention and Resolver load, combining the strengths of Strict-2PC and Pipelined-2PC commit strategies.
- We evaluate our system under a wide range of workload patterns and Resolver capacities, showing that our adaptive design consistently outperforms fixed commit strategies, providing better throughput, lower latency, and increased robustness to contention and resource imbalance.

The rest of the paper is organized as follows: Section 5.3 provides background on distributed commit protocols. Section 5.4 describes the design of our Resolver-based Pipelined-2PC implementation. Section 5.5 presents our adaptive commit protocol. Section 5.6 evaluates our system experimentally. Section 5.7 discusses related work, Section 5.8 discusses future work, and Section 5.9 concludes.

5.3 Background

This section provides an overview of the two foundational commit protocols: Strict-2PC and Pipelined-2PC. They form the basis of the adaptive protocol introduced in this paper. Figure 5.2 illustrates their behavior using an example: Figure 5.2(a) shows the execution under Strict-2PC, while Figure 5.2(b) presents the same transaction sequence under Pipelined-2PC.

The example involves two write-only transactions, T_1 and T_2, executing concurrently across four participants, P_1 through P_4, each corresponding to a data item accessed by the transactions. Transaction T_1 writes data from P_1, P_2 and P_3, while T_2 operates on P_3 and P_4. We refer to the initial phase where transactions stage writes as the *transaction logic*. In the figure, solid-colored regions indicate the period during which locks are held, while dashed-colored regions denote periods after locks are released. Although both transactions are submitted concurrently, T_1 arrives first and acquires exclusive locks on P_1, P_2, and P_3, thereby blocking T_2 on P_3. This introduces a dependency between the two transactions. However, T_2 can proceed with its transaction logic on



(a) Strict-2PC (b) Pipelined-2PC

Fig. 5.2: Strict 2PC vs Pipelined 2PC. (a) In Strict 2PC, locks are held throughout the entire commit protocol, resulting in long lock hold times and increased contention. (b) In Pipelined 2PC, locks are released earlier — immediately after the prepare record is appended to the WAL buffer — allowing subsequent transactions to proceed sooner and reducing contention, but introducing commit-time dependencies that require additional coordination to ensure correctness.

P_4, where there are no conflicts. Note that each transaction's write operations are applied independently across keys, allowing them to proceed in parallel on different participants.

Strict-2PC. In Strict-2PC (Figure 5.2(a)), the coordinator initiates the execution of transaction T_1, which acquires locks on each accessed participant (P_1, P_2, P_3) according to strict 2PL to execute its transaction logic. These locks are retained throughout the entire transaction lifecycle, spanning both execution and commit phases, thereby ensuring strict serializability. Once T_1 completes its transaction logic and the client requests to commit, the coordinator initiates the *prepare phase* by sending prepare requests to all participants. Each participant validates local constraints, ensures durability by writing a prepare record to persistent storage (i.e., Write Ahead Log or WAL), and responds with either a vote-commit or vote-abort. Importantly, locks remain held after voting to commit. The coordinator waits for all participants to vote before proceeding to the commit phase. If all participants vote to commit, the coordinator proceeds to the commit phase, logs the commit

decision, and sends commit messages to participants. Each participant writes a commit record to WAL, applies the committed changes to the local storage and, after completing the commit (shown at the "Unlock" point), releases its locks. This conservative design results in long lock hold durations, as shown by the solid-colored regions in Figure 5.2(a), which reflect the total contention footprint.

While Strict-2PC guarantees atomicity and strict serializability, the cumulative time locks are held — even after local work is done — limits concurrency, especially as commit coordination latency grows. This effect is clearly seen for T_2: while T_2 can process its transaction logic on P_4, it remains blocked on P_3 until T_1 fully completes its commit phase and releases the lock. As a result, T_2's contention footprint extends across much of T_1's commit lifecycle, limiting concurrency despite potential opportunities for overlap.

Pipelined-2PC. In Pipelined-2PC (Figure 5.2(b)), the protocol modifies the commit sequence to reduce lock holding times by allowing participants to release locks earlier in the commit process. As before, the coordinator initiates execution for T_1, acquiring locks on participants P_1, P_2, and P_3 during the transaction logic phase. Once transaction logic completes, the prepare phase begins as in Strict-2PC, with prepare requests sent to all participants.

However, the key difference is that participants may release their locks immediately after appending the prepare record to an in-memory WAL buffer — before the record is durably persisted to disk. This early lock release enables subsequent transactions to acquire locks and proceed without waiting for the preceding transaction to commit. In the example, once P_3 appends the prepare record for T_1 to its WAL buffer, it releases its lock, allowing T_2 to acquire the lock on P_3 and begin its transaction logic and prepare phase. The actual persistence to disk happens asynchronously, preserving the order of operations while enabling higher concurrency.

Despite this improved concurrency, early lock release introduces commit-time dependencies: because T_2 acquired locks on P_3 before T_1 committed, T_2's commit correctness now depends on T_1 successfully committing. If T_1 were to abort after releasing locks, any dependent transaction like T_2 would also need to abort to preserve correctness — a phenomenon known as *cascading aborts*.

To manage these dependencies, additional coordination is required during commit processing.

After it receives all prepare responses, the coordinator determines the commit outcome. As before, if all participants vote to commit, commit records are written to WAL and changes are finalized. The solid and dashed regions in Figure 5.2(b) show that, compared to Strict-2PC, lock hold durations are significantly shortened, allowing T_2 to proceed earlier and reducing overall contention footprint — at the cost of introducing dependency tracking and potential cascading aborts.

5.4 Dependency Tracking with Resolver

Our proposed Pipelined-2PC protocol employs a centralized Resolver component to manage transaction dependencies and enforce correct commit ordering. Figure 5.3 depicts the architectural design and communication patterns among the Resolver, transaction coordinator, and participants.

At its core, the Resolver maintains a dependency graph that tracks unresolved dependencies among in-flight transactions. Each transaction is represented as a node, and each directed edge indicates a dependency relationship — typically arising from conflicting accesses to one or more shared participants. The Resolver maintains a per-participant commit queue, where each queue holds transactions that are eligible to commit at that participant. A transaction becomes eligible when all of its dependencies are resolved. A dependency is considered resolved if the transaction it depends on has either already committed or was newly unblocked in the current resolution pass. By resolution pass, we are referring to the cascade of unblocks triggered by a transaction commit: its direct dependents may become eligible, which in turn may unblock their dependents, and so on. This recursive unblocking process is implemented in Algorithm 1 (lines 24–28), where each resolved transaction is enqueued at the corresponding participant's commit queue in dependency order.

This design enables the Resolver to coordinate group commits efficiently while preserving commit-order consistency. For each transaction, the Resolver logs its commit intent to durable storage and tracks which participant queues it has been inserted into. Once all relevant participants

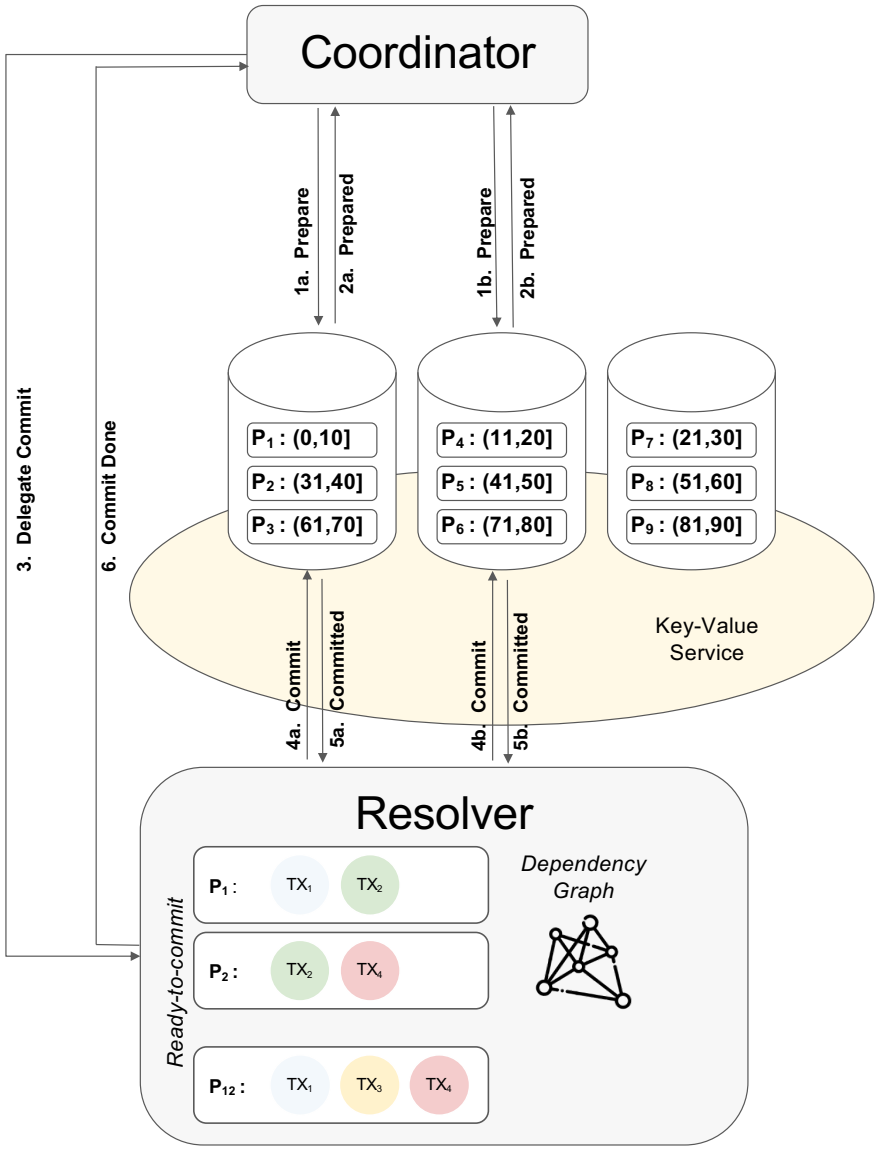


Fig. 5.3: Resolver architecture showing communication between resolver, coordinator, and participants

Algorithm 3 Resolver API

```
1: function COMMITONBEHALF(txn, dependencies)
2:     Create a response channel for txn
3:     if dependencies not empty then
4:         Add txn and dependencies to dependency graph
5:     else
6:         Add txn to all its participants' queues
7:         Call TRIGGERCOMMIT
8:     Wait for txn's commit on response channel
9: function TRIGGERCOMMIT
10:    for each participant with non-empty queue do
11:        txns ← remove all txns from participant.queue
12:        log commit decision for txns if not already logged
13:        Send commit msg with txns to participant
14:    Wait for all participants to respond
15:    // Collect txns whose all participants have committed
16:    done_txns ← get transactions fully completed
17:    for each txn in done_txns do
18:        Notify through txn's response channel
19:    Call REGISTERCOMMITTEDTXS on done_txns
20: function REGISTERCOMMITTEDTXS(txns)
21:    worklist ← txns
22:    while worklist not empty do
23:        Pop txn from worklist
24:        for each dependent of txn do
25:            Remove dependency edge (txn → dependent)
26:            if dependent has no remaining dependencies then
27:                Add dependent to worklist
28:                Add dependent to all its participants' queues
29:    Call TRIGGERCOMMIT
```

have committed their respective queued transactions, the Resolver notifies the coordinator that the transaction is fully committed.

The Resolver exposes the following API as shown in Algorithm 3. The `commitOnBehalf` function is called by coordinators to delegate commit coordination to the Resolver. When invoked, it first creates a response channel for the transaction to establish communication with the coordinator for reporting the final commit status. If no dependencies are detected, the transaction is immediately added to all its participants' queues and the Resolver triggers a commit by calling

`triggerCommit`. Otherwise, the transaction and its dependencies are added to the dependency graph. The coordinator then waits for the transaction's commit status on the response channel.

The `triggerCommit` internal function orchestrates the group commit process by extracting ready transactions from participant queues and initiating their commit phase. It sends commit messages — each containing a batch of transactions — to the appropriate participants and waits for their responses. As participants reply, the Resolver identifies transactions whose all participants have committed. For each such transaction, it notifies the original coordinator via the transaction's pre-established response channel. Finally, it invokes `registerCommittedTXs` to process the completed transactions, which updates the dependency graph and may trigger additional commits for newly unblocked transactions.

The `registerCommittedTXs` function updates the dependency graph when transactions complete their commit phase. It can be invoked internally by the Resolver when it coordinates a commit to completion, or externally by coordinators that bypassed the Resolver for transactions with no active dependencies. Upon receiving a list of committed transactions, the Resolver removes them from the dependency graph and iteratively identifies and processes newly unblocked transactions, cascading this process until no further transactions are unblocked. These are then enqueued in their corresponding participant queues. If any new transactions became ready to commit, the Resolver invokes `triggerCommit` to initiate group commits as previously described. Even when the Resolver is bypassed, notifying it of committed transactions remains essential to unblock any downstream transactions that may depend on them. This dependency tracking mechanism ensures that transactions are committed in the correct order while enabling efficient batching through coordinated group commits.

5.5 Sangria

Our goal is to allow a distributed database to dynamically adapt its commit algorithm based on runtime conditions. To this end, we introduce Sangria, a novel distributed commit protocol that generalizes both Strict-2PC and Pipelined-2PC.

Algorithm 4 COORDINATOR COMMIT PROTOCOL

```
1:  procedure COMMIT(transaction_id)
2:      // Prepare Phase
3:      RL ← getLocalStats(resolver_load)
4:      CL ← getLocalStats(contention_level)
5:      for all participant ∈ participants do
6:          send PREPARE(RL, CL) to participant
7:      wait for all PREPARERESPONSES
8:      collect locking decisions and dependencies from responses
9:      // Decide Commit Path
10:     if dependencies ≠ ∅ then
11:         // Delegate to Resolver
12:         send COMMITONBEHALF with dependencies to Resolver
13:         await Resolver completion
14:     else
15:         // Direct Commit Path
16:         persist transaction decision to durable storage
17:         for all participant ∈ participants do
18:             send COMMIT to participant
19:         wait for all participants to complete
20:         if any participant released locks early then
21:             notify Resolver of committed transaction (async)
```

5.5.1 Overview

Sangria allows each participant involved in a transaction to independently decide whether to release locks early (pipelining) or hold them conservatively (strict). These decisions are informed by lightweight signals collected and reported by the coordinator, such as the current load of the Resolver signaling its ability to resolve dependencies efficiently, and metrics about the contention level across participants. Participants communicate their decisions back to the coordinator, which determines if any interaction with the Resolver is needed. This hybrid design enables Sangria to dynamically range from fully serialized to aggressively pipelined commit modes — based on live workload conditions.

5.5.2 Coordinator Commit Protocol

Algorithm 4 presents the pseudocode for the coordinator's behavior during commit. The coordinator collects statistics locally about the current load of the Resolver and the contention levels of participants and includes this information in the PREPARE messages sent to each participant. The PREPARERESPONSES contain each participant's locking decision (early release or conservative hold) and any declared dependencies on previously executing transactions. If any participant reports unresolved dependencies (due to early lock release over conflicting keys), the coordinator invokes the `commitOnBehalf` API on the Resolver, delegating commit resolution. The resolver then manages dependency tracking and group commits as discussed in §5.4. If no dependencies are reported, the coordinator commits the transaction directly: it writes the transaction decision to durable storage, sends COMMIT messages to all participants, and waits for acknowledgments. Finally, if any participant performed early lock release, the coordinator asynchronously informs the Resolver when the transaction is committed via the `registerCommittedTXs` API, enabling it to unblock downstream dependents.

5.5.3 Participant Prepare Procedure

Algorithm 5 describes the participant's logic. Upon receiving a prepare request, the participant first validates the transaction and performs local checks. Then, it tracks and updates dependencies for the requested key, accordingly. If the key was last updated by a transaction whose commit is still pending, the participant records a dependency on that transaction. It then evaluates whether early lock release is beneficial and chooses the commit mode accordingly, guided by the contention level and the Resolver load included in the request. Depending on the commit mode selected, the key's dependency info is updated either to the current transaction or to EMPTY. The participant appends a prepare record to its WAL buffer and then releases the lock if the commit mode selected is early release. Once the WAL buffer is flushed, the participant returns a PREPARERESULT that includes an optional dependency and the selected commit mode. These results allow the coordinator to decide whether to delegate to the Resolver or commit the transaction directly. Note that

Algorithm 5 PARTICIPANT PREPARE PROCEDURE

```
1:  procedure PREPARE(transaction, prepare_request)
2:      Perform validation checks
3:      RL ← prepare_request.resolver_load
4:      CL ← prepare_request.contention_level
5:      KEY ← prepare_request.key
6:      // Record and update dependencies
7:      if KEY has existing pending writer transaction then
8:          record dependency on pending writer transaction
9:      commit_mode ← CHOOSECOMMITSTRATEGY(RL, CL)
10:     if commit_mode is early release then
11:         set key's pending writer to current transaction
12:     else
13:         set key's pending writer to EMPTY
14:     append prepare record to WAL buffer
15:     if commit_mode is early release then
16:         release lock
17:     flush WAL buffer asynchronously
18:     wait for WAL flush completion
19:     return PREPARERESULT with possible dependency and commit_mode
```

for simplicity, we describe each participant as managing a single key and omit differences in behavior when a key is read but not written. In practice, participants may hold multiple keys, and the protocol handles read-only keys differently; dependency tracking and commit mode selection are performed independently per key.

5.5.4 Discussion

Sangria generalizes both Strict-2PC and Pipelined-2PC by treating commit strategy as a per-participant decision rather than a system-wide choice. The centralized Resolver allows the system to maintain correctness in the presence of early lock release by managing dependencies and coordinating group commits in the right order. Importantly, enabling this adaptive behavior required lightweight extensions to the 2PC protocol. We augmented PREPARE and PREPARERESPONSE messages to carry contextual information: Resolver load, locking decisions, and dependencies. This extra information enables participants to make informed decisions and coordinators to route transactions along the most efficient commit path.

5.5.5 Adaptive Decision Logic

As a proxy for Resolver load, the coordinator periodically pings the Resolver to obtain the current length of its queue of transactions waiting to commit. To improve stability and avoid reacting to transient spikes, the coordinator maintains a history of the last 200 queue-length observations and computes the average. For contention, the coordinator tracks the number of open client connections and monitors how requests are distributed across keys. Both the Resolver load and contention metrics are included in the PREPARE request sent to each participant. Participants also evaluate local contention proxies such as the number of transactions waiting on key-level locks and the number of pending commits.

Based on these inputs, they apply empirically-tuned thresholds to decide whether early lock release is likely to be beneficial. These thresholds define the regimes in which Sangria should perform early lock release or fall back to traditional locking, allowing the protocol to adapt dynamically to workload and system state.

5.5.6 Correctness Guarantees

Our adaptive protocol maintains the same correctness guarantees as the underlying Strict-2PC and Pipelined-2PC protocols.

Atomicity. All-or-nothing execution is preserved through the 2PC structure. The coordinator ensures that either all participants commit or all abort, regardless of the selected commit mode.

Consistency. Database consistency is maintained through proper isolation mechanisms. In Strict-2PC mode, strict 2PL ensures serializable execution. In Pipelined-2PC mode, the Resolver component manages dependencies to prevent violations of serializability.

Durability. Both protocol modes ensure durability through write-ahead logging (WAL) as described in the algorithms. Participants write prepare records to persistent storage before voting to commit, guaranteeing that committed transactions survive system failures.

Isolation. The protocol enforces isolation using different strategies based on the chosen mode. In Strict-2PC mode, 2PL guarantees strict serializability. In contrast, when operating in Pipelined-2PC mode, the Resolver component (see Algorithm 3) upholds the same isolation level by monitoring transaction dependencies and enforcing correct commit order.

5.6 Evaluation

In this section, we evaluate the performance and adaptability of Sangria, our proposed commit protocol. We seek to answer the following key questions:

- **Q1:** How does Sangria perform under workloads with different contention levels and different Resolver capacities?
- **Q2:** How effectively does Sangria respond to runtime variations in contention intensity and Resolver capacity?
- **Q3:** How does Sangria perform with mixed workloads of varying contention levels?
- **Q4:** How well does the centralized Resolver perform in grouping and processing transactions in batches?

5.6.1 Methodology

Baselines. We evaluate Sangria by comparing it against the two foundational baselines, Strict-2PC and our version of Pipelined-2PC that uses the centralized Resolver.

Metrics. We focus on two primary metrics:

- **Throughput:** number of committed transactions per second.

Machine. All experiments are run on a 16 core Cloudlab server with 2 threads per core and 128 GiB RAM (type c220g1). We use a single-node setup to precisely control the resources allocated

to each component by pinning services to specific CPU cores. Our focus is not on stretching scalability to extreme cluster sizes, but rather on understanding the performance trade-offs of adaptivity in a tightly controlled environment.

Experimental Setup. Our prototype builds on Chardonnay [159], a distributed key-value store that employs strict 2PL in combination with 2PC to ensure atomicity and strict serializability. To enable a clean evaluation of the tradeoff space between workload contention levels and Resolver capacity, we create an execution environment that is isolated from abort-induced artifacts. This is achieved by avoiding deadlocks, which are the primary source of aborts in 2PL-based systems. With Chardonnay, the set of keys accessed by each transaction is known ahead of time, which allows it to acquire locks in a globally consistent order, ensuring that circular wait conditions do not occur.

The system shards data horizontally across shared-nothing range servers, each responsible for a configurable number of key ranges. In our setup, one CPU core is dedicated to the Resolver, while two additional cores are used to generate background load on the Resolver in order to modulate its available capacity. The remaining cores support the main workload generator (whose performance we evaluate), as well as the coordinator and range server components.

5.6.2 Workloads

We test Sangria under two workloads: the standard Yahoo! Cloud Serving Benchmark (YCSB) benchmark [89] and a custom synthetic workload generator we developed. In the experiments that use the custom workload generator, the number of keys are fixed, transactions are generated concurrently by multiple clients, and each transaction accesses exactly two distinct keys selected uniformly at random. To control contention, we introduce a tunable parameter called `concurrency-level`, which determines the number of clients issuing transactions in parallel. Each client operates in a closed loop, issuing one transaction at a time. For a fixed key set, increasing `concurrency-level` raises the probability of key overlap between transactions, thereby increasing contention. This concurrency model mirrors that of widely used benchmarking

tools such as BenchBase [161], where the same parameter is used to modulate parallelism and contention pressure during workload execution.

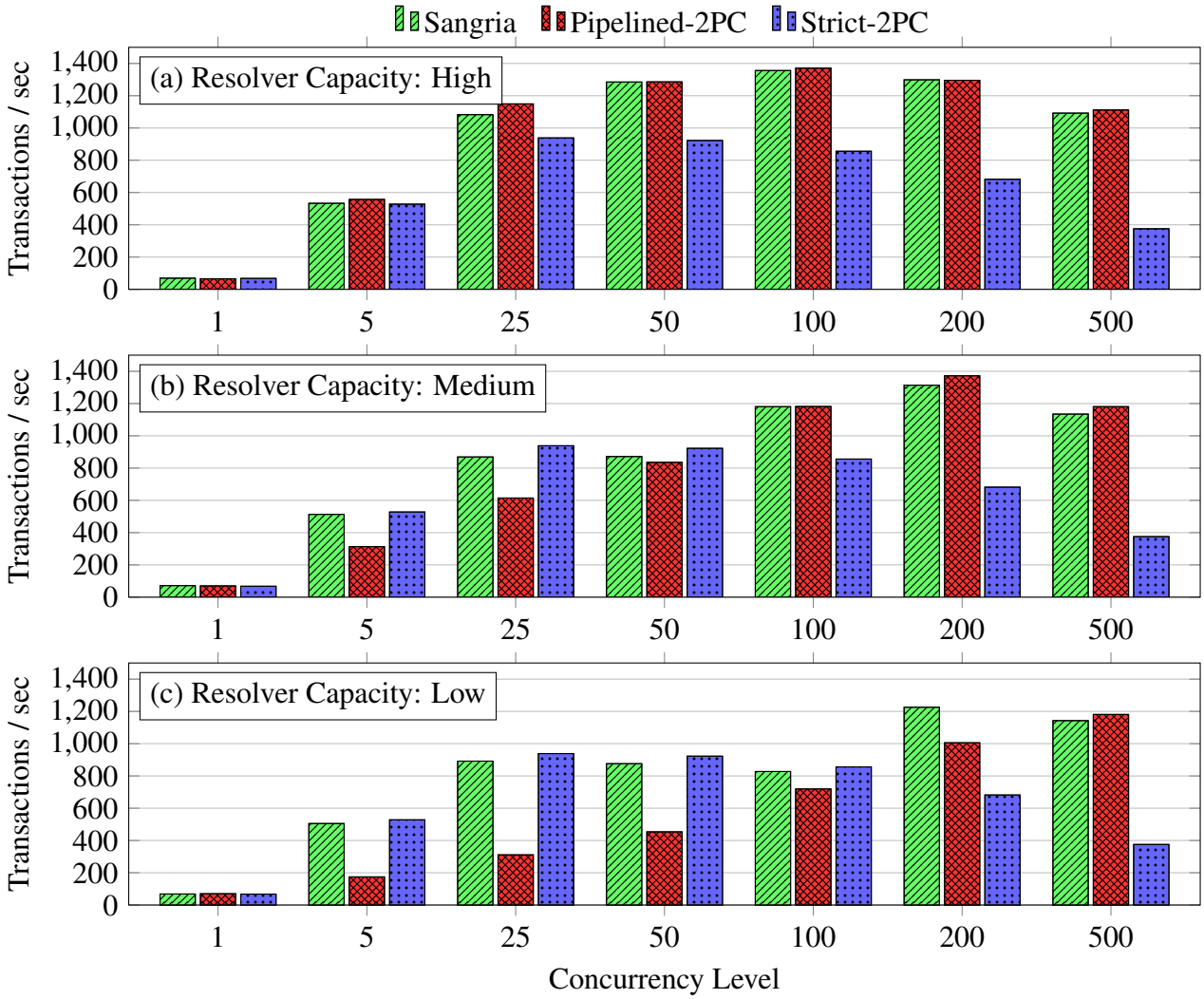


Fig. 5.4: (Q1) Throughput of the three protocols as a function of workload contention (x-axis: concurrency level) under three different Resolver capacity settings (a) high capacity (no background load), (b) medium capacity (moderate background load), and (c) low capacity (heavy background load). Sangria is able to adapt its behavior based on the Resolver's capacity and workload contention, matching or exceeding the throughput of the baselines in all regimes.

5.6.3 Contention vs. Resolver Capacity (Q1)

We evaluate how Sangria performs across a spectrum of contention levels and Resolver capacities, using two complementary experimental setups: our custom workload generator and the

YCSB.

Custom Workload. We use 50 keys and vary contention using the concurrency-level parameter, as described above. Figure 5.4 presents the results of this experiment. It consists of three subfigures, each corresponding to a different level of Resolver capacity, which we control using a secondary workload generator. This generator executes background transactions that consume the Resolver's resources by issuing RPCs, entering the dependency graph, acquiring internal locks, and so on. We control this background load by configuring the generator with three different concurrency levels: 0, 100, and 1000, which we refer to as high, medium, and low Resolver capacity, respectively. When the background load is zero, the Resolver is idle and has maximal capacity. As a proxy for Resolver load, we monitor the number of transactions waiting in response channels — i.e., transactions that are blocked waiting for the Resolver to finalize their commit.

Each subfigure shows throughput as a function of the main workload's contention level. On the x-axis, contention is varied by adjusting the concurrency level of the main workload generator. As an internal proxy for contention, the coordinator monitors system-wide metrics such as the number of in-flight transactions (i.e., active client connections) and the distribution of requests across keys. In addition, each participant tracks local indicators of contention, including the number of transactions waiting on key-level locks and the number of transactions whose commits are pending.

In Figure 5.4(a), the Resolver operates at full capacity with no external load (high capacity). In this case, the Pipelined-2PC baseline consistently outperforms Strict-2PC as contention increases. With no Resolver bottlenecks, the benefits of early lock release always dominate, enabling higher concurrency. The Sangria protocol closely follows the performance of Pipelined-2PC in this setting, as it correctly favors early lock release when the Resolver is under minimal load.

In Figure 5.4(b), Resolver capacity is moderately constrained. We now observe a more nuanced behavior: at low contention levels, the overhead of coordinating through the Resolver outweighs the modest benefits of pipelining, making Strict-2PC more efficient. However, once the contention level crosses a threshold (around concurrency equal to 100), pipelining becomes es-

sential to mitigate queuing delays caused by lock contention. Sangria identifies this crossover point and adapts accordingly, falling back to Strict-2PC when pipelining is harmful and switching only when pipelining yields tangible performance gains. We tune this threshold empirically in our system.

In Figure 5.4(c), the Resolver is heavily overloaded (low capacity). Under these conditions, pipelining offers diminishing returns when contention is low-to-moderate: the Resolver becomes the bottleneck, and the added overhead of dependency tracking and queueing outweighs the benefits of early lock release. Only at the highest contention levels (concurrency ≥ 200) do the advantages of pipelining re-emerge. Again, Sangria adapts its behavior accordingly, choosing Strict-2PC in the low or moderate contention regime and gradually shifting toward pipelining as contention intensifies.

Notice that across all Resolver capacities, at the lowest contention level (concurrency=1), all three protocols converge in performance: transactions do not conflict, dependencies never arise, and the Resolver is bypassed entirely. The Pipelined-2PC baseline still asynchronously notifies the Resolver of committed transactions to ensure correct dependency resolution for any future dependents, but this occurs outside the critical path and thus has no effect on throughput. Conversely, Sangria leans toward the Strict-2PC behavior in this regime to avoid introducing unnecessary overhead to the Resolver from commit-notification messages.

Across all scenarios, Sangria navigates the two-dimensional trade-off between Resolver load and contention effectively. It learns when to pipeline and when to revert to strict mode, delivering performance close to the best static strategy in each regime.

YCSB. In (Figure 5.5), we introduce contention through a different mechanism than in prior sections. We fix the keyspace to 50 keys and configure the benchmark to use only read-modify-write operations. Each YCSB operation is wrapped as a transaction by appending a commit step, effectively transforming the workload into a transactional one. Each transaction operates on a single key, which is sampled from a Zipfian distribution. We run the benchmark with 50 concurrent clients (threads), each issuing transactions in a closed loop. We then vary the Zipfian constant

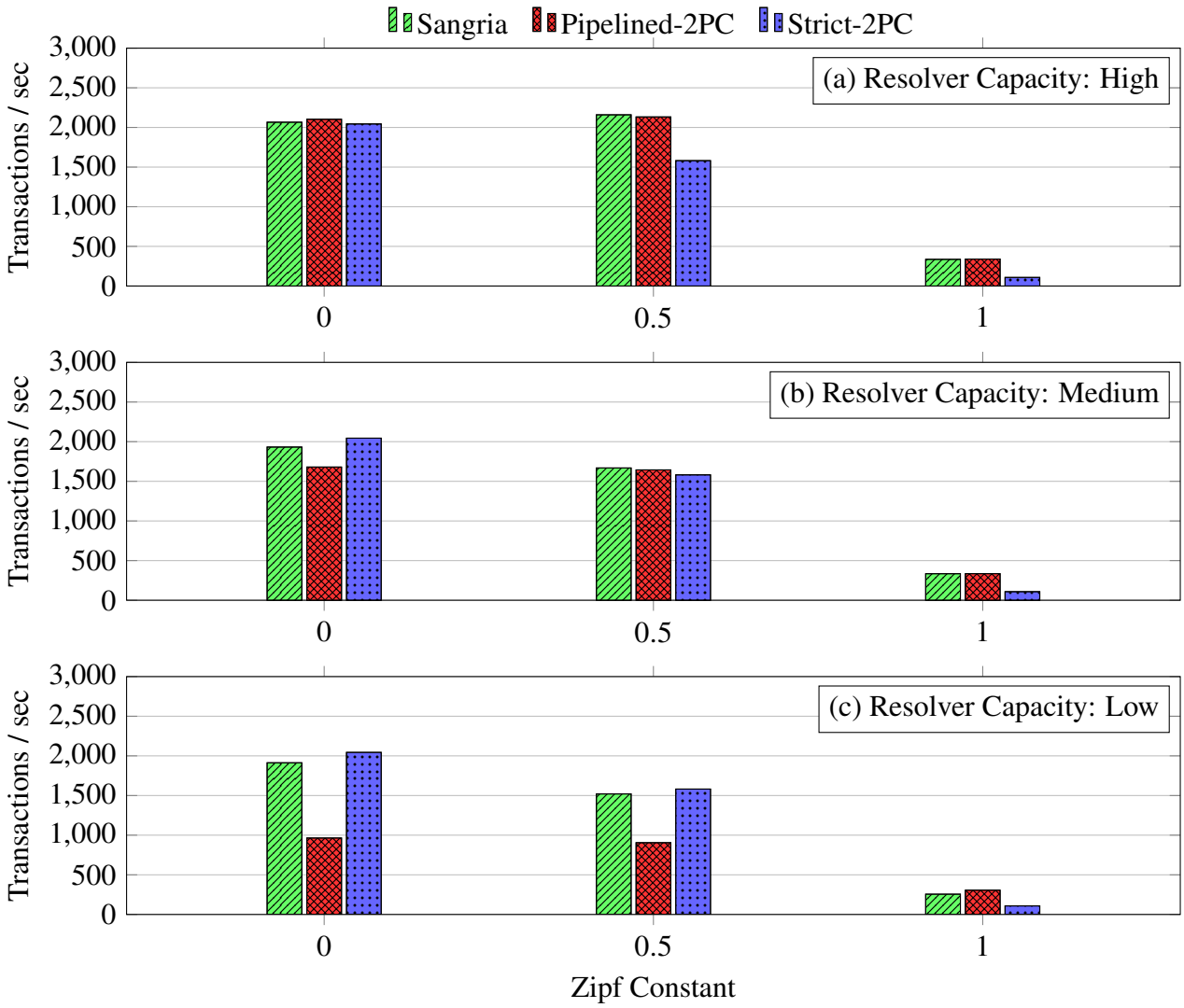


Fig. 5.5: (Q1) YCSB: Throughput comparison as contention increases (by increasing the Zipf Constant) under varying Resolver capacities.

to control the skew in key access: a value of 0.0 yields uniform key selection, while increasing the constant toward 1.0 leads to increasingly skewed workloads where a few keys are accessed disproportionately often.

This setup allows us to systematically sweep from low- to high-contention scenarios while holding other parameters constant. As shown in the figure, Sangria continues to match or exceed the performance of the best static protocol across all regimes. In low-skew settings, it behaves similarly to Strict-2PC by holding locks until commit. As the skew increases and contention concentrates on a small subset of keys, Sangria dynamically switches to early lock release for those keys — matching the behavior of Pipelined-2PC — while conservatively holding locks for less contended keys. This enables it to consistently deliver high throughput across the spectrum.

5.6.4 Online Adaptation (Q2)

While the previous set of experiments fixed workload characteristics — such as contention level and Resolver capacity — in advance, the following experiments introduce dynamic runtime variation in both dimensions using the custom workload generator. We issue 16,000 transactions, each reading and then writing two keys selected uniformly at random without replacement.

Online Workload Contention Shift. In Figure 5.6, we present the throughput of each protocol under dynamic workload contention, plotted across three different Resolver capacities: high, medium, and low. In all cases, the number of keys is fixed to 50, and the workload alternates at runtime between low-contention (concurrency = 25) and high-contention (concurrency = 500) phases. These alternating phases simulate realistic runtime variations that a distributed system may encounter in production. The two baselines, Strict-2PC and Pipelined-2PC, apply a fixed commit strategy for all transactions, regardless of workload shifts. As a result, they exhibit lower overall throughput, as they cannot adapt to changing contention dynamics. In contrast, Sangria uses runtime feedback from the coordinator — such as local traffic of transactions and the load observed on the Resolver — to make commit path decisions on a per-transaction basis. This enables Sangria to choose the most appropriate behavior at each phase. When contention spikes, pipelining becomes

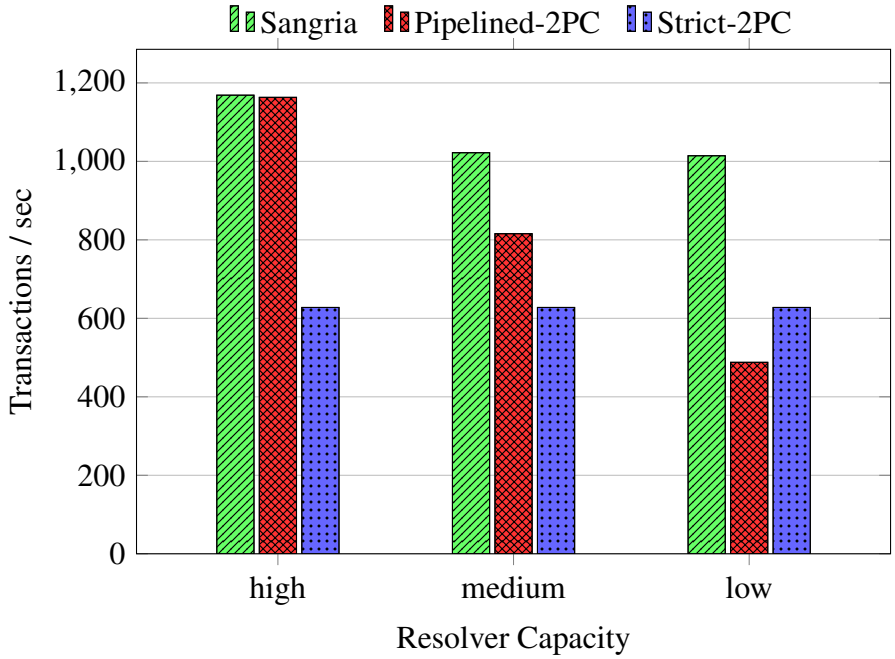


Fig. 5.6: (Q2) Throughput of each protocol as workload contention alternates between low and high phases at runtime, under three different resolver capacities (high, medium, low). Sangria adapts to changing contention, matching or exceeding the best static baseline in each regime.

essential to mitigate long lock hold times; as expected, Strict-2PC suffers during these periods due to its conservative locking and so its overall throughput declines. Conversely, when contention is low and the capacity of the Resolver is limited (medium and low), the overhead of dependency tracking and queue management outweighs the benefits of pipelining, causing Pipelined-2PC to degrade and its overall throughput decreases similarly. Sangria is able to detect all these regimes and switch accordingly. This ability to adapt to runtime variation demonstrates the robustness of Sangria and highlights the importance of dynamic commit path selection.

Online Resolver Capacity Shift. We next evaluate how the protocols respond to online shifts in Resolver capacity, while holding each experiment's workload contention level fixed. Specifically, we fix again the number of keys to 50 and evaluate three different contention regimes — low (concurrency = 5), moderate (concurrency = 50), and high (concurrency = 500) — as shown in Figure 5.7. For each contention regime, we dynamically vary Resolver load at runtime by adjusting the concurrency level of a secondary background workload that interacts with the Resolver — alternating between phases of low capacity (high background concurrency, heavily overloading the

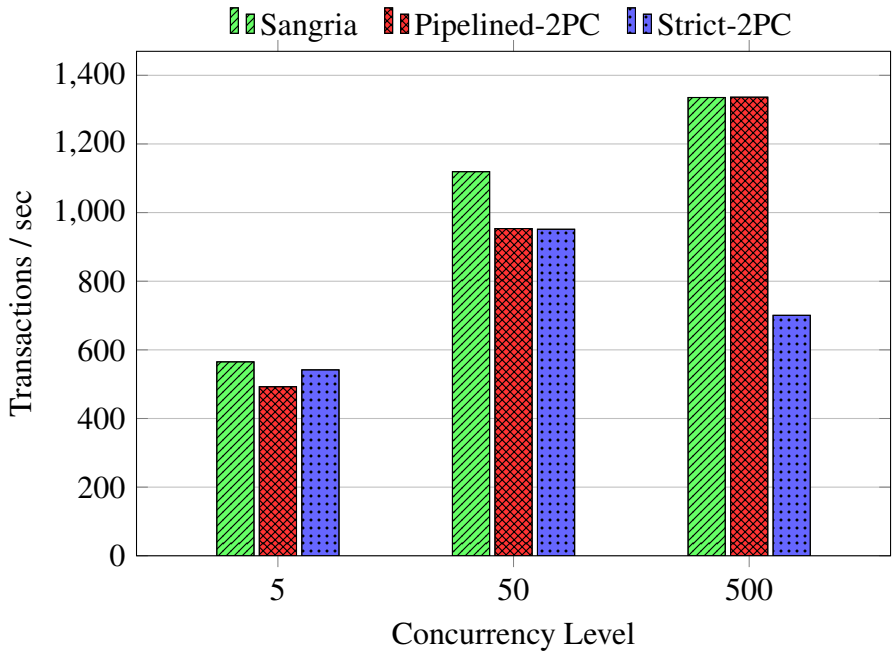


Fig. 5.7: (Q2) Throughput of each protocol as resolver capacity alternates between high and low phases at runtime, under three different concurrency levels (5, 50, 500). Sangria adapts to changing resolver capacity, matching or exceeding the best static baseline in each regime.

Resolver) and high capacity (no background traffic). Across all contention regimes, the baselines suffer when conditions deviate from their ideal operating assumptions. The Strict-2PC baseline performs reliably when the Resolver is overloaded (i.e., low Resolver capacity), since it avoids any coordination with the Resolver. However, it fails to leverage pipelining even when the Resolver is idle yielding lower overall throughput. On the other hand, Pipelined-2PC performs best when the Resolver is responsive, but incurs high coordination overhead and stalls when Resolver capacity is low. Sangria, by contrast, dynamically adjusts its commit behavior in real time. As a result, it matches or exceeds the best-performing baseline in each phase. For extremely high contention levels (concurrency = 500), Sangria mirrors the behavior of Pipelined-2PC, as it identifies the substantial benefits of pipelining even under low Resolver capacity.

5.6.5 Mixed Workloads (Q3)

To further highlight the fine-grained adaptability of Sangria, we run a workload composed of both high-contention and low-contention transactions. Specifically, the keyspace is partitioned into

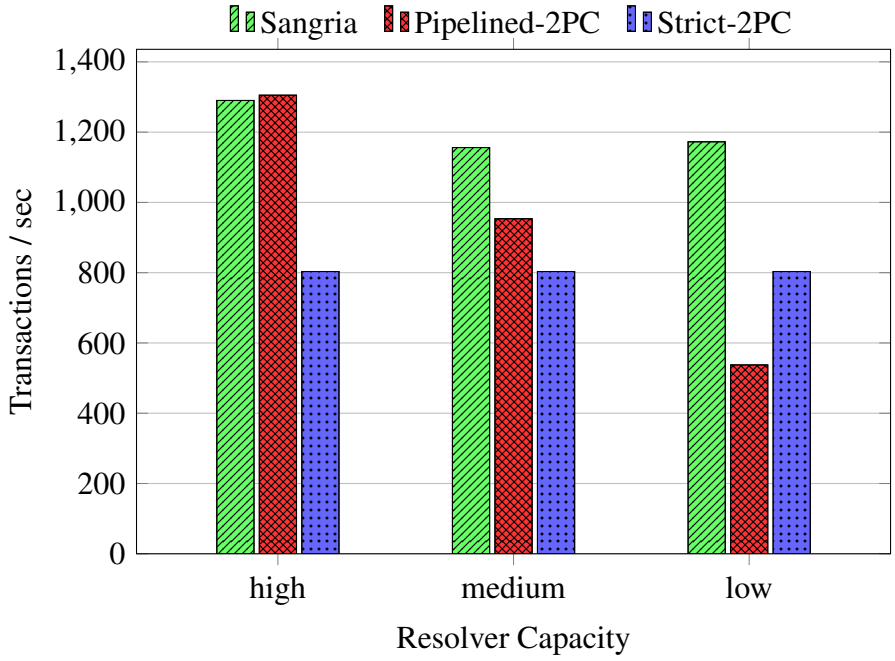


Fig. 5.8: (Q3) Throughput of each protocol under a mixed workload with both high-contention (hot) and low-contention (cold) key regions, across three resolver capacities (high, medium, low). Sangria dynamically applies pipelining for hot keys and strict commit for cold keys, matching or exceeding the best baseline in each region.

100 keys, of which 50 are designated as hot and 50 as cold. We generate two disjoint workloads targeting each key region: the hot keyset is accessed by 500 concurrent clients (high contention), while the cold keyset is accessed by only 25 (low contention). This mixed workload introduces asymmetric contention across the keyspace, simulating more realistic scenarios encountered in multi-tenant or skewed-access applications.

In Figure 5.8, we report results for three levels of Resolver capacity: high, medium, and low. The two foundational baselines, Strict-2PC and Pipelined-2PC, are static by design: they apply the same commit strategy to all transactions, regardless of key-level contention or system state. As a result, Pipelined-2PC incurs unnecessary dependency-tracking overhead when operating over the cold keyset, where contention is minimal and Resolver use is avoidable. Conversely, Strict-2PC suffers when operating over the hot keyset, as it holds locks throughout the entire commit process, degrading concurrency.

In contrast, Sangria leverages its fine-grained decision mechanism to adjust behavior on a per-

participant basis. Each participant uses information about its local contention and the current load on the Resolver, to determine whether to release locks early (enabling pipelining) or to follow a traditional commit path. As a result, participants serving hot keys opt for pipelined behavior to improve concurrency, while those operating on cold keys bypass the Resolver and avoid unnecessary overhead.

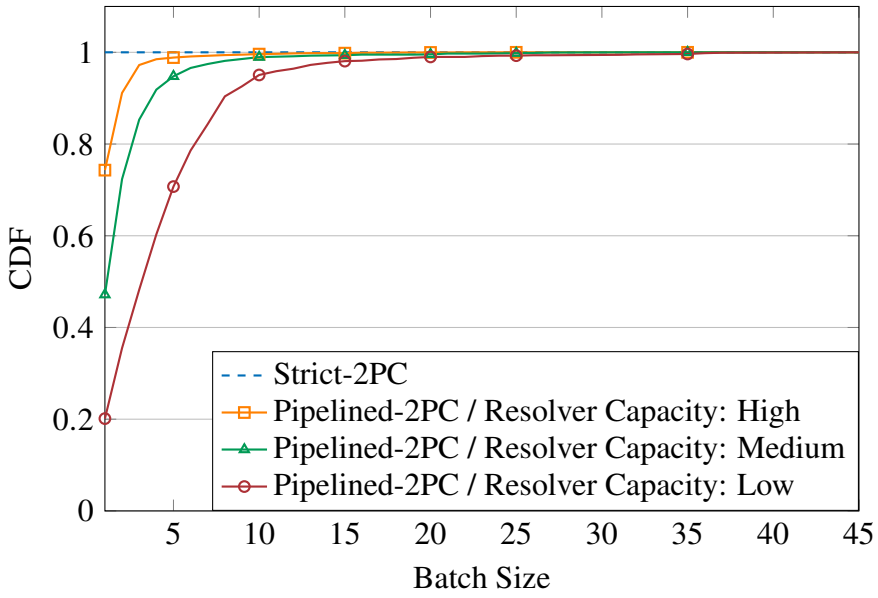
This ability to apply different commit strategies within the same transaction workload leads to significant gains. Across all Resolver configurations, Sangria consistently matches or outperforms both baselines. For high Resolver capacity, as shown in prior experiments, Pipelined-2PC consistently outperforms Strict-2PC. Sangria achieves higher throughput by adapting its commit behavior to the local contention profile and global system load, demonstrating that static policies are insufficient in complex, mixed contention workloads.

5.6.6 Resolver Performance (Q4)

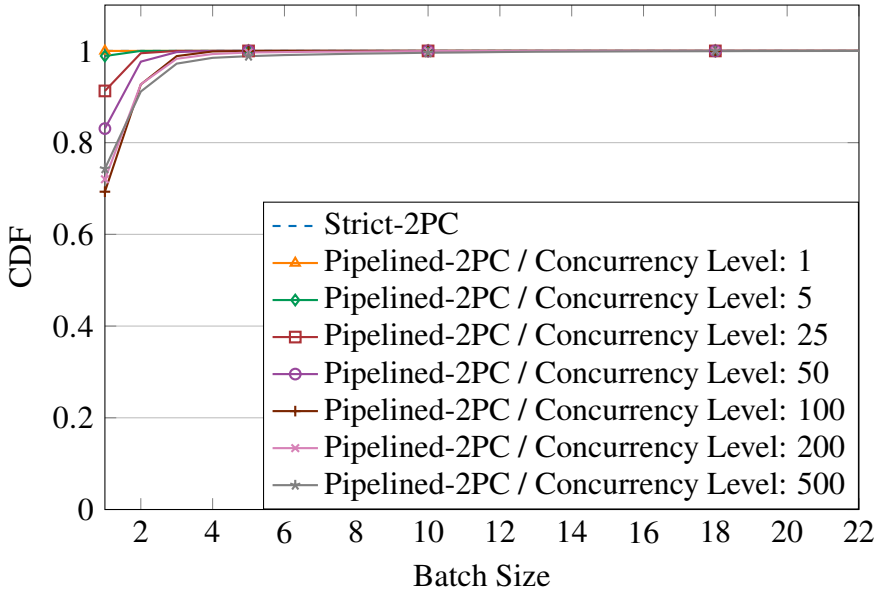
To better understand the internal behavior and batching effectiveness of the Resolver, we conduct a set of experiments measuring its ability to group transactions that become ready concurrently, thereby enabling them to commit in a single batched operation to the storage layer. In this evaluation, we focus on the sizes of commit batches sent to the participants. For each batch created by the Resolver, we record the number of transactions it contains, and we plot the cumulative distribution function (CDF) over all observed batch sizes.

For comparison, we include the Strict-2PC baseline, which — by design — does not perform any batching. Each transaction commits in isolation, resulting in a constant batch size of one. As expected, its CDF is a step function that jumps immediately to 1. This serves as a lower bound and helps visualize the relative gains of dependency-aware batching enabled by the Resolver.

We present two subfigures in Figure 5.9. In Figure 5.9(a), we fix the concurrency level to 500 — representing high contention — and vary the Resolver's capacity by adjusting the background load from the secondary workload. In Figure 5.9(b), we fix the Resolver's capacity to its maximum (i.e., no background load) and vary the primary workload's concurrency level to vary contention.



(a) Batch size CDF under different Resolver capacities.



(b) Batch size CDF under different concurrency levels.

Fig. 5.9: (Q4) Cumulative distribution function (CDF) of batch sizes for commit groups formed by the Resolver. (a) Varying Resolver capacity under high contention (concurrency = 500) shows that lower capacity leads to larger batch sizes due to more transactions accumulating before being unblocked. (b) Varying workload contention under maximum Resolver capacity demonstrates that higher concurrency increases batching opportunities, while low contention results in mostly single-transaction commits.

In Figure 5.9(a), we observe that under high contention, the Resolver is often able to batch together multiple transactions that become unblocked simultaneously. As the Resolver's capacity decreases, the CDF curve becomes smoother and shifts rightward, indicating larger batch sizes. This happens because reduced Resolver responsiveness causes transactions to accumulate in the dependency graph. When a dependency is resolved, a larger set of transactions may be unblocked at once and committed as a group. Thus, even though the Resolver is slower, the batching effect helps mitigate performance degradation — highlighting a self-compensating behavior under high contention and constrained Resolver capacity.

In contrast, Figure 5.9(b) shows the impact of varying contention levels under maximmum Resolver capacity. As expected, higher concurrency increases the likelihood of transaction dependencies, leading to more opportunities for batching. The CDF curves for higher concurrency levels (e.g., 500) have more gradual slopes than those with lower concurrency (e.g., 5), reflecting an increase in batch size. However, batch sizes remain smaller compared to those in Figure 5.9(a), since the responsive Resolver processes transactions quickly, reducing the likelihood that large groups of transactions will accumulate before resolution.

These results demonstrate the ability of the Resolver to maximize batching opportunities. Even under constrained capacity, it can batch transactions to improve overall throughput.

5.7 Related Work

Early Lock Release. Relaxing strict 2PL to increase concurrency by releasing locks at earlier stages [156, 162] is a technique that was applied to single-node databases with large buffer pools, since a transaction may run in less time than it takes to log the transaction's commit record on stable storage [157], and was later rigorously formalized and further developed in works such as *controlled lock violation* [157] and Bamboo [155]. The distributed forms of this technique have recently been proposed to optimize performance in distributed databases [163]. Orleans [154] pioneered the use of a distributed form of early lock release by releasing all locks of a transaction during phase one of 2PC. These works always apply early lock release and, therefore, are not

adaptive.

Commit Dependencies and Resolvers. The notion of commit dependency was introduced in the ACTA framework [164]. We know of few prior works that use the concept.

In Speculative Locking (SL) [165], if a transaction T_1 updates x and a later transaction T_2 reads x, then T_2 speculates by having two incarnations, T_{21} that reads T_1's before-image of x and T_{22} that reads its after-image [42]. T_{21} and T_{22} both take a commit dependency on T_1. If T_1 commits, T_{22} is retained, else T_{21} is retained. The simulation study [165] of a distributed DBMS shows that SL gets better throughput than 2PL by overlapping speculative executions of T_2 with T_1, at the cost of more CPU load. By contrast, in our design, T_2 only takes a dependency on T_1 after T_1 terminates so it has no more CPU load.

Microsoft's Hekaton uses a more limited form of commit dependency [166]. It allows T_2 to take a commit dependency on T_1 if T_2 started after T_1 finished execution and entered the validation phase but has not yet committed. Thus, it benefits from overlapping T_2's execution with T_1's validation.

Orleans' [154] transactions track their commit dependencies on other transactions that have yet to complete 2PC and implement cascading abort by aborting if any commit dependency fails to commit. An earlier design [153] used a centralized resolver, called Transaction Manager, in a fashion similar to the Resolver in our design, but had to involve the transaction manager in every transaction, causing it to be a scalability bottleneck.

Handling Skewed Workloads. TurboDB [167] takes a different approach in handling skewed workloads, by assigning records in the highly contended subset of the database to a single node subsystem (called the turbo) within an otherwise distributed and sharded DBMS. The turbo can utilize single-node optimizations and performance multipliers allowing it to handle the skewed, contended records efficiently. For this approach to work effectively, it assumes that the popularity of records is stable over time, making it less dynamic and adaptive than our Sangria design. Furthermore, while we also introduce a singleton component in the Resolver, the work it needs to do per transaction is much less than the turbo in TurboDB, and hence we expect to be able to achieve higher scalability.

5.8 Future Work

This work opens several avenues for future exploration. First, while Sangria uses empirically tuned thresholds to determine when to switch between commit protocols, future systems could benefit from learning-based approaches that automatically infer regime boundaries based on runtime signals. Additionally, richer proxies for workload contention such as the average time transactions wait on participant locks could improve decision making even more.

Second, our current batching mechanism at the Resolver is opportunistic. Introducing lightweight scheduling techniques could help shape batch formation in a more controlled way, potentially improving commit efficiency and better exploiting the pipelining benefits under diverse load conditions.

Finally, our results suggest that adaptivity in distributed transaction processing is a promising and underexplored direction. Beyond commit protocol selection, adaptive techniques could be extended to other layers of the transaction stack, including concurrency control mechanisms or protocols that benefit from early lock release. Developing general principles for runtime-aware adaptation in distributed systems remains an exciting area for future work.

5.9 Conclusions

We presented Sangria, an adaptive 2PC protocol that dynamically switches between strict and pipelined commit strategies based on runtime conditions. By monitoring real-time signals such as workload contention and Resolver load, Sangria selects the most efficient commit path, achieving up to 1.61× higher throughput than the best static baseline in our experiments.

Our key contribution is a runtime-aware decision mechanism that adapts commit behavior to live workload conditions while preserving full ACID semantics. To our knowledge, Sangria is the first commit protocol to incorporate such adaptive capabilities.

Conclusion

This thesis explored the design of efficient systems under resource constraints, spanning two distinct domains: privacy-preserving computation and distributed transaction processing. In the first part of the thesis, we treated privacy as a scarce and quantifiable system resource. Through the systems DPack, Turbo, and Cookie Monster, we developed new techniques for maximizing utility under fixed differential privacy (DP) budgets. These systems addressed different layers of the privacy stack—from workload scheduling to caching to on-device budgeting for DP-based ad measurement —but shared a common goal: increasing the usefulness of private data while respecting strict privacy guarantees. Collectively, these contributions offer practical ways to close the growing gap between the theoretical promise of DP and its real-world applicability.

The second part of the thesis shifted to a more classical systems challenge: improving the throughput of distributed transaction processing. With Sangria, we showed how commit protocols can dynamically adapt to runtime conditions to better manage coordination overhead and contention. While this work does not involve privacy, it reinforces a central thesis goal: resource-aware design that adjusts to workload and system variability to improve efficiency.

Together, these contributions highlight two broader takeaways. First, systems can benefit from treating non-traditional resources—like privacy budgets—as first-class constraints, worthy of algorithmic and systems-level attention. Second, adaptability is key: whether in scheduling DP workloads or coordinating transactions, systems that respond to changing conditions can deliver significantly better performance under tight constraints.

References

[1] C. Dwork, A. Smith, T. Steinke, and J. Ullman, "Exposed! A survey of attacks on private data," *Annual Review of Statistics and Its Application*, 2017.

[2] N. Carlini, C. Liu, Ú. Erlingsson, J. Kos, and D. Song, "The secret sharer: Evaluating and testing unintended memorization in neural networks," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds., USENIX Association, 2019, pp. 267–284.

[3] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *sp*, 2017.

[4] N. Carlini et al., "Extracting training data from large language models," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds., USENIX Association, 2021, pp. 2633–2650.

[5] M. Nasr et al., *Scalable extraction of training data from (production) language models*, 2023. arXiv: 2311.17035 [cs.LG].

[6] I. Dinur and K. Nissim, "Revealing information while preserving privacy," in *pods*, 2003.

[7] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *tcc*, 2006.

[8] 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)*, USENIX Association, Jul. 2021, pp. 55–74, ISBN: 978-1-939133-22-9.

[9] M. Lécuyer, R. Spahn, K. Vodrahalli, R. Geambasu, and D. Hsu, "Privacy Accounting and Quality Control in the Sage Differentially Private ML Platform," in *sosp*, 2019.

[10] N. Küchler, E. Opel, H. Lycklama, A. Viand, and A. Hithnawi, "Cohere: Managing differential privacy in large scale systems," in *2024 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2024, pp. 991–1008.

[11] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in multi-resource clusters," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 65–80, ISBN: 978-1-931971-33-1.

[12] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA: USENIX Association, Mar. 2016, pp. 407–424, ISBN: 978-1-931971-29-4.

[13] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang, "Multiresource allocation: Fairness–efficiency tradeoffs in a unifying framework," *IEEE/ACM Transactions on Networking*, vol. 21, no. 6, pp. 1785–1798, 2013.

[14] A. Gutman and N. Nisan, "Fair allocation without trade," in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, ser. AAMAS '12, Valencia, Spain: International Foundation for Autonomous Agents and Multiagent Systems, 2012, 719âĂŞ728, ISBN: 0981738125.

[15] D. C. Parkes, A. D. Procaccia, and N. Shah, "Beyond dominant resource fairness: Extensions, limitations, and indivisibilities," *ACM Transactions on Economics and Computation (TEAC)*, vol. 3, no. 1, pp. 1–22, 2015.

[16] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14, Chicago, Illinois, USA: Association for Computing Machinery, 2014, 455âĂŞ466, ISBN: 9781450328364.

[17] Q. Weng et al., "MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA: USENIX Association, Apr. 2022, pp. 945–960, ISBN: 978-1-939133-27-4.

[18] M. Backes, P. Berrang, M. Humbert, and P. Manoharan, "Membership privacy in microRNA-based studies," in *ccs*, 2016.

[19] C. Dwork, A. Smith, T. Steinke, J. Ullman, and S. Vadhan, "Robust traceability from trace amounts," in *focs*, 2015.

[20] N. Homer et al., "Resolving individuals contributing trace amounts of DNA to highly complex mixtures using high-density SNP genotyping microarrays," *PLoS Genetics*, 2008.

[21] B. Jayaraman and D. Evans, "Evaluating differentially private machine learning in practice," in *usenixsec*, 2019.

[22] S. Vadhan, "The complexity of differential privacy," in *Tutorials on the Foundations of Cryptography*, 2017.

[23] I. Mironov, "Rényi Differential Privacy," in *Computer Security Foundations Symposium (CSF)*, 2017.

[24] Google Differential Privacy, https://github.com/google/differential-privacy/tree/main/python/dp_accounting, 2022.

[25] Google, *TensorFlow Privacy*, https://github.com/tensorflow/privacy, Accessed: 2020-11-10.

[26] Facebook, *Opacus*, https://opacus.ai/, Accessed: 2020-11-10.

[27] J. Murtagh and S. Vadhan, "The Complexity of Computing the Optimal Composition of Differential Privacy," in *Theory of Cryptography*, Berlin, Germany: Springer, Dec. 2015, pp. 157–175.

[28] TensorFlow Extended Guide, https://www.tensorflow.org/tfx/guide/examplegen, 2022.

[29] 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*, vol. 2020, no. 2, pp. 230–250, 2020.

[30] S. Berghel et al., "Tumult Analytics: a robust, easy-to-use, scalable, and expressive framework for differential privacy," *arXiv*, Dec. 2022. eprint: 2212.04133.

[31] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack problems*. Springer, 2004, ISBN: 978-3-540-40286-2.

[32] Gurobi Optimization, *Gurobi Optimization homepage*, www.gurobi.com/, 2021.

[33] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," Tech. Rep., 2011.

[34] T. Luo, M. Pan, P. Tholoniat, A. Cidon, R. Geambasu, and M. Lécuyer, *Privacy Resource Scheduling (extended version)*, https://github.com/columbia/privatekube, 2021.

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

[36] V. Feldman and T. Zrnic, "Individual privacy accounting via a renyi filter," in *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[37] M. Lécuyer, "Practical Privacy Filters and Odometers with RÃl'nyi Differential Privacy and Applications to Differentially Private Deep Learning," in *arXiv, v2*, 2021.

[38] Goop generalized mixed integer optimization in Go, *Goop homepage*, https://github.com/mit-drl/goop/, 2021.

[39] Simpy, *Discrete event simulation for Python*, https://simpy.readthedocs.io/en/latest/index.html, 2020.

[40] Q. Li, Z. Wu, Z. Wen, and B. He, "Privacy-preserving gradient boosting decision trees," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, AAAI Press, 2020, pp. 784–791.

[41] N. Grislain and J. Gonzalvez, "Dp-xgboost: Private machine learning at scale," *CoRR*, vol. abs/2110.12770, 2021. arXiv: 2110.12770.

[42] M. Abadi et al., "Deep learning with differential privacy," in *ccs*, 2016.

[43] P. Kairouz, B. McMahan, S. Song, O. Thakkar, A. Thakurta, and Z. Xu, "Practical and private (deep) learning without sampling or shuffling," in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, 2021, pp. 5213–5225.

[44] 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).*, Hong Kong, China: Association for Computational Linguistics. https://nijianmo.github.io/amazon/index.html, Nov. 2019, pp. 188–197.

[45] L. T. Kou and G. Markowsky, "Multidimensional bin packing algorithms," *IBM Journal of Research and development*, vol. 21, no. 5, pp. 443–448, 1977.

[46] Y. Azar, I. R. Cohen, S. Kamara, and B. Shepherd, "Tight bounds for online vector bin packing," in *Proceedings of the forty-fifth annual ACM symposium on Theory of Computing*, 2013, pp. 961–970.

[47] G. J. Woeginger, "There is no asymptotic PTAS for two-dimensional vector packing," *Information Processing Letters*, vol. 64, no. 6, pp. 293–297, 1997.

[48] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "Graphene: Packing and dependency-aware scheduling for data-parallel clusters," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, Savannah, GA, USA: USENIX Association, 2016, 81âĂŞ97, ISBN: 9781931971331.

[49] J. Gu et al., "Tiresias: A GPU cluster manager for distributed deep learning," in *USENIX NSDI*, 2019, pp. 485–500.

[50] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, D. G. Andersen and S. Ratnasamy, Eds., USENIX Association, 2011.

[51] L. Yu, L. Liu, C. Pu, M. E. Gursoy, and S. Truex, "Differentially private model publishing for deep learning," in *sp*, 2019.

[52] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang, "Learning differentially private recurrent language models," in *iclr*, 2018.

[53] B. Barak, K. Chaudhuri, C. Dwork, S. Kale, F. McSherry, and K. Talwar, "Privacy, accuracy, and consistency too: A holistic solution to contingency table release," in *sigmod*, 2007.

[54] J. Xu, Z. Zhang, X. Xiao, Y. Yang, G. Yu, and M. Winslett, "Differentially private histogram publication," in *icde*, 2012.

[55] OpenDP, `https://smartnoise.org/`, Accessed: 2020-11-10.

[56] IBM, *Diffprivlib*, `https://github.com/IBM/differential-privacy-library`, Accessed: 2020-12-7.

[57] Google, *Differential Privacy*, `https://github.com/google/differential-privacy/`, Accessed: 2020-11-10.

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

[59] R. Cummings, S. Krehbiel, K. A. Lai, and U. Tantipongpipat, "Differential privacy for growing databases," in *nips*, 2018.

[60] F. D. McSherry, "Privacy integrated queries: An extensible platform for privacy-preserving data analysis," in *sigmod*, 2009.

[61] D. Proserpio, S. Goldberg, and F. McSherry, "Calibrating data to sensitivity in private data analysis: A platform for differentially-private analysis of weighted datasets," *vldb*, 2014.

[62] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for MapReduce.," in *nsdi*, 2010.

[63] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler, "GUPT: Privacy preserving data analysis made easy," in *Proc. of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

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

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

[66] A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in *sp*, 2008.

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

[68] *Citibike system data*, `https://www.citibikenyc.com/system-data`, 2018.

[69] D. Desfontaines, *Real world DP use-cases*, `https://desfontain.es/privacy/real-world-differential-privacy.html`, Accessed: 2023-04-13.

[70] S. Bavadekar et al., "Google COVID-19 Search Trends Symptoms Dataset: Anonymization Process Description (version 1.0)," *arXiv*, Sep. 2020. eprint: `2009.01265`.

[71] R. Rogers et al., "Linkedin's audience engagements api: A privacy preserving data analytics system at scale," *arXiv preprint arXiv:2002.05839*, 2020.

[72] 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)*, 2020, pp. 535–551.

[73] J. M. Abowd et al., "The 2020 census disclosure avoidance system topdown algorithm," *Harvard Data Science Review*, no. Special Issue 2, 2022.

[74] I. Dinur and K. Nissim, "Revealing information while preserving privacy," in *sigmod*, 2003.

[75] 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*, 2010, pp. 61–70.

[76] 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.

[77] S. R. Ganta, S. Kasiviswanathan, and A. Smith, "Composition attacks and auxiliary information in data privacy," in *kdd*, 2008.

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

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

[80] J. Hsu et al., "Differential privacy: An economic method for choosing epsilon," vol. 2014, Jul. 2014.

[81] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends® in Theoretical Computer Science*, vol. 9, no. 3–4, pp. 211–407, 2014.

[82] 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*, 2013, pp. 29–42.

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

[84] 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.*, vol. 2022, no. 1, pp. 253–273, 2022.

[85] S. Vadhan, "The Complexity of Differential Privacy," in *Tutorials on the Foundations of Cryptography*, Cham, Switzerland: Springer, Apr. 2017, pp. 347–450.

[86] T. Liu, G. Vietri, T. Steinke, J. Ullman, and S. Wu, "Leveraging public data for practical private query release," in *Proceedings of the 38th International Conference on Machine Learning*, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, 2021, pp. 6968–6977.

[87] *Tableau*, `https://public.tableau.com/app/discover`, Accessed: 2023-04-13.

[88] *Citibike tableau story*, `https://public.tableau.com/app/profile/james.jeffrey/viz/CitiBikeRideAnalyzer/CitiBikeRdeAnalyzer`, Accessed: 2023-04-13.

[89] 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*, 2010, pp. 143–154.

[90] 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*, 2020, pp. 191–208.

[91] 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*, 2012, pp. 53–64.

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

[93] S. Aydöre et al., "Differentially private query release through adaptive projection," in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, 2021, pp. 457–467.

[94] N. Beckmann, H. Chen, and A. Cidon, "Lhd: Improving cache hit rate by maximizing hit density," in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'18, Renton, WA, USA: USENIX Association, 2018, 389âĂŞ403, ISBN: 9781931971430.

[95] J. Yang, Y. Yue, and K. V. Rashmi, "A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter," *ACM Trans. Storage*, vol. 17, no. 3, 2021.

[96] H. Che, y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *Selected Areas in Communications, IEEE Journal on*, vol. 20, pp. 1305–1314, Oct. 2002.

[97] N. M. Johnson, J. P. Near, and D. Song, "Towards practical differential privacy for SQL queries," *Proc. VLDB Endow.*, vol. 11, no. 5, pp. 526–539, 2018.

[98] 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*, USENIX Association, 2020, pp. 1065–1081.

[99] K. Amin, J. Gillenwater, M. Joseph, A. Kulesza, and S. Vassilvitskii, "Plume: Differential Privacy at Scale," *arXiv*, Jan. 2022. eprint: 2201.11603.

[100] I. Kotsogiannis et al., "Privatesql: A differentially private sql query engine," *Proc. VLDB Endow.*, vol. 12, no. 11, 1371âĂŞ1384, 2019.

[101] 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.*, vol. 16, no. 4, pp. 574–586, 2022.

[102] C. Li, G. Miklau, M. Hay, A. McGregor, and V. Rastogi, "The matrix mechanism: Optimizing linear counting queries under differential privacy," *VLDB J.*, vol. 24, no. 6, pp. 757–781, 2015.

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

[104] 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*, ser. Proceedings of Machine Learning Research, vol. 119, PMLR, 2020, pp. 9765–9774.

[105] R. McKenna, G. Miklau, M. Hay, and A. Machanavajjhala, "Optimizing error of high-dimensional statistical queries under differential privacy," *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1206–1219, 2018.

[106] C. Li, M. Hay, G. Miklau, and Y. Wang, "A data- and workload-aware query answering algorithm for range queries under differential privacy," *Proc. VLDB Endow.*, vol. 7, no. 5, pp. 341–352, 2014.

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

[108] T.-H. Hubert Chan, E. Shi, and D. Song, "Private and continual release of statistics," in *Automata, Languages and Programming*, S. Abramsky, C. Gavoille, C. Kirchner, F. Meyer auf der Heide, and P. G. Spirakis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 405–417, ISBN: 978-3-642-14162-1.

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

[110] *Intelligent tracking prevention 2.3*, https://webkit.org/blog/9521/intelligent-tracking-prevention-2-3/, 2019.

[111] *Over a decade of anti-tracking work at mozilla*, https://blog.mozilla.org/en/privacy-security/mozilla-anti-tracking-milestones-timeline/, 2022.

[112] A. Chavez, *A new path for privacy sandbox on the web*, https://privacysandbox.com/news/privacy-sandbox-update/, 2024.

[113] *Icloud private relay overview*, https://www.apple.com/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf, 2021.

[114] Apple, Inc., *Apple announces powerful new privacy and security features*, https://www.apple.com/newsroom/2023/06/apple-announces-powerful-new-privacy-and-security-features/, 2023.

[115] G. P. Sandbox, *Privacy Sandbox for the Web*, https://privacysandbox.com/intl/en_us/open-web, 2023.

[116] *Privacy preserving ad click attribution for the web*, https://webkit.org/blog/8943/privacy-preserving-ad-click-attribution-for-the-web/, 2019.

[117] G. Chrome, *Federated Learning of Cohorts (FLoC)*, https://privacysandbox.com/proposals/floc/.

[118] *Understanding appleâĂŹs private click measurement*, https://blog.mozilla.org/en/mozilla/understanding-apples-private-click-measurement/, 2022.

[119] *GoogleâĂŹs floc is a terrible idea*, https://www.eff.org/deeplinks/2021/03/googles-floc-terrible-idea, 2021.

[120] G. Chrome, *Protected Audience API overview*, https://developers.google.com/privacy-sandbox/relevance/protected-audience.

[121] *Private advertising technology community group*, https://www.w3.org/community/patcg, 2024.

[122] *Attribution reporting api (ara))*, https://github.com/WICG/attribution-reporting-api/blob/main/AGGREGATE.md, 2022.

[123] *Interoperable private attribution (ipa))*, https://github.com/patcg-individual-drafts/ipa, 2022.

[124] *Private ad measurement (pam)*, https://github.com/patcg-individual-drafts/private-ad-measurement, 2023.

[125] *Hybrid proposal*, https://github.com/patcg-individual-drafts/hybrid-proposal, 2024.

[126] H. Ebadi, D. Sands, and G. Schneider, "Differential Privacy: Now it's Getting Personal," in *Proceedings of the 42nd Annual ACM SIGPLAN SIGACT Symposium on Principles of*

Programming Languages, Mumbai India: ACM, Jan. 14, 2015, pp. 69–81, ISBN: 978-1-4503-3300-9.

[127] D. Kifer, S. Messing, A. Roth, A. Thakurta, and D. Zhang, "Guidelines for implementing and auditing differentially private systems," Tech. Rep., 2020.

[128] *Privacy-preserving attribution: Level 1*, https://private-attribution.github.io/api/, 2024.

[129] R. M. Rogers, A. Roth, J. Ullman, and S. Vadhan, "Privacy odometers and filters: Pay-as-you-go composition," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29, Curran Associates, Inc., 2016.

[130] Google, *Attribution reporting api with aggregatable reports*, https://github.com/WICG/attribution-reporting-api/blob/main/AGGREGATE.md#contribution-bounding-and-budgeting/, 2024.

[131] V. Feldman and T. Zrnic, "Individual privacy accounting via a rényi filter," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34, Curran Associates, Inc., 2021, pp. 28 080–28 091.

[132] D. Yu, G. Kamath, J. Kulkarni, T.-Y. Liu, J. Yin, and H. Zhang, "Individual Privacy Accounting for Differentially Private Stochastic Gradient Descent," *Transactions on Machine Learning Research*, Apr. 27, 2023.

[133] B. Ghazi, N. Golowich, R. Kumar, P. Manurangsi, and C. Zhang, "Deep Learning with Label Differential Privacy," in *Advances in Neural Information Processing Systems*, vol. 34, Curran Associates, Inc., 2021, pp. 27 131–27 145.

[134] *Patcg attribution synthetic data*, https://docs.google.com/document/d/1Vxq4LrMe3A2WIlu-7IYP1Hycr_nz3_qTpPAICX9fLcw, 2024.

[135] M. Tallis and P. Yadav, "Reacting to variations in product demand: An application for conversion rate (CR) prediction in sponsored search," *arXiv preprint arXiv:1806.08211*, 2018.

[136] C. Dwork and A. Roth, "The Algorithmic Foundations of Differential Privacy," *Foundations and Trends® in Theoretical Computer Science*, vol. 9, no. 3-4, pp. 211–407, 2013.

[137] F. D. McSherry, "Privacy integrated queries: An extensible platform for privacy-preserving data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09, New York, NY, USA: Association for Computing Machinery, Jun. 29, 2009, pp. 19–30, ISBN: 978-1-60558-551-2.

[138] S. P. Kasiviswanathan, H. K. Lee, K. Nissim, S. Raskhodnikova, and A. Smith, "What can we learn privately?" *SIAM Journal on Computing*, vol. 40, no. 3, pp. 793–826, 2011.

[139] E. Margolin, K. Newatia, T. Luo, E. Roth, and A. Haeberlen, "Arboretum: A planner for large-scale federated analytics with differential privacy," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23, , Koblenz, Germany: Association for Computing Machinery, 2023, 451âĂŞ465, ISBN: 9798400702297.

[140] A. Bittau et al., "Prochlo: Strong privacy for analytics in the crowd," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, Shanghai, China: Association for Computing Machinery, 2017, 441âĂŞ459, ISBN: 9781450350853.

[141] A. Cheu, A. Smith, J. Ullman, D. Zeber, and M. Zhilyaev, "Distributed differential privacy via shuffling," in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds., Cham: Springer International Publishing, 2019, pp. 375–403, ISBN: 978-3-030-17653-2.

[142] *Introducing private click measurement, pcm*, `https://webkit.org/blog/11529/introducing-private-click-measurement-pcm/`, 2021.

[143] B. Case et al., *Interoperable private attribution: A distributed attribution and aggregation protocol*, Cryptology ePrint Archive, Paper 2023/437, `https://eprint.iacr.org/2023/437`, 2023.

[144] M. Dawson et al., *Optimizing Hierarchical Queries for the Attribution Reporting API*, Comment: Appeared at AdKDD 2023 workshop; Final proceedings version, Nov. 27, 2023. arXiv: `2308.13510 [cs]`.

[145] H. Aksu et al., *Summary Reports Optimization in the Privacy Sandbox Attribution Reporting API*, Nov. 22, 2023. arXiv: `2311.13586 [cs]`.

[146] J. Delaney et al., *Differentially private ad conversion measurement*, 2024. arXiv: `2403.15224 [cs.CR]`.

[147] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, 463âĂŞ492, 1990.

[148] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, 624âĂŞ633, 1976.

[149] B. W. Lampson and D. B. Lomet, "A new presumed commit optimization for two phase commit," in *Proceedings of the 19th International Conference on Very Large Data Bases*, ser. VLDB '93, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, 630âĂŞ640, ISBN: 155860152X.

[150] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," *Proc. VLDB Endow.*, vol. 10, no. 5, 553âĂŞ564, 2017.

[151] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: Dependably fast multi-core in-memory transactions," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, Chicago, Illinois, USA: Association for Computing Machinery, 2017, 21âĂŞ35, ISBN: 9781450341974.

[152] E. Zamanian, J. Shun, C. Binnig, and T. Kraska, "Chiller: Contention-centric transaction execution and data partitioning for modern networks," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, 2020, 511âĂŞ526, ISBN: 9781450367356.

[153] T. Eldeeb and P. A. Bernstein, "Transactions for distributed actors in the cloud," Tech. Rep. MSR-TR-2016-1001, 2016.

[154] T. Eldeeb, S. Burckhardt, R. Bond, A. Cidon, J. Yang, and P. A. Bernstein, "Cloud actor-oriented database transactions in orleans," *Proc. VLDB Endow.*, vol. 17, no. 12, pp. 3720–3730, Aug. 2024.

[155] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a DHT," in *usenix*, 2004.

[156] E. Soisalon-Soininen and T. Ylönen, "Partial strictness in two-phase locking," in *Proceedings of the 5th International Conference on Database Theory*, ser. ICDT '95, Berlin, Heidelberg: Springer-Verlag, 1995, 139âĂŞ147, ISBN: 3540589074.

[157] G. Graefe, M. Lillibridge, H. Kuno, J. Tucek, and A. Veitch, "Controlled lock violation," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, New York, New York, USA: Association for Computing Machinery, 2013, pp. 85–96, ISBN: 9781450320375.

[158] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, 94âĂŞ162, Mar. 1992.

[159] T. Eldeeb, X. Xie, P. A. Bernstein, A. Cidon, and J. Yang, "Chardonnay: Fast and general datacenter transactions for On-Disk databases," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, Boston, MA: USENIX Association, Jul. 2023, pp. 343–360, ISBN: 978-1-939133-34-2.

[160] G. Prasaad, A. Cheung, and D. Suciu, "Handling highly contended oltp workloads using fast dynamic partitioning," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, 2020, 527âĂŞ542, ISBN: 9781450367356.

[161] A. Pavlo et al., *Benchbase: A benchmarking toolkit for database systems*, `https://github.com/cmu-db/benchbase`, GitHub repository, 2023.

[162] E. Soisalon-Soininen and T. Ylönen, "Partial strictness in two-phase locking," in *Proceedings of the 5th International Conference on Database Theory*, ser. ICDT '95, Berlin, Heidelberg: Springer-Verlag, 1995, 139âĂŞ147, ISBN: 3540589074.

[163] H. Guo, X. Zhou, and L. Cai, "Lock violation for fault-tolerant distributed database system," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 1416–1427.

[164] P. K. Chrysanthis and K. Ramamritham, "Acta: A framework for specifying and reasoning about transaction structure and behavior," *SIGMOD Rec.*, vol. 19, no. 2, 194âĂŞ203, 1990.

[165] P. Krishna Reddy and M. Kitsuregawa, "Speculative locking protocols to improve performance for distributed database systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 2, pp. 154–169, 2004.

[166] C. Diaconu et al., "Hekaton: Sql server's memory-optimized oltp engine," ser. SIGMOD '13, New York, New York, USA: Association for Computing Machinery, 2013, 1243âĂŞ1254, ISBN: 9781450320375.

[167] J. Lam, J. Helt, W. Lloyd, and H. Lu, "Accelerating skewed workloads with performance multipliers in the TurboDB distributed database," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1213–1228, ISBN: 978-1-939133-39-7.