TAOBench: An End-to-End Benchmark for Social Network Workloads

Audrey Cheng[†*], Xiao Shi[‡], Aaron Kabcenell[‡], Shilpa Lawande[‡], Hamza Qadeer[†], Jason Chan[†], Harrison Tin[†], Ryan Zhao[†], Peter Bailis[◊], Mahesh Balakrishnan[‡], Nathan Bronson[▽], Natacha Crooks[†], Ion Stoica[†]

[†]UC Berkeley, [‡]Meta, [◊]Sisu Data, [▽]Rockset

accheng@berkeley.edu,akabcenell@fb.com

ABSTRACT

The continued emergence of large social network applications has introduced a scale of data and query volume that challenges the limits of existing data stores. However, few benchmarks accurately simulate these request patterns, leaving researchers in short supply of tools to evaluate and improve upon these systems. In this paper, we present a new benchmark, TAOBench, that captures the social graph workload at Meta. We open source workload configurations along with a benchmark that leverages these request features to both accurately model production workloads and generate emergent application behavior. We ensure the integrity of TAOBench's workloads by validating them against their production counterparts. We also describe several benchmark use cases at Meta and report results for five popular distributed database systems to demonstrate the benefits of using TAOBench to evaluate system tradeoffs as well as identify and address performance issues. Our benchmark fills a gap in the available tools and data that researchers and developers have to inform system design decisions.

PVLDB Reference Format:
Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, Ion Stoica. TAOBench: An End-to-End Benchmark for Social Network Workloads. PVLDB, 15(9): 1965-1977, 2022.
doi:10.14778/3538598.3538616

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/audreyccheng/taobench.

1 INTRODUCTION

Despite the ubiquity of social networks, including those at Meta[1] [20], ByteDance [19], LinkedIn [40], Twitter [3], and WeChat [47], there is a lack of *publicly available, realistic workloads* to guide research on their underlying database infrastructure. In academia, this scarcity makes it difficult to probe the limits of existing systems and develop novel mechanisms to overcome them. In industry, it is challenging for practitioners to evaluate new features and resolve issues without a way to reproduce these request patterns.

While there are standard benchmarks for OLTP workloads, such as TPC-C [27], few equivalents exist for social networks. Most are derived from synthetic data [17, 23, 29, 45, 46] and have not been shown to fully capture the skew, high correlation, and read-heavy nature of these workloads [20, 24]. Others lack transactions [1, 2] or information about colocation preferences and constraints (e.g., data may have to reside on specific shards due to legal reasons).

To address the gap in representative workloads, we present TAOBench, an open-source benchmark that accurately simulates the production request patterns of an online social network. Since benchmarks are only as useful as the workloads they are derived from, we identify five crucial properties that should be captured by their request patterns. A comprehensive social network benchmark should 1) accurately emulate social network requests, 2) capture any transactional requirements, 3) express data colocation preferences and constraints, 4) model request distributions without prescriptive query types, and 5) exhibit multi-tenant behavior on shared data (Section 2). To satisfy these properties, we profile requests served by TAO, an online graph data store at Meta [20].

TAO is a read-optimized, geographically distributed data store that provides access to the social graph for diverse products and backend systems [20]. In aggregate, TAO serves over ten billion requests per second on a changing dataset of many petabytes. Its workload contains a variety of notable attributes. For example, read and write skew often manifest on different keys: over 99.0% of data items that are frequently written to are, on average, read less than once per day (Section 3). While the scale and features of this workload may be distinctive, TAO's API and data model, which TAOBench replicates, are intentionally simple. Consequently, they generalize to those used by other systems, including OLTP stores [6] and graph databases [9].

To accurately generate TAO's workloads at flexible scale, we characterize these request patterns and identify a small set of parameters, including transaction size, key to shard mapping, and frequency of operation types, that are sufficient to replicate production workloads. We then leverage these features in TAOBench to both accurately downscale Meta's social network workload and model emergent application behavior. Our parametrized framework is open-source and extensible, allowing it to simulate a range of different request patterns.

To illustrate TAOBench's applicability, we report on how Meta uses this tool to test new features, optimizations, and reliability

[*]Work done while at Meta.

Proceedings of the VLDB Endowment, Vol. 15, No. 9 ISSN 2150-8097.
doi:10.14778/3538598.3538616

[1]Formerly known as Facebook.

(e.g., hotspots, worst-case scenarios) as well as experiment with speculative workloads that would otherwise be difficult or infeasible to assess in production. We describe four examples: 1) analyzing new transaction use cases, 2) assessing contention under longer lock hold times, 3) evaluating new APIs, and 4) quantifying the performance of high fan-out transactions. Furthermore, we provide the results for TAOBench on five widely used distributed databases (Cloud Spanner, CockroachDB, PlanetScale, TiDB, YugabyteDB) to demonstrate how our benchmark can be used to study performance tradeoffs and identify optimization opportunities.

To the best of our knowledge, TAOBench is the first open-source benchmark that generates end-to-end, transactional request patterns derived from a large-scale social network and addresses the lack of representative workloads for a major application area. With our benchmark, we make Meta's social graph workload accessible to the database community and provide visibility into real-world challenges of supporting such workloads.

In summary, we make the following contributions in this paper:

- We identify a small set of representative features that accurately characterizes TAO's production workload (Section 3).
- We share Meta's social graph workload with the broader research community via workload configurations in our open-source benchmark (Section 4) and validate our methodology for accurately downscaling these request patterns (Section 5).
- We describe how Meta uses TAOBench to evaluate new features and test system reliability (Section 6).
- We provide results of our benchmark on five databases to demonstrate its usefulness as a tool for evaluating distributed data stores (Section 7).

2 MOTIVATION

Despite the enduring popularity of social network systems, researchers have limited tools to understand their performance. In particular, there is a shortage of benchmarks that generate realistic workloads for social graphs. The application context of a benchmark's workload determines its relevance and extensibility. Accordingly, we identify several crucial properties for benchmark workloads of this application domain: they should accurately emulate social network requests (**P1**), capture any transactional requirements (**P2**), express colocation preferences and constraints (**P3**), model request distributions without prescriptive query types (**P4**), and exhibit multi-tenant behavior on shared data (**P5**).

2.1 Desired Properties

We describe why each of these properties is necessary and how existing benchmark workloads stack up against this criteria. These features motivate us to capture and parametrize TAO's workloads.

P1. Accurately emulates social network requests: The only social network benchmark we are aware of that is derived from production traces is LinkBench [16] from Meta (Table 1). However, its workload is centered around the persistent storage layer, so it excludes the majority of application requests that hit cache. Other benchmarks, such as BigDataBench [46], focus on graph data rather than accesses patterns, so they may not fully capture the extreme skew, high correlation, and read-dominance of these workloads.

Table 1: We compare how the workloads of various benchmarks satisfy the five properties we identify.

Benchmark	P1	P2	P3	P4	P5
TAOBench	✓	r/w-only[2]	✓	✓	✓
AuctionMark [42]	✗	read-write	✗	✗	✗
BigDataBench [46]	✗	none	✗	✗	✓
BG [17]	✗	none	✗	✗	✗
Epinions [1]	✗	none	✗	✗	✗
Graphalytics [23]	✗	none	✗	✗	✓
LDBC [29]	✗	read-write	✗	✗	✓
LinkBench [16]	✓	none	✗	✓	✓
SEATS [43]	✗	read-write	✗	✗	✗
SmallBank [44]	✗	read-write	✗	✗	✗
TPC-C [27]	✗	read-write	✗	✗	✗
Twitter [2]	✗	none	✗	✗	✓
YCSB [26]	✗	none	✗	✓	✗

P2. Captures any transactional requirements: Transactions are a critical part of the social network workload [25, 29]. However, among existing social network benchmarks, only LDBC [29] contains (read-write) transactions. TAO provides one-shot read-only and write-only[2] transactions for improved performance and scalability [25]. These requests constitute a significant portion of the production workload and should be considered explicitly.

P3. Expresses colocation preferences and constraints: Given the rampant growth of social networks [3, 20, 38], sharding is essential to their underlying systems. For those that expose sharding to users [20, 33], data placement is not simply an implementation detail but can be a reflection of user intent, privacy constraints, or regulatory compliance. As a result, data colocation patterns are a significant part of the workload because they can have significant concurrency control and performance consequences [31]. This information is also useful for evaluating the effectiveness of different (possibly implicit) sharding schemes. To the best of our knowledge, no social network benchmark contains derived data on the sharding constraints of the supporting data store (Table 1).

P4. Models request distributions without prescriptive query types: Most existing benchmarks consist of small, fixed sets of query types representing the behavior of specific applications. For example, LDBC's social network benchmark contains 29 query types meant to simulate a social network akin to Meta's [29]. In contrast, many large-scale, real-world platforms have a wide range of applications that exhibit changing request patterns with continued development (TAO's daily workload involves tens of thousands of distinct query types). Our approach captures system behavior through representing workloads in a way agnostic to application "actions" [17] by using probability distributions. As a result, we can evaluate performance without having to enumerate individual queries or replicate their code. These different strategies expose a tradeoff between isolating particular query types for understandability and modeling workloads via distributions for adaptability.

[2]Write-only transactions on TAO include both blind writes and *atomically preconditioned* writes for compare-and-swap functionality (Section 2.2). The latter can depend on data outside the write set.

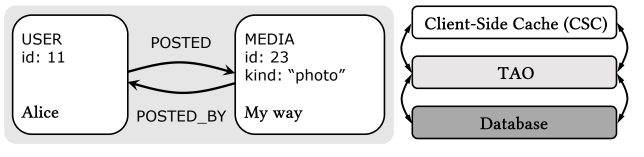


Figure 1: TAO provides online access to the social graph (a) through its tiered architecture (b).

P5. Exhibits multi-tenant behavior on shared data: Most benchmarks only generate the workload of a specific application. In contrast, the distributed databases that support social networks often have multiple tenants with shared data ownership and varying transactional needs. As the LDBC benchmark paper acknowledges [29], running mixed workloads does not necessarily reflect how different applications actually interact on real-world data. In production, we have found that applications can access the same data and consequently affect the behavior of other products. The complex patterns that arise out of these indirect interactions are an important aspect of these workloads. Moreover, product developers often layer reusable infrastructure on top of multi-tenant data stores. This "infra on infra" phenomenon can lead to coordinated behavior between *product groups*, or sets of applications that access the same data or use the same infrastructure. These patterns should be expressed by the workloads of a social network benchmark.

2.2 TAO

In this paper, we present a benchmark that captures the workload observed by TAO [20], the system that provides online access to the social graph at Meta. In aggregate, TAO serves over ten billion reads and tens of millions of writes per second on a changing dataset of many petabytes. Our benchmark workload derived from TAO contains all five properties described above.

Given that this system supports the family of applications used by Meta's over 3.6 billion monthly active users [36], TAO's production workload offers unique insights into the modern social network (**P1**). TAO provides access to objects (nodes) and associations (edges) in the social graph with high availability and at massive scale (Figure 1a). Objects are uniquely identified by an `id` while associations are represented by a `(id1, type, id2)` tuple. TAO's simple API consists of point get, range, and count queries, operations to insert, update, and delete objects and associations, and failure-atomic write (multi-put) transactions. Most product developers access TAO through one of two higher-level query frameworks that makes it easy to express complex operations over the social graph. These frameworks define the semantic boundary between the application logic and the data store. Of these two interfaces, the majority of requests originate from Ent [25], which decomposes complex queries into hundreds of reads and tens of writes to TAO.

While TAO was initially designed to be eventually consistent, it has since added stronger consistency and isolation guarantees to meet the diverse and growing needs of the applications it serves. In addition to read-your-writes (RYW) consistency [39], TAO now provides one-shot write-only transactions with failure atomicity and has prototyped read-only transactions [25]. TAO's write transactions are implemented with two-phase locking (2PL) [18] and a two-phase commit protocol (2PC) [30]. While these requests are less expressive than the interactive, read-write transactions offered in SQL, they represent an important option in the tradeoff between stronger semantics and greater concurrency (**P2**). TAO also supports *atomic preconditions* on writes. Semantically equivalent to compare-and-swap, this API allows an application to issue a write that will commit only if its preconditions (e.g., whether an item exists or is at a particular version) are satisfied.

TAO employs a sharded, tiered architecture to scale both horizontally and vertically (**P3**). It implements two layers of graph-aware caches as part of a three-tiered system with a client-side cache (CSC) and an underlying, statically-sharded MySQL database [35] as shown in Figure 1b. Applications can choose to explicitly colocate data in the MySQL tier.

Finally, TAO supports a wide range of application types as a general-purpose data store. Its simple API has enabled this system to serve as a building block for layering on applications and other infrastructure (**P4**). As a result, TAO's many tenants exhibit differing access patterns on shared data (**P5**). Furthermore, product groups display coordinated behavior that can lead to distinctive trends in request patterns. We are able also to filter for and examine single-tenant workloads (as described in the next section).

3 WORKLOAD CHARACTERIZATION

To build a comprehensive benchmark, we begin by characterizing Meta's social network workload. We describe our data collection process, explain our sampling methodology, and address privacy considerations. We then study key features of TAO's request patterns. For example, over 99.6% of keys that are frequently written to in transactions are read less than once per day on average. Perhaps surprisingly, these keys are not distributed uniformly across shards: over 93.9% of them are colocated with at least one other frequently requested item for one product group we profile. We also identify three types of distinctive transactional access patterns, which result from diverse application needs for atomicity. We report other notable aspects of TAO's workload in the rest of this section, and we use this analysis to identify a set of parameters that can be used to reliably reproduce these workloads.

3.1 Methodology

To characterize TAO's workload, we analyze traces of requests collected over three days. While different days may have varying peak volumes, we find that access distributions do not change significantly between these periods. As daily periodicity is the most prominent pattern in these workloads, we aggregate over multiple days for a representative dataset. To minimize production performance overhead, we uniformly sample traces based on the hash of object `id` or association `(id1, id2)` and record every request to these items. This ensures that we capture all conflicts on these keys to accurately measure contention.[3]

We categorize operation types into three semantic groups: reads, writes, and write transactions. We define *reads* as all types of queries on TAO (point get, range, count), *writes* as all single-key operations that modify data (insert, update, delete), and *write transactions* as

[3]For write-read conflicts, we record every instance of these errors rather than all requests to these data items due to the large volume of reads.

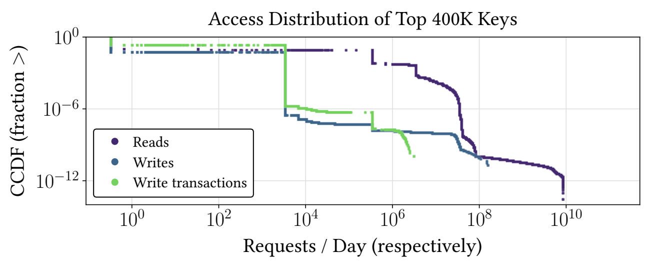


Figure 2: Key access distribution of the top 400K read, write, and write transaction keys. These operation types have widely different access frequencies.

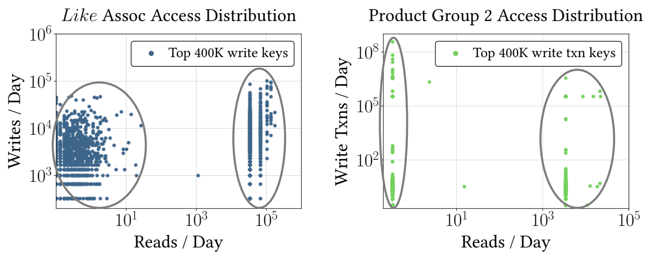


Figure 3: Read and write (transaction) frequencies of the top 400K write (transaction) keys. Read, write, and write transaction hotspots do not always occur on the same keys.

requests using either the single-shard or cross-shard, failure-atomic API [25]. We observe differences in write transactions based on product group, so we further categorize these requests (Section 3.3). The trace we collect is comprised of 99.7% reads, 0.211% writes, and 0.0162% write transactions.

We perform our analysis on aggregated statistics that do not expose individual user information to eliminate any risk of compromising user privacy. Most of our analysis is limited to the top 400K keys of the particular group due to memory constraints of our internal querying service; we find that the tail does not differ significantly beyond 400K keys.

For the graphs in this section, a *key* refers to a unique object or association in the social graph. The *key access* distribution captures the request frequency over a set of keys. The *shard access* distribution captures the request frequency over a set of shards.

3.2 Operation Skew

We summarize differences between operation types in this section. We find that key access distributions, hotspots, and request latency vary between reads, writes, and write transactions. Furthermore, popular keys for reads also vary between tiers.

Key access distributions. *Reads, writes, and write transactions have significantly different key access patterns.* Social graph workloads are known for their extreme access skew [20, 24], but we find this asymmetry manifests differently between operation types. Figure 2 shows that there is significant variation between the key access distributions of the top 400K keys of reads, writes, and write transactions. The P90 access frequency is 3.3M requests per day for reads and 3.3K requests per day for both writes and write transactions. The larger read volume is expected due to the nature of the social network in which users access data more often than they update it (e.g., a user can scroll through many posts before deciding to "like" one). However, the query volume is not the only point of divergence: the distribution tails across operation types are clearly distinct. We explore other differences between these operation types in the rest of this section.

Hotspots. *Read and write (transaction) hotspots do not always align.* Skewed access patterns in social networks are often expressed on different keys across operation types. Specifically, not all keys that are written to regularly are also read frequently. For instance, while a post by a celebrity may be both viewed and liked often (right circle on Figure 3a), data items generated by internal applications (data migration and processing) may be read infrequently (left circle). In fact, over 99.0% of hot write keys are read less than once per day on average for the *Like* association (and are thus excluded from this plot), demonstrating the extent to which skew differs.

A similar pattern emerges for popular keys in write transactions. Figure 3b shows the read and write transaction frequency for Product Group 2. Some items are rarely read (left circle) while others are queried more frequently (right circle). This graph is sparse because over 99.6% of the top 400K write transaction keys are, on average, read less than once per day (so they are not visible on this plot). This high concentration of writes can result from developer dependence on atomicity guarantees. For instance, an application may send concurrent write requests and rely on the system to ensure that only one association of a certain type is created.

Reads across tiers. *Key access distributions for reads vary between tiers.* We study how read frequency correlates across the three layers in our system. The client-side cache (CSC) serves 14.1% of reads, TAO serves 85.0%, and the database (DB) serves the remaining 0.872% of queries. Figure 4 shows how the read frequencies of the top 400K keys of each tier compare in the two other tiers. For example, Figure 4a shows that read frequency is correlated between the CSC and TAO (green points), but the top CSC keys have varying popularity in the DB tier (blue points). For the CSC, 83.5% of its top keys overlap in TAO and 3.84% in the DB. For TAO, 6.24% of its top keys overlap in the CSC and 1.17% in the DB. Finally, 0.330% of the top DB keys overlap in the CSC and 3.31% in TAO.

The two main takeaways are: 1) the caches are clearly effective and important for performance and 2) the caching tiers are not perfect. The first point is apparent through the circled points on Figures 4a, 4b, and 4c. These keys are popular in higher tiers and less frequently requested in lower tiers. Furthermore, many keys do not appear on each graph. For example, only 1.17% of the top 400K TAO keys are also requested in the DB, indicating that the vast majority of queries to these keys are served by the cache.

Even though the caches are effective, access is still skewed in lower tiers, and some hot items are highly requested throughout because of cache eviction or consistency misses. For instance, 83.5% of the top CSC keys are also queried in TAO. Since the CSC does not actively request newer information from lower tiers and relies on short Time-To-Live (TTL) or FlightTracker tickets [39] for cache invalidation, reads that need updated data must go to TAO (or to the DB in rare cases). This trend is apparent through the clustering of points in the upper right corners of Figures 4a, 4b, and 4c. These points represent data items that are highly requested in multiple tiers, indicating that the caches cannot always serve requests to

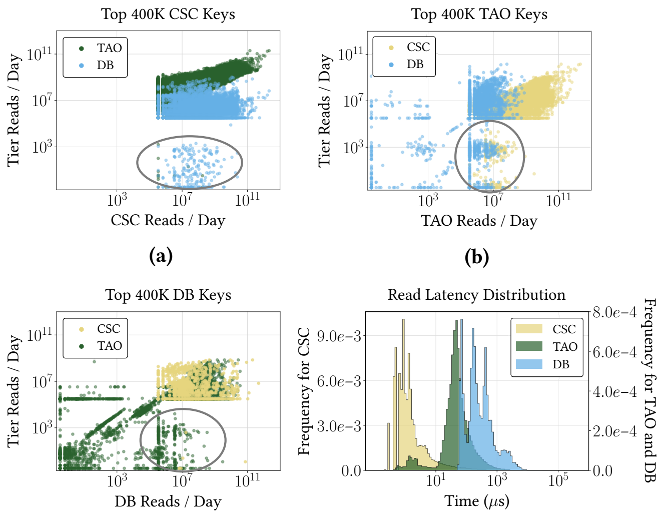


Figure 4: Read frequencies of the top 400K keys and request latencies for each tier. For (a), (b), and (c), the x-axis shows the read frequency of hot keys for a particular tier, and the y-axis shows the read frequencies for the same set of keys in the other tiers. While the caching tiers are effective, key access distribution is still skewed across tiers.

these items. Furthermore, hot keys in lower tiers are not necessarily popular in the higher tiers (shown by the points in the upper left corners of these three graphs). These keys may have aged out of cache or may be updated frequently, forcing reads to go lower tiers for new information.

We observe that there is a tradeoff between proactively invalidating the cache (i.e., doing work at write time) and taking consistency misses (doing work at read time). Figure 4d illustrates that request times in lower tiers are over an order of magnitude higher, due to reading from disk rather than memory and cross-region communication delays. The average latency of reads is 6.0 ms from the CSC, 15.4 ms from TAO, and 76.5 ms from the DB.[4] More effective caching policies to reduce skew and load while balancing consistency considerations could further reduce request times.

Request latency. *Read and write (transaction) latency distributions are bimodal due to cache misses and cross-region requests.* As Figure 5 shows, request latency between read and write operations differs significantly since all writes must go to the primary region database (on average, 13.4 ms for reads, 80.0 ms for writes, and 61.9 ms for write transactions). Writes and write transactions have similar request latencies because transactional use cases are carefully vetted for performance, and most currently involve small write sets. All operation types exhibit long tail latencies due to client-side delays (e.g., network latency) and asynchronous tasks, which are off-peak, analytical jobs that involve more complex queries. Each operation type presents a bimodal latency pattern. Read latency clusters around two peaks: the first corresponds to requests hitting the CSC, while the second captures reads that must go to a local

[4]Note the x-axis of Figure 4d is in log scale, so the long tail of each distribution is not apparent in the graph. For example, the P50 latency is 1.76 ms for the CSC, 1.87 ms for TAO, and 13.7 ms for the DB.

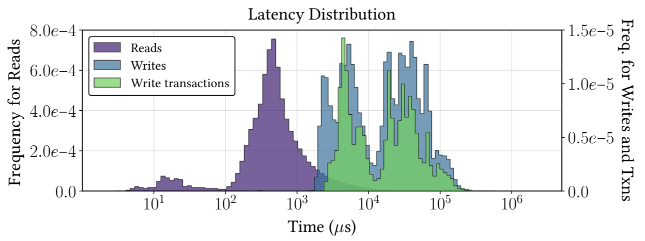


Figure 5: Latency distributions of different operation types. Reads have noticeably shorter latencies than writes, which must go to the database and possibly cross-region.

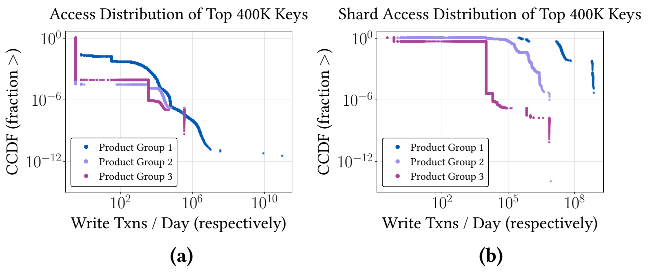


Figure 6: Key and shard access distributions of the top 400K write transaction keys for Product Groups 1, 2, and 3. These distributions vary dramatically, illustrating the need to capture diverse access patterns in a benchmark's workloads.

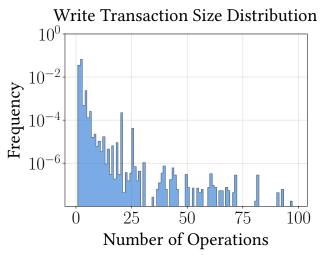
or cross-region instance of TAO or the DB. Writes and write transactions also have bimodal latency distributions but for a different reason. The second peak arises when the primary database for a key is not in the local region, necessitating a cross-region request.

3.3 Write Transactions

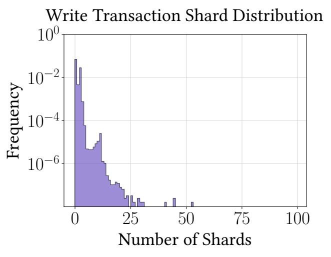
We study write transactions in detail because they represent a challenging but important part of TAO's workload. Specifically, we profile the size, contention, sharding, and skew of the transactions executed by the three main product groups on TAO.

Product groups. *Transactions are highly skewed and vary across applications.* Product groups often build common infrastructure, which may lead to shared request patterns and emergent behavior lower in the stack. Figure 6 shows that the key and shard access distributions differ significantly between the three product groups. Items requested more than 1K times per day make up 67.2%, 3.89%, and 41.4% of the top 400K write transaction keys for Product Groups 1, 2, and 3, respectively. This variation demonstrates that it is important for a distributed database benchmark to have a wide range of workloads to inform the design of multi-tenant, large-scale systems. Specifically, a benchmark should capture how applications interact with each other on parts of the social graph they share (e.g., intra-product group behaviors) as well as the diversity of request patterns that arises with a large number of tenants.

Transaction size. *Write transactions can span many keys and shards.* Currently, more than 0.233% of write transactions on TAO contain 20 or more operations, and over 28.7% involve keys on more than one shard (Figure 7). Most existing large transactions are updates to associations on a single shard. Their inverse associations can be asynchronously updated without needing to undergo



(a)



(b)

Figure 7: Write transaction size and shard distributions. Write transactions can span many keys (a) and shards (b). Many large transactions on TAO currently undergo an optimized protocol [25] to maintain high performance.

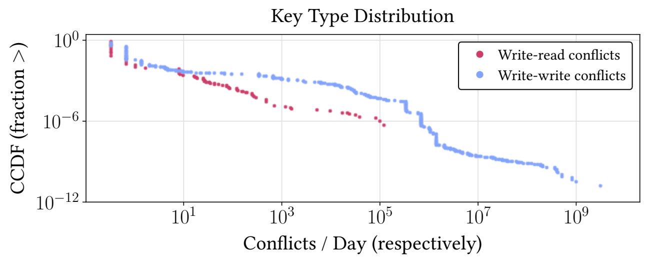


Figure 8: Contention distribution across key type, which corresponds to distinct transaction use cases. Contention varies greatly across use cases, which are carefully vetted to ensure they do not impact other requests.

2PC [25] as long as they do not have preconditions (e.g., uniqueness or version constraints). This optimization ensures that high fan-out transactions can maintain efficient performance and avoid affecting other requests to these keys.

Contention. *Contention varies greatly across key type, which serves as a proxy for different application use cases.* Figure 8 demonstrates how the diversity and skew of the social graph impacts conflicts between transactions. We find that both write-read and write-write conflicts [15] are skewed across key type. Moreover, certain application request patterns can cause high contention. There are no read-write conflicts because all reads are currently non-blocking on TAO.

Altogether, write-read conflicts affect 0.000567% of all read requests and span 82 object and association types. Given the large, read-heavy workload on TAO, even a low contention rate is significant in practice. Write-write conflicts impact 10.1% of all write transaction requests and span 456 types. Since there is no exclusive ownership of data in the social graph, it is essential to manage contention, which may also impact non-transactional requests. As a result, TAO developers proactively investigate and mitigate high contention cases, resulting in low conflict rates on average.

Over 97.3% of write-write conflicts occur as a result of intentionally racing inserts. For example, an application can send redundant requests when creating associations for live video time slices. The application requires only one association for each time slice but wants to ensure timely processing, so it intentionally retries requests, leading to higher contention. While this is a legitimate use of transactions for compare-and-swap functionality, applications are generally expected to keep contention to a minimum to avoid affecting other products. Though the data store ensures transactional guarantees, applications should be jointly responsible for performance isolation on shared data.

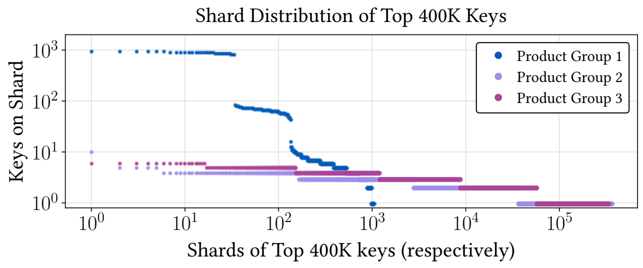


Figure 9: Shard distribution of the top 400K keys of Product Groups 1, 2, and 3. Since many of these keys are colocated on the same shards, a sharding strategy that distributes these items more uniformly would enable better load balancing.

Contention and colocation. *Hot keys in write transactions tend to be colocated on the same shards.* More than 93.9% of hot data items are colocated with at least one other frequently requested item for Product Group 1, 10.0% for Product Group 2, and 17.0% for Product Group 3 (Figure 9). While popular data items are often clustered on a small number of shards, we find that they are rarely part of the same transactions. Hot keys may have been intentionally colocated by application developers to improve the efficiency of batched requests. These results demonstrate a tradeoff between load balancing and the performance of a subset of queries.

Distinct transactional access patterns. *Transactions exhibit distinct request patterns across product groups.* To identify these patterns, we examine the correlation between transaction size and key access frequency for each product group. We find that there are three main transaction types: *TX1* are small transactions (<10 operations) on hot keys, *TX2* involve large transactions on both hot and cold keys, and *TX3* are large transactions that access keys of similar frequency. Product Group 1 contains *TX1* and *TX2*. *TX1* transactions are involved in state updates (e.g., changing the value of an object or edge). For this product group, one use case of *TX2* transactions is to create a large number of new objects and attach them to an existing popular object. Product Group 2 executes *TX1* and *TX3* transactions. A representative *TX3* transaction could atomically update all associations connected to a node. Product Group 3 contains *TX2* and *TX3*. For example, *TX2* transactions are used to log segments of a video clip as it is uploaded. When this process has completed, all the clips (cold keys) can be linked together to a video reference node (hot key), and transactional guarantees prevent duplicate content in the case of retries during media upload.

3.4 Workload Parameters

Our observations above suggest that a small set of parameters is sufficient to fully characterize the social network workload (Table 2). We divide these parameters into two groups: either generalizable to OLTP and graph databases or unique to TAO. Features in the first category include transaction size, sharding strategy, operation type, and data size. We omit a data size distribution graph due to space constraints. TAO-specific features include association

Table 2: We parametrize TAO's workload with this set of features, which we use to create a benchmark.

Parameter	Description
Transaction sizes	Discrete distr. for read- & write-only txns
Sharding	Discrete distr. for objects & associations
Op. types	Proportions for single- & mutli-key reqs.
Request sizes	Discrete distr. of data sizes
Association types	Proportions of association types
Preconditions	Proportions of precondition categories
Read tiers	Proportions of reqs. served by each tier

types, preconditions, and read tiers. Different association types may have varying constraints (e.g., uniqueness, no inverse) that impact request handling. Object types in TAO do not influence system behavior, so we exclude them from our parameters. Write operations and transactions can have atomic preconditions (Section 2.2). Finally, queries can be served from different tiers of the system.

To specify a particular workload, we set parameters in a workload configuration file as discrete probability distributions or modeled after a well-known distribution (e.g., uniform). For example, transaction size is currently specified by an array of sizes and their corresponding weights. We run an offline analysis pipeline periodically to create and refresh workload configurations based on production traces (sampled as described in Section 3.1).

4 BENCHMARK

From our large-scale analysis of TAO's request patterns, we identify a set of key features for social network workloads. In this section, we describe how we leverage these parameters to design a benchmark. TAOBench can accurately reproduce the current TAO workload and is also sufficiently flexible to enable the evaluation of new scenarios.

4.1 Request API

Our benchmark supports the following set of requests, which are mapped to the underlying data store through an adapter layer. We choose to align most of TAOBench's API directly with TAO's. Though this API is intentionally simple, it has enabled engineers to construct complex queries and applications. It also generalizes to the OLTP stores [4] and graph databases [9] underlying many social networks. TAOBench translates TAO's API of point get, range, and count queries, operations to insert, update, and delete objects and associations, and more recently, one-shot read-only and write-only transactions into the following set of requests:

- **`read(key)`**: Read a record
- **`read_txn(keys)`**: Read a group of records atomically
- **`write(key,[preconditions])`**: Write to a record, optionally with a set of preconditions
- **`write_txn(keys,[preconditions])`**: Write to a group of records atomically, optionally with a set of preconditions

While TAOBench does not produce interactive, read-write transactions in its current workloads, we have found restricted transactional semantics to be sufficient for application needs in production. TAOBench can easily support interactive transactions once there are workloads on the benchmark (possibly from other social networks) that contain them. We also leave range and count queries to future work.

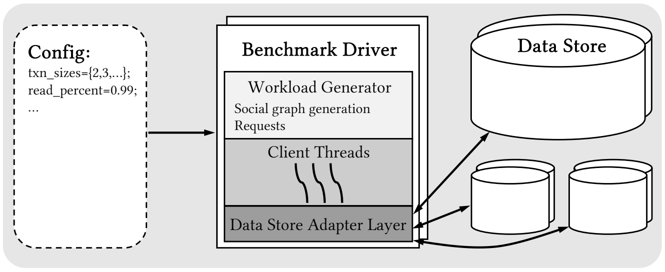


Figure 10: Our benchmark takes in workload configurations and uses them to generate requests to a specified data store.

4.2 Architecture

We describe the distributed, scalable architecture of TAOBench (Figure 10). Based on the workload parameters specified in Section 3.4, our benchmark takes in a workload configuration file containing discrete or piecewise linear probability distributions. TAOBench also takes in several benchmark parameters (duration, target load, warm-up period), which are specific to each run. The benchmark driver, which can be distributed across multiple machines, uses these parameters to generate requests. Each driver creates multiple client threads, which independently execute requests through the data store adapter layer. Each thread measures throughput and latency, and these statistics are aggregated and reported at the end of each run. The benchmark currently produces steady-state workloads, and future work will capture time variation and periodicity.

4.3 Workload Generation

Our benchmark generates workloads in two phases for preparation and execution. The first phase ensures the data store is in a desired initial state (e.g., containing a baseline social graph, having caches warmed up). The second phase produces requests based on the given workload configuration.

In the preparation phase, we generate the baseline social graph that subsequent requests operate on. Since the execution phase randomly draws from these nodes and associations, we need to ensure the starting graph is sufficiently large. Otherwise, we may observe unintended contention as a result of our setup (since the probability of picking the same keys is higher in a small pool). We create all nodes in the baseline graph and preallocate association tuples `(id1, type, id2)`. However, we do not write all associations during the preparation phase to enable testing of uniqueness and other type constraints. We may also choose to warm up the cache during this phase by batch reading certain sets of data items.

In the execution phase, workload generation is a straightforward application of the parameters in Table 2. Transaction size, shard placement, operation type, request size, association type, precondition, and read tier are chosen from their respective distributions. For preconditions that depend on a prior read (e.g., write to an object if it has a previously observed value), the benchmark applies them only on keys that have been queried by each client thread.

Table 3: We open-source three workloads that focus on different aspects of TAO's request patterns.

Workload	Description
T — Transaction	Current transactional workload
A — Application	Speculative transactional workload
O — Overall	Comprehensive TAO workload

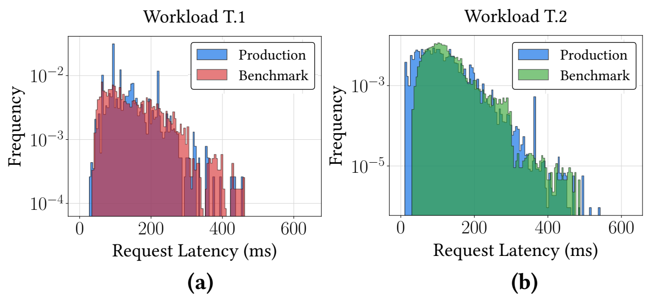


Figure 11: Workloads T.1 and T.2 write transaction latency distributions. Our synthetic workloads have similar distributions compared to those from production.

4.4 Extensibility

We open source Workloads T, A, and O based on production traces (Table 3). Workload T is the ***t***ransactional workload, which represents all requests to keys involved in existing write transactions. Workload A is the ***a***pplication workload that captures requests with transactional intent from the higher-level Ent framework. This workload can be considered a speculative workload because it groups together operations that do not yet use the transactional API in production but may use them in the future. Finally, Workload O is the ***o***verall TAO workload, which is notably read-heavy.

Our benchmark can also be used to model additional workloads. By tuning the set of parameters we identify in Section 3.4, users can experiment with different profiles, reproduce worst-case scenarios, and test new features. We describe Meta use cases in Section 6.

Finally, our benchmark can be easily deployed on different systems. In addition to our internal TAO adapter layer, we currently provide drivers for Cloud Spanner [7], CockroachDB [41], PlanetScale [10], TiDB [32], and YugabyteDB [11] as well as MySQL [5] and PostgreSQL [11]. Since our benchmark API is a subset of most database interfaces, extending to new systems is simple. Most of TAOBench's requests can be translated directly into the corresponding SQL queries. Our data schema mirrors that of TAO, consisting of an objects table and an associations table. We index on `id` for objects and on the `(id1, type, id2)` tuple for associations. For preconditions, we apply a `WHERE` clause for constraints.

5 VALIDATION

We proceed to experimentally validate that TAOBench can accurately downscale production workloads. We compare the performance of several workload configurations and show that our benchmark's generated request patterns have request latencies and contention rates in line with those from production.

Table 4: The limited difference between latency metrics for our generated and production requests across two transactional workloads demonstrates that our benchmark accurately reproduces realistic request patterns.

Workload T.1	Benchmark	Production	% diff.
Mean	155.6 ms	145.9 ms	6.43%
Median (P50)	139.0 ms	130.0 ms	6.69%
P99	389.0 ms	359.8 ms	7.80%
Chi-square ($p = 0.01$)	$Z = 1.99$	$\chi^2_4 = 13.28$	H_0 true
Workload T.2	**Benchmark**	**Production**	**% diff.**
Mean	114.5 ms	103.4 ms	10.2%
Median (P50)	107.0 ms	96.0 ms	10.8%
P99	278.0 ms	274.0 ms	1.45%
Chi-square ($p = 0.01$)	$Z = 1.27$	$\chi^2_5 = 15.09$	H_0 true

Table 5: The contention breakdown for the two workloads is similar for both synthetic and production requests.

Workload T.1	Benchmark	Production	% diff.
Obj. lock conflict	0.138%	0.151%	9.00%
Assoc lock conflict	0.00315%	0.00357%	12.5%
Overall contention	0.141%	0.155%	9.46%
Workload T.2	**Benchmark**	**Production**	**% diff.**
Obj. lock conflict	0.0434%	0.0452%	4.06%
Assoc lock conflict	0.112%	0.114%	1.77%
Unique assoc exists	0.403%	0.443%	9.46%
Overall contention	0.558%	0.602%	7.59%

5.1 Implementation

We implement an internal adapter in C++ for our benchmark to send requests to TAO. Each thread acts as an individual TAO client, mirroring how Meta's applications access this system. Our experimental setup imitates that of production with the following exceptions. First, we send requests to a separate TAO (cache and database) deployment that is smaller than and isolated from the one in production. To simulate cross-region performance, we inject network latency based on empirically determined communication times between Meta data centers; our evaluation below shows that this approximation is sufficient for producing realistic workloads. We also exclude the portion of reads that would hit the client-side cache since we focus on the validation of TAO requests.

5.2 Validation Results

We evaluate two transactional workloads to show that our generated workloads match those in production. Workload T.1 involves transactions that update an object along with an incoming unique association. Workload T.2 focuses on write transactions that update multiple shards. We choose these two workloads because they have different latency and contention profiles, so they provide varying points of comparison.

Table 6: We use TAOBench to find the latency and contention profiles of a new transactional use case.

Workload T.3			
Latency		**Contention**	
Mean	133.1 ms	Obj. lock conflict	0.172%
Median (P50)	107.5 ms	Assoc lock conflict	0.00576%
P99	385.0 ms	Overall contention	0.178%

Request latency. We find that the per operation type latency distributions of our synthetic workload and those from production to be statistically indistinguishable. While TAOBench injects network latencies based on empirical request times between Meta data centers, these latencies are not workload specific. Thus, we can compare the resulting *end-to-end* request latency profiles with those in production to validate our benchmark. Here, we focus on write transaction latency (Figure 11) and omit other operation types due to space constraints since they have similar takeaways. There are limited differences between the key metrics of these distributions (Table 4). In particular, our benchmark is able to accurately reproduce latency tails, which have important performance implications [21]. Finally, the chi-squared goodness-of-fit test confirms that these distributions are not significantly different.

Contention. Next, we verify that the contention rates (the percentage of operations that conflict over a key when attempting to acquire a lock) of our workloads are also statistically indistinguishable from those observed in production (Table 5). Workload T.1 transactions, which access both objects and associations, have similar contention rates for both types of operations. Workload T.2 additionally contains a precondition, which requires the corresponding association to remain unique. We find that transactions abort due to precondition conflict and other types of contention at rates similar to those from production.

6 LESSONS AND EXPERIENCES

In this section, we describe the impact of TAOBench at Meta. Our benchmark has been used to test new features, optimizations, reliability (hotspots, worst-case scenarios, etc.), and speculative workloads since June 2021. We discuss four examples: 1) analyzing new transaction use cases, 2) assessing contention under longer lock hold times, 3) evaluating new APIs, and 4) quantifying the performance of high fan-out transactions. Prior to our benchmark, TAO developers had only a limited stress-testing tool, which could not reproduce realistic workloads or assess new request patterns. We leverage this framework to implement our benchmark, which now allows developers to simulate a wider range of workloads.

6.1 New Transaction Use Cases

Our benchmark enables end-to-end testing early in the development process. Since transactional workloads impact overall write availability (2PL blocks other writes), it is important for applications to thoroughly evaluate the effects of adopting transactions. Before our benchmark, engineers could only run a small portion of these requests in the final pre-production stage. At this phase, any issues, such as contention hotspots, could delay rollouts.

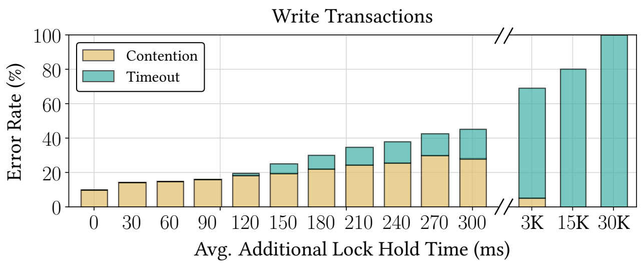


Figure 12: Write transaction error rate over lock hold time. Contention (lock conflict) increases as transactions occupy locks for longer periods of time. After a certain threshold, more and more write transactions timeout.

A Meta application team used our benchmark to evaluate the impact of adding a new transaction use case (Workload T.3), which creates an object and adds an association. While Workload T.3 has a similar request structure to Workload T.1 from Section 5 (though the former creates objects and associations rather than updating them), our benchmark shows that the latency of Workload T.3 is lower than that of Workload T.1 (Tables 4 and 6). On the other hand, Workload T.3 has 26.2% more lock conflicts compared to Workload T.1. Subsequently, a limited rollout to internal test users revealed the same breakdown of errors, demonstrating that our benchmark is able to successfully anticipate production issues.

6.2 Contention

In another use case, engineers at Meta wanted to understand how TAO would perform if locks were held for extended periods of time (e.g., due to network delays, regional overload, or disaster recovery). Using our benchmark, they were able to quickly evaluate a representative workload. TAO's performance is shown in Figure 12: longer locking periods lead to higher conflict rates because greater request times increase the probability that writes contend over the same keys. Beyond a certain threshold, transactions time out before they can acquire locks. Given that most web requests complete in under 30 seconds, slower transactions have decreasing utility to the application. Thus, failing fast is preferable so that occupied resources can freed for other purposes. This example demonstrates that we can use TAOBench to assess different design choices under varying scenarios, including ones that are challenging or infeasible to assess in production.

6.3 New APIs

In order to optimize for application concurrency, TAO provides the ability to atomically check a set of preconditions on writes (semantically equivalent to compare-and-swap). Applications typically read data items, execute some business logic, and (possibly much later) write with the precondition that the data items have not changed. This request pattern imitates optimistic transactions, except that the initial reads may be executed by other code paths or products. The chance of failure for the write increases as the latency between the read and subsequent write grows. The asynchronous nature, geo-distribution, and shared data ownership of the system can exacerbate this issue by introducing longer delays and races between these operations.

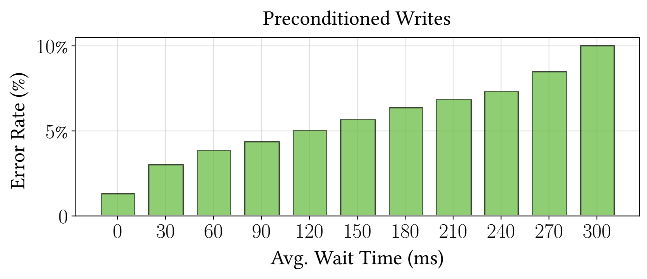
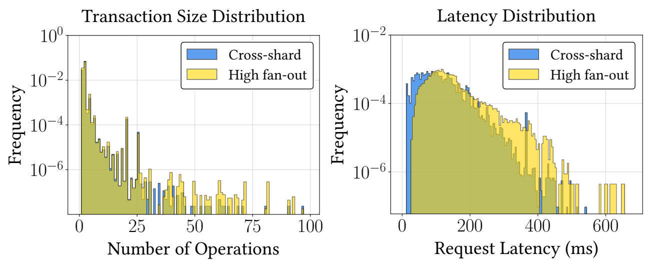


Figure 13: Write error rate over wait time. The error rate of writes preconditioned on a previous read increases as the time between these two operations grows.



(a) **(b)**

Figure 14: Transaction size and latency distributions. We find that the high fan-out write transaction workload spans many keys (a), which can lead to increased latency (b).

We evaluate the impact of increasing latency between a read and a preconditioned write with our benchmark by measuring the write failure rate. Using our benchmark, we inject wait time between these two operations for a generated workload based on existing single-shard transactions with preconditions. We find that the error rate doubles if we increase the average wait time from 30 ms to 150 ms (Figure 13). Given that a cross-region request can take hundreds of milliseconds to complete, higher latency between these operations can significantly lower the success rate of preconditioned writes. These errors can become more problematic as additional cross-shard use cases are adopted and the write rate increases.

Adding a new operation on TAO could reduce error rates by enabling the read and preconditioned write to be completed in the primary region database. For example, TAO currently supports a counter increment API (within the object update operation) that allows the read and subsequent write to be completed by the TAO writer in the primary region of the key rather than by a remote TAO client or writer. This decreases the likelihood that the read will return stale information, enabling higher success rates for the increment. Our benchmark will enable engineers to explore specific use cases in detail and measure the impact of adding new APIs.

6.4 High Fan-Out Transactions

Finally, we describe how TAOBench is used to evaluate high fan-out transactions. A significant portion of applications that seek to adopt transactions want to opt-in existing groups of writes that could benefit from atomicity guarantees. These operations are currently encapsulated in Ent changesets [25], which represent write transaction boundaries TAO developers are considering supporting in the long-term.[5] As Figure 14a shows, some of these transactions touch many keys, which may lead to slower requests and higher conflict rates. To help teams at Meta assess the impact of moving to the transactional API, we quantify the performance impact of high fan-out transactions using our benchmark.

Table 7: High fan-out transactions have greater latency and higher contention that the existing cross-shard transaction workload. While expected, these results suggest the need for further optimizations to efficiently support larger requests.

Latency	Cross-shard	High fan-out	% diff.
Mean	103.4 ms	128.5 ms	21.6%
Median (P50)	96.0 ms	119.0 ms	21.4%
P99	274.0 ms	363.0 ms	27.9%
Contention	**Cross-shard**	**High fan-out**	**% diff.**
Obj. lock conflict	0.0452%	0.0591%	26.7%
Assoc lock conflict	0.114%	0.101%	12.1%
Unique assoc exists	0.443%	0.547%	21.0%
Overall contention	0.602%	0.707%	16.0%

We find that there is a significant increase in latency (Table 7), especially tail latency, for the high fan-out workload compared to current cross-shard transactions (Figure 14b). There is also an increase in contention errors. While unsurprising, these results demonstrate that high fan-out transactions will be more challenging to support and are inline with past work showing that large transactions can lead to significant performance degradation [31]. The ability to test this high fan-out workload will allow engineers to evaluate future optimizations in application use cases, concurrency control mechanisms, and implementation strategies.

7 DISTRIBUTED DATABASE EVALUATION

In this section, we present TAOBench results for five widely used distributed database services: Cloud Spanner [7], CockroachDB (CRDB) [41], PlanetScale [10], TiDB [32], and YugabyteDB [12]. Specifically, we evaluate Workload A (write transaction-heavy, application workload) and Workload O (read-heavy, overall workload). We omit Workload T due to space constraints.

Our goal is not to provide comparisons of these databases nor to draw any conclusion about their relative performance. Instead, we aim to illustrate that TAOBench provides effective workloads to evaluate these systems. Our benchmark enables researchers and engineers to explore tradeoffs on social network request patterns and can be used to guide the development of these systems.

System details. The five systems we evaluate are geo-distributed SQL databases. Spanner supports its own SQL query language while CRDB and YugabyteDB are compatible with PostgreSQL. PlanetScale and TiDB support MySQL. In terms of architecture, CRDB and TiDB run separate compute and storage tiers. Spanner implements distributed SQL layers above its transactional key-value store; YugabyteDB does so above a distributed document store. PlanetScale supports sharded MySQL instances with varying isolation levels within a shard and Read Committed across shards.

[5]This workload is captured by the write transactions in Workload A (Section 4.4).

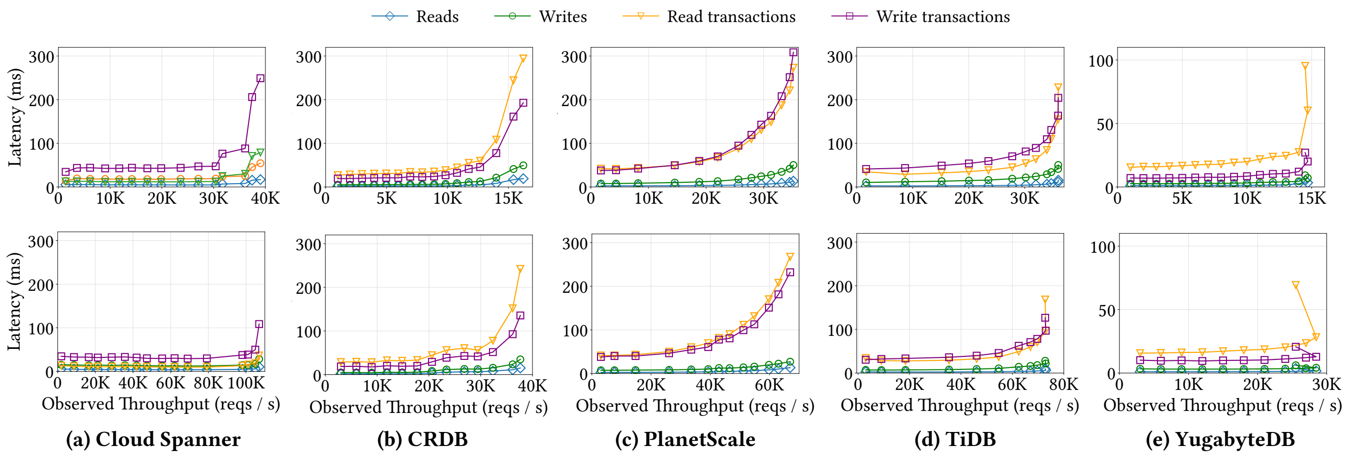


Figure 15: Performance of Workload A (top row) and O (bottom row) for the five databases. We observe significant differences between the two workloads. Note that the axes are different and that the systems are evaluated under varying conditions, so they are *not* directly comparable.

For replication, Spanner uses Paxos [34] while the other systems except PlanetScale use Raft [37]; PlanetScale uses MySQL semisynchronous replication. For concurrency control, CRDB, PlanetScale, and YugabyteDB leverage variations of multi-version concurrency control (MVCC). CRDB supports Serializability, and YugabyteDB supports Snapshot Isolation (SI) and Serializability. Spanner relies on synchronized clocks to generate monotonically increasing timestamps to ensure Strict Serializability for its transactions. TiDB implements optimistic and pessimistic locking protocols to provide Read Committed, Repeatable Read, and SI.

7.1 Experimental Setup

We evaluate our benchmark against similarly-sized, hosted cloud clusters for the five systems. All clusters except Spanner have 48 cores. Since the number of cores per Spanner machine is not publicly available, we use a six node cluster of a similar price point with 12 TB of total storage. We use a three node cluster for CRDB with 18 TB of total storage. Our three node YugabyteDB cluster has 2.4 TB of total storage and provides SI. Our PlanetScale cluster has eight shards with four cores each, and the remaining cores are allocated to gateways nodes for load balancing; each shard provides Serializability. For TiDB, we run three TiDB nodes and three TiKV nodes with SI. While each database can be geo-distributed, we use clusters that are replicated in three availability zones within one region, where the client machine also resides. We received extensive assistance from CRDB, PlanetScale, TiDB, and YugabyteDB engineers in this benchmarking effort. For Spanner, we followed publicly available "best practice" guides [8] when tuning the database. We launch our benchmark on a separate 64 core machine, with clients running in a closed-loop. During experiments, we observe that the CPU of our client machine stays below 15%, indicating that the benchmarking client is never the bottleneck.

7.2 Results

Our experimental results illustrate the benefits of using TAOBench to highlight system performance on different social network workloads (Figure 15). Throughput and latency for the five systems vary across Workloads A and O, with higher performance on the latter due to its read-heavy nature. As expected, latency rises as we increase target throughput. While in all cases this growth can be attributed to the system reaching high CPU utilization (>90%) and becoming overloaded with requests, latency degradation varies for the five databases due to varying design decisions and implementation tradeoffs. For example, Spanner shows nearly no change in latency until high load, while PlanetScale exhibits gradually increasing latency as load grows. We also find that different TAOBench workloads can elucidate performance differences on the same system. For example, PlanetScale has higher write transaction latency than read transaction latency on Workload A but the opposite on Workload O. While expanding on the reasons behind these differences is beyond the scope of this paper, our results demonstrate that TAOBench can be useful for evaluating the performance impact of various systems tradeoffs and guiding future optimizations.

7.3 System Impact

As a direct result of running our benchmark, we were able to assist several of these databases in identifying issues and performance improvement opportunities. For PlanetScale, we found that we were unable to run `INSERT INTO ... SELECT` queries, which are used to enforce uniqueness constraints on associations. Due to our benchmarking effort, PlanetScale prioritized support for these requests, and this functionality is now generally available.

TAOBench was also able to reveal bugs and optimization opportunities for YugabyteDB. When we initially ran our benchmark on this system, we found that performance was unexpectedly low. We flagged this anomalous behavior for their engineers, who then profiled the system during TAOBench runs and discovered a performance bottleneck: a Postgres monitoring extension using exclusive locks [13]. This example illustrates how TAOBench's ability to reproduce workloads on demand enables detailed investigations.

We also helped YugabyteDB identify an optimization for scans, which are called during the benchmark preparation phase. We noticed these queries were unusually slow and sometimes led to out-of-memory errors. Using our benchmark, YugabyteDB engineers found that their system did not push down filters on scans to

Postgres but instead materialized all rows from the relevant table into memory. The subsequent fix [14] resulted in substantial performance and memory overhead improvements for these queries.

8 RELATED WORK

Social network benchmarks. There are a number of existing social network benchmarks. LinkBench [16], which derives its workload from requests of a single MySQL instance that underlies TAO at Meta, is the only benchmark we are of aware of that is based on production traces. However, it lacks graph level transactions, does not provide data colocation information, and only captures a small subset of the full social graph workload.

A range of other benchmarks are based on production storage data (e.g., graph characteristics) rather than request traces. BigDataBench [46] uses a small subset (fewer than 5K nodes and 90K associations) of the Meta social graph to generate a workload for graph analytics benchmarking. The Epinions benchmark [1] in OLTP-Bench is derived from data of a consumer review website and focuses on a single application. The Twitter benchmark [2] from the same testbed is based on a snapshot of the micro-blogging site from 2009. It lacks transactions as well as colocation information.

LDBC [29] has developed a synthetic social network benchmark that targets graph databases. This benchmark focuses on complex, processing-intensive queries rather than serving workloads. Graphalytics [23] is a benchmark with six graph processing algorithms that uses synthetic data from LDBC. Terevinto et al. [45] and Cao et al. [22] describe social network benchmarks that are not open-source. As we detail in Section 2.1, none of these benchmark workloads satisfy all five of the crucial properties needed for a comprehensive evaluation tool.

OLTP benchmarks. A plethora of benchmarks are available for evaluating OLTP workloads. The TPC-C benchmark [27], designed to represent a wholesale supplier's transaction processing requirements, is the industry standard for measuring OLTP system performance. The Yahoo! Cloud Serving Benchmark (YCSB) [26] is a popular microbenchmarking framework developed to measure the performance and scalability of cloud serving systems. Its workloads represent simple key-value store applications and involve individual reads and writes as well as range scans. Other common transactional benchmarks include AuctionMark [42], SEATS [43], and SmallBank [44]. OLTP-Bench [28] presents a unified framework that includes 15 different workloads. While there is overlap between OLTP and social network workloads, the skew, high correlation, and read-heavy of the latter motivates the need for a new domain-specific benchmark.

9 CONCLUSION

In this work, we present TAOBench, a new benchmark that generates workloads modeled on a real-world social network. We characterize the request patterns of the social graph data store at Meta, identify key parameters to design and construct a benchmark, and validate that this benchmark accurately simulates production request patterns. We report on how TAOBench is used internally to evaluate new features, optimizations, and reliability. We also present benchmarking results from five distributed databases to demonstrate that TAOBench can be used to investigate system tradeoffs and address performance issues. While current workloads focus on Meta's social graph (one of our key contributions is to make these workloads accessible to the general research community), our simple model can be easily adapted to other systems. TAOBench's goal is to serve as a broad platform for social network evaluation, and we encourage other social networks to contribute workloads to this open-source framework.

ACKNOWLEDGMENTS

We thank Matt Pugh, Tushar Pankaj, and the VLDB reviewers for their insightful feedback; Emmanuel Geay, Evan Liang, Neil Chao, Eric Kuecks, Soham Shah, Amin Saeidi, Anthony Simpson, Zhuo (Gordon) Huang, Rahul Tekawade, and Scott Pruett for their contributions to the data and systems in this paper. We are also grateful to Andy Pavlo for connecting us to individuals at companies of the various databases we evaluate. Audrey Cheng is supported in part by the NSF CISE Expeditions Award CCF-1730628, a Facebook Next-Generation Data Infrastructure Research Award, and gifts from Alibaba Group, Amazon Web Services, Ant Group, CapitalOne, Ericsson, Meta, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

REFERENCES

[1] 2013. Epinions.com Benchmark in OLTP-Bench. https://github.com/oltpbenchmark/oltpbench/tree/master/src/com/oltpbenchmark/benchmarks/epinions/
[2] 2013. Twitter Benchmark in OLTP-Bench. https://github.com/oltpbenchmark/oltpbench/tree/master/src/com/oltpbenchmark/benchmarks/twitter/
[3] 2014. Manhattan, our real-time, multi-tenant distributed database for Twitter scale. https://blog.twitter.com/engineering/en_us/a/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale.html
[4] 2015. HBaseCon 2015 General Session: Zen - A Graph Data Model on HBase. https://www.slideshare.net/HBaseCon/keynote-3-pinterest-49043320
[5] 2020. MySQL Transactional and Locking Statements. https://dev.mysql.com/doc/refman/8.0/en/sql-transactional-statements.html
[6] 2022. Apache HBase. https://hbase.apache.org/
[7] 2022. Cloud Spanner. https://cloud.google.com/spanner
[8] 2022. Cloud Spanner best practices. https://cloud.google.com/spanner/docs/best-practice-list
[9] 2022. DGraph. https://github.com/dgraph-io/dgraph
[10] 2022. PlanetScale. https://planetscale.com/
[11] 2022. TAOBench. https://github.com/audreyccheng/taobench
[12] 2022. Yugabyte DB. https://www.yugabyte.com
[13] 2022. YugabyteDB Postgres Monitoring Issue. https://github.com/yugabyte/yugabyte-db/issues/10805
[14] 2022. YugabyteDB Row Comparison Issue. https://github.com/yugabyte/yugabyte-db/issues/11463
[15] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized Isolation Level Definitions. (2000), 67–78.
[16] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.
[17] Sumita Barahmand and Shahram Ghandeharizadeh. 2013. BG: A Benchmark to Evaluate Interactive Social Networking Actions. In *CIDR*. Citeseer.
[18] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.
[19] ByteDance Official Tech Blog. 2020. Design and Implementation of ByteDance's Trillion-Edge, 10M+ QPS Graph Database and Computation System. https://blog.csdn.net/ByteDanceTech/article/details/104509642
[20] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC '13)*. 49–60.
[21] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. 2020. Gryff: Unifying Consensus and Shared Registers. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. 591–617.
[22] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In

18th USENIX Conference on File and Storage Technologies (FAST '20). 209–223.

[23] Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. 2015. Graphalytics: A Big Data Benchmark for Graph-Processing Platforms. In *Proceedings of the GRADES'15* (Melbourne, VIC, Australia) *(GRADES'15)*. Association for Computing Machinery, New York, NY, USA, Article 7.

[24] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. *Proceedings of the International AAAI Conference on Web and Social Media* 4, 1 (May 2010), 10–17.

[25] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. 2021. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3014–3027.

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[27] The Transaction Processing Performance Council. 2010. TPC-C. http://www.tpc.org/tpcc/

[28] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment* 7, 4, 277–288.

[29] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network benchmark: Interactive Qorkload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[30] J. N. Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmüller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 393–481.

[31] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (jan 2017), 553–564.

[32] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: A Raft-Based HTAP Database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[33] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[34] Leslie Lamport et al. 2001. Paxos Made Simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[35] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12, 3217–3230.

[36] Meta. 2022. Meta Reports First Quarter 2022 Results. https://investor.fb.com/investor-news/press-release-details/2022/Meta-Reports-First-Quarter-2022-Results/default.aspx

[37] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC '14)*. 305–319.

[38] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jgadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. 2013. On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1135–1146.

[39] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. 2020. FlightTracker: Consistency across Read-Optimized Online Stores at Facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 407–423.

[40] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. 2012. Serving Large-Scale Batch Computed Data with Project Voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'12)*. USENIX Association, USA, 18.

[41] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509.

[42] The H-Store team. 2013. AuctionMark: An OLTP Benchmark for Shared-Nothing Database Management Systems. https://hstore.cs.brown.edu/projects/auctionmark/

[43] The H-Store team. 2013. SEATS Benchmark. https://hstore.cs.brown.edu/projects/seats/

[44] The H-Store team. 2013. SmallBank Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/

[45] Pablo Nicolas Terevinto, Miguel Pérez-Francisco, Josep Domenech, José A. Gil, and Ana Pont. 2016. Benchmarking Online Social Networks. *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)* (2016), 164–169.

[46] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. 2014. BigDataBench: a Big Data Benchmark Suite from Internet Services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 488–499.

[47] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. 2017. PaxosStore: High-Availability Storage Made Practical in WeChat. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1730–1741.