Fair Transaction Processing for Multi-Tenant Databases

Audrey Cheng
UC Berkeley
accheng@berkeley.edu

Aaron Kabcenell
Meta
akabcenell@meta.com

Xiao Shi
Unaffliated
xiao.shi@aya.yale.edu

Jolene Huey
UC Berkeley
jolenehuey@berkeley.edu

Peter Bailis
Stanford, Workday
pbailis@cs.stanford.edu

Natacha Crooks
UC Berkeley
ncrooks@berkeley.edu

Ion Stoica
UC Berkeley
istoica@berkeley.edu

ABSTRACT

Multi-tenant transactional databases frequently observe contention on shared data, leading to a need for performance isolation. Databases typically provide this via per-tenant request rate limit or quota, but this approach can lead to system underutilization. Fair sharing has been widely applied in other domains to achieve both performance isolation and high utilization. In this paper, we address the problem of fair sharing for transactions, which present new challenges because they do not acquire resources all at once. We propose the first fair transaction scheduling algorithm, DRFT, that ensures both the share guarantee and strategy-proofness by accurately accounting for transactional resource usage. We evaluate DRFT on a range of standard benchmarks and real-world workloads, showing that it ensures fairness with less than a 5% throughput overhead compared to state-of-art scheduling policies.

PVLDB Reference Format:
Audrey Cheng, Aaron Kabcenell, Xiao Shi, Jolene Huey, Peter Bailis, Natacha Crooks, Ion Stoica. Fair Transaction Processing for Multi-Tenant Databases. PVLDB, 18(8): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

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

1 INTRODUCTION

With the rapid growth of data-intensive applications and cloud-based services [76], databases must increasingly support multi-tenant workloads, especially ones that share data. For instance, modern data lakes now support both transactional and analytical access by different tenants to the *same* tables [14, 48, 65]. Furthermore, multi-tenant shared databases, such as Meta's social graph data store TAO [15] and Databricks SQL [29], serve a variety of applications (e.g., Facebook, Instagram) that operate on shared data. As a result, contention between the transactions of different applications affects both individual tenant and overall system performance.

A key concern of multi-tenant systems is *performance isolation*—each tenant should be guaranteed some portion of the resource regardless of the demand of other tenants. This issue is of particular concern to applications that share data. For instance, a client (tenant) can compromise performance isolation by sending a disproportionate number of transactions that access a "hot" (popular) data item, decreasing the throughput of all the other clients accessing the same data. Our interviews with several companies that operate multi-tenant databases confirm that interference due to transactional contention is a prominent issue in practice (Section 2).

A popular strategy to achieve performance isolation in databases is to impose a request rate limit (e.g., Cloudflare rate limiting [24]) or resource quota per client (e.g., Amazon RDS quotas [10]). While this approach provides some protection against cross-application interference, it can lead to system underutilization: even if there is only one client sending requests to certain data items, it will still be restricted by its rate limit, resulting in low overall throughput.

To address this issue, *fair sharing* has been widely applied to achieve performance isolation in other domains, such as networking [38] and job processing [39]. The core tenet of fair sharing is to guarantee that each client its "fair share" of resources (i.e., it is guaranteed some portion of its desired resource). Any unused portion of a client's share is dynamically made available to others, leading to higher resource utilization. In the transactional setting, we consider data items to be the *logical* resources that the database must manage contention over (e.g., in contrast to physical resources, such as CPU, disk, memory). These items are often exclusively allocated (e.g., under two-phase locking [13], only one transaction can hold the write lock at a time). Moreover, transactions access multiple data items in the same request. Given this model, we aim to equalize *item usage*, or the time that a transaction makes an item inaccessible to other requests. To provide fairness across multiple resources, we take as a starting point Dominant Resource Fairness (DRF) [39], which ensures that each client gets an equal share of its dominant (i.e., most used) resource over time.

While promising, adopting fair sharing for transactions requires addressing two challenges: (i) incremental resource acquisition and (ii) the tension between fairness and high throughput. First, most prior fair schedulers assume that all resources are acquired at the beginning of a request (e.g., sufficient CPU and memory are simultaneously needed for a compute task). In contrast, transactions acquire resources dynamically throughout execution (e.g., under two-phase locking [13], locks are usually acquired piecewise as the transaction proceeds), leading to potential resource wastage. For instance, if a transaction T accesses item x and later needs to access y, but y is being locked by another transaction, then T will hold the lock on x while it waits for y to become available. This transaction wastes resource x without making progress and

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:XX.XX/XXX.XX

blocks other transactions waiting on x. This usage should be taken into account by a fair scheduler. Second, the system must navigate between achieving fairness and maximizing throughput. If several transactions need to be scheduled and one of the requests can run without conflicting with the current workload, the scheduler would increase throughput by allowing it to execute first, even though this may unfairly increase service for its client.

In this work, we introduce the Dominant Resource Fair Transaction (DRFT) scheduler to provide fair sharing for transactions. We describe how DRFT tackles each of these challenges in turn to ensure both performance isolation and high throughput.

First, DRFT addresses the challenge of incremental resource acquisition by accurately accounting for how transactions can delay subsequent requests in the schedule. The catch-22 in the transactional setting is that a client's workload dictates the resulting contention, so item usage depends on the order in which *future* requests are executed. To handle this, DRFT first charges each transaction, independently of other requests, a *baseline* usage that is based on the concurrency control protocol. It then *retroactively* updates each transaction's *schedule-dependent* usage after subsequent conflicting requests have been scheduled.

Second, DRFT introduces a Δ parameter to enable system operators to smoothly trade off between fairness and high throughput. On one hand, we want to equalize usage without delay to achieve fairness as fast as possible. On the other hand, we aim to increase system throughput by executing transactions in schedules that maximize concurrency. Therefore, if clients can tolerate delays between periods of service, the system can achieve higher throughput. As such, we make this delay configurable via our Δ parameter, which bounds how much usage can differ between clients.

We implement DRFT on RocksDB and evaluate this policy on a range of standard benchmarks and OLTP workloads. DRFT matches the performance of a state-of-the-art transaction scheduling policy (less than 5% difference in throughput) while ensuring fairness. Furthermore, we demonstrate through extensive experiments that DRFT provides performance isolation, in contrast to other baseline methods. Finally, our algorithm imposes minimal overheads, showing less than a 4% drop in throughput on low contention workloads.

In summary, we make the following three contributions:

- We characterize the problem of fair scheduling for transactions, which differs from traditional fair sharing in that resource acquisition occurs incrementally.
- We propose and provide analysis on the first multi-resource fair scheduling algorithm for transactions, DRFT, which provides the standard fairness properties: the share guarantee and strategy-proofness.
- We evaluate DRFT on a range of workloads and confirm our algorithm provides fair sharing without significantly harming overall database throughput.

2 MOTIVATION

As multi-tenant database deployments become increasingly common, these systems must carefully manage contention on shared data among diverse application requests—a challenge widely acknowledged in industry [20]. In this section, we present case studies drawn from several production systems on performance interference due to logical contention. These scenarios often involve common application functionality and are hard to avoid. Despite their prevalence, most existing systems rely on static per-tenant request rate limits or quotas, which can lead to severe system underutilization (up to a 2.0× throughput disparity as we show in Section 7.3).

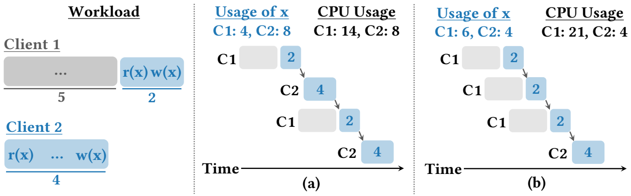


Figure 1: Logical contention differs from physical contention.

2.1 Unfairness in Practice

In this paper, we consider the problem of performance isolation on logical resources (i.e. data items). Prior work has largely focused on the isolation of physical resources (e.g., CPU, disk, memory) [44, 51, 64]. However, even if a client is guaranteed a share of CPU or bandwidth, its isolation can still be compromised due to transactional contention on data items. Figure 1 highlights how contention can prevent logical resource isolation and how this differs from physical resource contention. Two clients send transactions to a database that uses two-phase locking (2PL) [13] to ensure serializability. For simplicity, we assume each operation takes one unit of time. Furthermore, we assume both clients send requests at equal rates and are backlogged (i.e., at least one of its requests is queued) on item x. C1 sends longer transactions, acquiring read locks on five non-conflicting items (represented by the gray box) before holding the lock on item x while C2 sends shorter transactions that hold the lock on x for four time units. Since both clients contend on x, access time (e.g., lock hold time) on this item is the performance bottleneck. Thus, for performance isolation on logical resources, each client should have equal access time on contended items.

However, standard policies and techniques do not execute requests in a manner that equalizes access time. Databases typically execute transactions in first-in, first-out (FIFO) order. Under FIFO in Figure 1a, C1's and C2's requests would execute consecutively, leading to C1 having less access time on x. A physical resource scheduler would also de-prioritize C1's requests. From a physical resource standpoint, C1 is using more (CPU, I/O, etc.) resources since it sends more operations per transaction, so fewer of its requests should execute. From the logical resource perspective, C1's requests actually hold x's lock for less time than that of C2's. Thus, we should execute two C1 transactions for every C2 transaction to equalize access time on x for performance isolation.

Case studies. Our conversations with engineers at Databricks, Meta, and Neo4j reveal that performance interference due to logical contention is a pervasive and growing issue for real-world systems. This problem is exacerbated by increasingly diverse workloads (e.g., cloud traffic [76]) as well as the convergence of transactional and analytical processing on shared data (e.g., Snowflake's Unistore [65] provides hybrid tables to serve both OLAP and OLTP requests). Logical contention has led to significant incidents in production, ranging from end user delays to database outages [20]. Broadly, we classify problematic access patterns into two categories: large transactions and bursty workloads.

(1) Large transactions. A frequent source of logical contention between tenants is large transactions on shared data. At Databricks, a data analytics platform, data purging tasks, which usually involve many reads and writes that touch large portions of data, are one such example. These requests must be transactional to avoid inconsistent results while the deletions occur: a task locking a range of data to decide what to delete would slow down progress of many concurrent requests trying to access the same data. Engineers often attempt to mitigate this issue by moving larger requests to the background, but this can result in starvation, forcing application developers to manually monitor and intervene in these tasks.

Large transactions also appear in other use cases. Data migration tasks at Databricks, which acquire exclusive access to items for extended periods of time, can unfairly stall concurrent user requests accessing the same data. Furthermore, *mammoth* transactions (i.e., large read-write requests that are especially prevalent on graph data [21]) present further examples of performance interference. For instance, pattern matching queries (e.g., large-scale authorization queries) observed at Neo4j [2] often conflict with shorter transactions. Balancing larger transactions with the rest of the workload without starvation or underutilizing system resources is a significant challenge in practice.

(2) Bursty workloads. Bursty workloads are also problematic for data contention. At Meta, many product teams operate on the same social graph data (e.g., user profiles) stored in TAO [15]. Unexpected high traffic periods (e.g., caused by viral videos, posts by celebrities, etc.) by one application can severely impact others. A recurring pattern observed at Meta and Databricks is that application developers often fail to foresee future contention issues when initially designing the data model of a product. For example, a new social media application can quickly gain traction, leading this application to send a high volume of requests that conflict with the existing workload. While some systems provide mechanisms to manage cross-application interference (e.g., TAO's higher-level query language [20]), building bespoke infrastructure to ensure scalability in the face of contention is expensive. As a result, many engineers start with "ad-hoc" implementations [69] of transactions that lead to contention bottlenecks as products grow. At this point, developers need to manually intervene and redesign parts of the application, which can have significant engineering and monetary costs. These examples highlight the need for a dynamic mechanism to fairly serve transactions from different tenants.

2.2 The Status Quo

Despite the prevalence of unfairness due to logical contention, no database solutions, to the best of our knowledge, provide both performance isolation and high throughput. Most production systems simply impose a request rate limit or enforce a quota to minimize the impact of worst-case behavior. For instance, TAO [15] deprioritizes requests from high QPS clients once they exceed a pre-specified threshold. However, these static mechanisms can lead to system underutilization—if a client does not use its reservation, no one else can, so the system cannot run at full capacity. Furthermore, request rate limits/quotas often require careful tuning and manual intervention when client requests exceed these thresholds (e.g., companies carefully allocate quotas among various applications on internal infrastructure [23]). Indeed, we find that the problem of logical contention is undercharacterized in industry: there are no formal metrics to measure unfairness of item access, though application developers often observe the effects of this issue through unexpected drops in throughput and/or increases in latency.

Towards this end, our goal in this paper is to provide a mechanism to deliver both performance isolation and high throughput in the face of transactional contention. Specifically, we aim to ensure that each client is guaranteed a minimum share of resources, regardless of the demand of other clients. Furthermore, a client should not be able to increase its service at the detriment of others. Finally, the system should be able to balance between prioritizing individual client and overall system performance. We achieve these goals via a trusted scheduler, which we expand upon next.

3 FAIRNESS FOR TRANSACTIONS

To address the challenges caused by logical contention, we start with fair sharing, a well-known mechanism used in other domains to provide both performance isolation and high utilization [41]. This technique ensures that each client is guaranteed a "fair" share of resources regardless of the demand of other clients (for performance isolation) and allows these resources to be otherwise distributed when the client is not active (for high utilization). In this section, we first describe the properties needed in a fair transaction scheduler in the context of a single resource, before extending them to the multi-resource setting with Dominant Resource Fairness. We then explain the challenges in providing a fair scheduler for transactions.

3.1 A Fair Scheduler for One Resource

When client transactions conflict over shared data, one client can cause others to receive unfair service. To see this, we revisit the workload in Figure 1 assuming 2PL. We consider two scenarios to illustrate what properties a fair scheduler should provide.

Share guarantee. First, we consider how logical resources should be shared among clients. We denote each item to be a logical resource, equivalent to how CPU and memory are treated as physical resources [39, 55]. Subsequently, we use the term *item usage (IU)* to refer to the period of time during which an item is inaccessible by other requests due to execution constraints (we describe how to calculate IU for different concurrency control protocols in Section 4). Figure 1a shows the item usage of x for two clients under FIFO. Since the clients send requests at equal rates, their requests execute roughly in consecutive order, leading to C1 having less usage of x.

Intuitively, it would be "fair" to give each client equal access time on x (e.g., C1 executes at twice the rate of C2). This requirement is captured by the *share guarantee*, which ensures that each of n clients should be guaranteed at least $\frac{1}{n}$ fraction of the resource, regardless of the demand of other clients [55]. This guarantee provides performance isolation: C1 would be ensured an appropriate amount of item usage on x, regardless of C2's workload.

Strategy-proofness. Another concern for a fair scheduler is its vulnerability to *manipulation*: clients can exploit the scheduler to receive better service by artificially inflating their demand for resources they do not need (i.e., adding extraneous operations and/or transactions). Figure 1b shows how FIFO can be manipulated: if C1

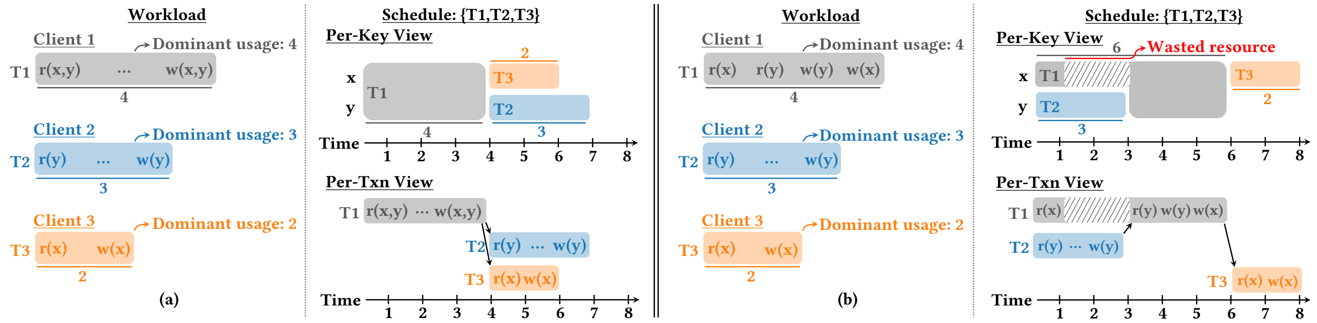


Figure 2: Two workloads under 2PL. DRF is insufficient for transactions, which do not access all items at once. In (b), T1 holds the lock on x (due to its conflict with T2) without making any progress, causing additional delays for T3. /* accheng: check grayscale of colors */

increases its demand by sending three times as many requests as C2, C1 receives disproportionately more item usage on x. A fair scheduler should be resistant to such manipulation, which incentivizes clients to waste resources and lead to lower overall throughput.

Specifically, the scheduler should ensure *strategy-proofness*: a client should not be able to finish faster by lying about its resource demands. While malicious clients are less likely in databases (with proper authorization and security measures), unintentional scheduler manipulation can still occur. For instance, one client may send many retries for transactions involving hot items, causing feedback loops that impact the throughput of other clients [46]. Furthermore, database clients usually monitor only their own applications, so they are unaware about how their workload affects others.

A fair scheduler. When there is only one contended item, ensuring both the share guarantee and strategy-proofness is simple: the scheduler should *equalize item usage over time* across client transactions. For the workload in Figure 1, a fair scheduler would schedule two C1 requests for every C2 request to provide equal item usage of x. Even if C1 sends more requests than C2, it cannot manipulate the scheduler as long as the scheduler tracks the item usage of each client. As a result, the fair scheduler can ensure performance isolation between clients while maintaining high utilization.

3.2 Scheduling Over Multiple Resources

In practice, most transactions require more than one resource (i.e., item) at a time. The canonical solution to ensure fairness in the presence of multiple resources is *dominant resource fairness* (DRF) [39]. DRF equalizes each client's *dominant share*, which is the maximum share that the client has been allocated of any resource. By definition, a client is bottlenecked by the dominant resource since it uses this resource the most. Thus, performance depends only on the dominant share—other resources are irrelevant for fairness because they are not the bottleneck. In our setting, each item is an exclusive resource (e.g., only one transaction can write to an item at a time), so the dominant share of a transaction is its maximum item usage. DRF ensures both the share guarantee (each client is guaranteed at least some share of its dominant resource) and strategy-proofness (artificially increasing the demand for any resource other than the dominant one has no impact on this client's service).

Crucially, DRF assumes that each request acquires all resources it needs before starting execution. For instance, a compute job cannot begin until sufficient CPU and memory are simultaneously available. However, this assumption does not hold in our setting. Transactions acquire resources *incrementally* as a function of program order and isolation level guarantees. For example, under 2PL [13], transactions do not grab all locks before execution but acquire locks gradually. As a result, we cannot apply DRF directly to transactions. In the rest of this section, we show several examples to first provide intuition for DRF before demonstrating why it fails for transactions.

DRF examples. To illustrate how DRF applies to multiple resources, we consider the example workload in Figure 2a. First, we assume transactions grab exclusive locks for all items they access simultaneously. Thus, the usage of each item in a given transaction is the same (and trivially equal to the dominant usage). For instance, T1's usage for x and for y is four time units. Similarly, T2 has a dominant usage of three units on y, and T3 has two units on x. As both the per-item and per-transaction views of the schedule show, T1 occupies x and y for four time units, conflicting with both T2 and T3. DRF would equalize each client's dominant usage, which is based on the *individual* item usages of each transaction.

Next, we show how DRF can be unfair for standard transactions that access items incrementally. In Figure 2b, transactions obtain locks as execution proceeds. Here, T1's operations on y begin after its $r(x)$. When T1 and T2 start simultaneously in this example, T2 occupies the lock on y for three units, so T1 must wait until T2 completes before it can proceed. Consequently, T1 occupies the lock on x for six units of time due to its conflict with T2 on y. For two of these six units, T1 holds the lock on x without making any progress, wasting this resource and decreasing system throughput. However, DRF assumes that T1's dominant usage on x is four units of time because it only considers transactions in isolation. Thus, DRF underestimates T1's usage, leading to unfair service.

3.3 Challenges in Fair Transaction Scheduling

As the previous example shows, DRF fails because it does not account for incremental resource acquisition in transactions. The main implication of this behavior is that item usage can vary as a function of the schedule. In the remainder of this section, we highlight the challenges in scheduling transactions fairly and describe a performance tradeoff we must navigate.

Share guarantee. This property requires dominant item usage to be equalized across clients. Since item usage depends on the schedule, we cannot determine it only based on an individual transaction's operations but also must consider its conflicts with other requests. Specifically, we need to account for how subsequent transactions conflict with and are delayed by a given transaction. However, we do not know which requests will be scheduled in the future in most databases. Accordingly, we need a way to *retroactively* account for item usage to ensure fairness between clients.

Strategy-proofness. To guarantee that a client cannot increase its dominant resource usage by lying about its demand (i.e., adding extraneous operations and/or transactions), a fair scheduler should carefully account for how each client's requests contribute to resource wastage. Specifically, a client should not be able to deliberately cause conflicts that increase the item usage of other clients. This is unaddressed by previous schedulers [38, 39, 55] because they assume all resources are acquired before request execution.

Tradeoff between fairness and throughput. Achieving fairness can cause overall system throughput to drop, and the database must handle this tradeoff. At a high level, a scheduler prioritizing fairness can constrain execution order, while higher throughput is possible with more flexibility to order transactions in ways that reduce conflicts. This tradeoff is especially pronounced in transactional workloads for which the schedule can greatly impact performance (e.g., up to 3.9x based on prior work [18]). On one hand, we want to equalize usage across clients as fast as possible to ensure fairness without delay. This is typically achieved through *memoryless scheduling*, where a client's current share of resources should be independent of its share in the past. This approach avoids penalizing clients that were active in the past and reduces workload burstiness to provide fairness quickly. On the other hand, *maximizing system throughput* is essential for high-performance database systems. However, schedules that maximize concurrency to improve throughput can delay equalizing usage between clients. An effective scheduling policy should enable system administrators to balance between these goals depending on workload needs.

4 ACCOUNTING FOR ITEM USAGE

To ensure fairness, we first need to accurately account for each transaction's item usage. Since usage depends not only on a transaction's own operations but also its conflicts with other requests, we measure it in two parts: (i) *baseline usage* (BU), which is known by the time a transaction finishes execution and is determined based on the concurrency control protocol and (ii) *schedule-dependent usage* (SDU), which reflects how conflicts from *subsequent* transactions may increase a transaction's usage beyond its baseline.

Transactions, schedules, and time. We consider a *transaction* to be a set of read and write operations on items with ordering constraints. Traditionally, database literature discusses transaction *history* as the post-facto execution order [6, 12]. We instead define the transaction *schedule*, following prior work [18], to be a partially ordered set constrained by (i) operation dependencies within a transaction and (ii) operation dependencies across transactions, which are the result of enforcing a given level of isolation via a specific concurrency control protocol during execution. This enables us to quantify how long schedules take to execute, which we measure by applying the notion of *virtual time* from the networking literature [83]. Virtual time is an abstraction that measures logical work performed by the system. For simplicity, we assume in this section that each operation on an item takes one unit of virtual time (multiple reads can occur during the same time unit). We explain how we measure real time in Section 6. /* accheng: mention serializability here? */

Baseline usage. We define baseline usage (BU) as the minimal possible usage of a given item by a transaction based on the concurrency control protocol, regardless of the schedule. Specifically,

Definition 1. **Baseline usage $\mathrm{BU}_T(x)$.** For item x in transaction T, the virtual time interval during which the concurrency control protocol disallows any other transaction from writing to x.

Concurrency control protocols traditionally disallow writes at two moments: (i) during the physical act of modifying an item and (ii) when operations to that item by *another* transaction violate serializability. For pessimistic protocols, BU begins with lock acquisition and ends when locks are released, representing the time during which the lock is occupied. For optimistic protocols, we model BU as the time during which another transaction can cause an abort to capture the effects of contention. While in the former we assign BU to the transaction occupying the lock and in the latter we assign it to the transaction that is aborted due to contention, this asymmetry enables us to remain consistent in charging higher BU for transactions with greater potential to conflict. We give examples for several popular protocols below:

- In 2PL [13], transactions acquire locks as they read/write items and release those locks after committing. Thus, BU starts from when the lock is acquired to when the transaction commits.
- In OCC [54], BU encompasses the period between a write to an item and the end of the validation phase (operations from other transactions are not allowed since they would cause aborts).
- In MVTSO [12], reads are non-blocking, and operations from other transactions with higher timestamps can proceed immediately after writes. BU is one unit of time for these operations. For transactions that read and write to the same item in a single request, MVTSO would abort other transactions accessing this item between these operations, so this duration represents BU. In contrast, MV2PL acquires locks that are held for the duration of the entire transaction, so BU would end at commit time.

As a concrete example of BU, consider the workload in Figure 3ii assuming 2PL. Here, $\mathrm{BU}_{T1}(x)$ is the time interval $[1-2]$ since $T1$ holds the exclusive lock on x for this period while $\mathrm{BU}_{T1}(y)$ is [2]. /* accheng: check example */ /* natacha: the time unit 2 */

Schedule-dependent usage. In contrast to baseline usage, which is determined solely by the concurrency control protocol, we define schedule-dependent usage (SDU) to account for additional delays that result from a given schedule. We calculate the time interval for which each subsequent conflicting operation is delayed and take the union of these intervals to obtain the SDU. SDU captures the effects of data dependencies across multiple items. Precisely calculating it requires care: overestimating or underestimating SDU can misrepresent a transaction's usage and violate the share guarantee.

First, we identify the relevant data dependencies in the schedule. Namely, for a given item x of a given transaction T, we determine the set of transactions scheduled after T that contain operations which conflict on x (i.e., two transactions conflict when they access the same item and at least one of them writes to the item [6]). We term $\hat{T}$'s access to x as a *succeeding conflicting operation (SCO)* if $\hat{T}$ is ordered after T in the schedule and conflicts on x. Note that there can be multiple SCOs for x as schedules are partially ordered (e.g., if x is a write, there can be several read SCOs). In effect, the data dependency between T's operation on x and its SCOs blocks

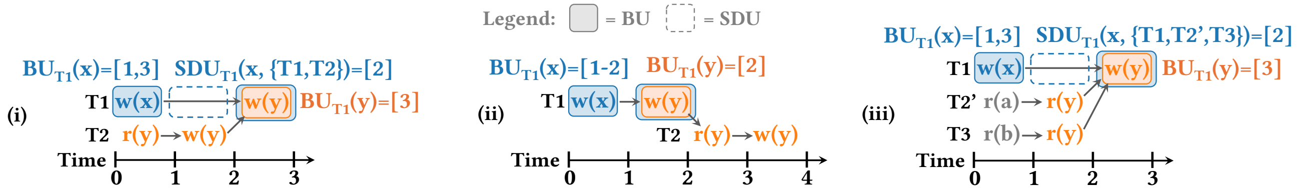

Legend: = BU = SDU
(i)
BU_{T1}(x)=[1,3]
SDU_{T1}(x, {T1,T2})=[2]
T1 w(x)
w(y)
BU_{T1}(y)=[3]
T2 r(y)→w(y)
Time
0 1 2 3
(ii)
BU_{T1}(x)=[1-2]
BU_{T1}(y)=[2]
T1 w(x)→w(y)
T2 r(y)→w(y)
Time
0 1 2 3 4
(iii)
BU_{T1}(x)=[1,3]
SDU_{T1}(x, {T1,T2',T3})=[2]
T1 w(x)
w(y)
BU_{T1}(y)=[3]
T2' r(a)→r(y)
T3 r(b)→r(y)
Time
0 1 2 3

Figure 3: BU and SDU for T1 under 2PL for different schedules.

the SCOs from executing until T completes its operation.

Next, we define several intermediate notions that quantify the delay imposed on each SCO:

- *Transaction start time* (T_{start}): the virtual time at which transaction T begins execution in the schedule.
- *Ideal operation time* (o_{itime}): the earliest time at which operation o can begin execution when considering only a transaction's individual operations, calculated as T_{start} plus duration of operations that precede o in T. For example, if an operation o is the fifth operation in T and T starts at virtual time three, then o_{itime} is virtual time eight.
- *Actual operation time* (o_{atime}): the virtual time at which o actually begins execution when accounting for execution constraints that arise in the schedule.
- *Operation delay set* ($delay_o$): the time for which SCO is delayed, calculated as the time interval $[o_{itime} - o_{atime-1}]$ if $o_{itime} < o_{atime}$ and an empty interval otherwise.

Finally, we define SDU based on the delays of all SCOs:

Definition 2. **Schedule-dependent usage SDU$_T(x, S)$.** For item x in transaction T under schedule S, SDU equals:

- For pessimistic protocols, the union of each SCO's $delay_o$ (i.e., $\bigcup_{o \in \mathrm{SCO}} delay_o$) of subsequent items in T
- For optimistic ones, the union of each SCO's $delay_o$ on x

We note that SDU is calculated only based on the transactions succeeding T that *directly conflict* with T's operations (this follows from the definition of SCO). The benefit of this approach is that we do not need to maintain the history of transactional conflicts—once the set of SCOs is scheduled, we only need to calculate SDU once for this item, enabling us to minimize memory overheads. Furthermore, we take the union across SCOs to avoid double-counting: this ensures we do not overcharge for the usage of a given request. We describe how we account for SDU among clients in Section 5.

Figure 3 provides several examples of SDU calculations. Each subfigure shows BU (solid boxes) and SDU (dashed boxes) for $T1$ in different schedules under 2PL. In (i), we have $\mathrm{BU}_{T1}(x)$=$[1, 3]$ since the transaction needs to hold the lock on x until commit. We have $\mathrm{SDU}_{T1}(x, \{T1, T2\})$=[2] since T1's progress is delayed by T2 when waiting for the lock on y (o_{itime} for y is at time unit two but o_{atime} is at time unit three). As a result, it holds the lock on x for an additional time unit. Finally, $\mathrm{BU}_{T1}(y)$=[3] since T1 needs to write to y before commit. In (ii), there is no SDU for T1 since o_{itime} is equal to o_{atime} for all operations. In (iii), both T2' and T3 cause T1 to be delayed, leading to $\mathrm{SDU}_{T1}(x, \{T1, T2', T3\})$=[2].

Item usage. Taking BU and SDU together, we define item usage to represent the delay imposed by a transaction on a given item:

Definition 3. **Item usage IU$_T(x, S)$.** For item x in transaction T under schedule S, the virtual time interval from the union of $\mathrm{BU}_T(x)$ and $\mathrm{SDU}_T(x, S)$ (i.e., $BU_T(x) \cup SDU_T(x, S)$).

Our definition of item usage captures the difference between applying fairness for physical resources and for transactions. Specifically, with SDU, we account for how acquiring resources as a request executes (rather than all at the start) impacts other requests, enabling us to quantify any resource wastage. We leverage our notion of item usage in the next section to design a fair scheduler.

5 ACHIEVING FAIRNESS

We now present our fair scheduler for transactions, DRFT, which ensures both the share guarantee and strategy-proofness to achieve performance isolation with high throughput. DRFT schedules requests based on client virtual times, which are calculated based on dominant item usages. DRFT first accounts for the baseline usage and retroactively adds schedule-dependent usage once subsequent conflicting transactions are scheduled. Furthermore, DRFT navigates the tradeoff between memoryless scheduling and improving throughput by providing a tunable Δ parameter. We first introduce memoryless DRFT before describing the complete algorithm.

5.1 Memoryless DRFT

In this section, we provide an overview of virtual time fair schedulers before describing how memoryless DRFT adapts virtual time to schedule transactions in a way that quickly achieves fairness.

Background: virtual time fair schedulers. Virtual time schedulers are a well-established mechanism to ensure fairness on resources that are shared over time (i.e., exclusively accessed resources) [31, 41, 55]. At a high level, the scheduler computes a virtual time per client and uses this value to determine which client's requests to execute next. Consider for example n clients that share one exclusive resource, and assume each client request acquires this resource for one unit of time. The scheduler increments each client's virtual time as each request executes. To ensure the share guarantee, the scheduler equalizes virtual time across clients (i.e., choose the client with the lowest virtual time to run next). The scheduler guarantees strategy-proofness by accounting for the virtual time of the dominant resource of each request.

Memoryless scheduling. Memoryless DRFT is a virtual time fair scheduler for transactions that achieves fairness with minimal delay by quickly equalizing virtual times between clients. For each transaction, the scheduler assigns a virtual start time upon its arrival and later uses this value to calculate a virtual finish time. The finish time is then used to determine the client's virtual time. Accordingly, we schedule at the granularity of transactions (not operations) and account for resource usage after each transaction has executed. For memoryless scheduling, the scheduler picks the client with the lowest virtual time to be scheduled next: this ensures that usage is equalized as quickly as possible across clients. We also ensure that a client is not penalized for using more resources in the past when fewer client were active to prevent starvation [55].

We adapt standard definitions and assumptions from the fair sharing literature [38]. We assume a set of n clients that share a

Table 1: Main notation for DRFT algorithm.

Notation	Description
$T_{c,j}$	j^{th} transaction of client c
$a_{c,j}$	Arrival time of transaction $T_{c,j}$
$b_{c,j,k}$	Baseline usage of $T_{c,j}$ on item k
$d_{c,j,k}$	Schedule-dependent usage of $T_{c,j}$ on item k
$V(c,t)$	Virtual time of client c at time t
$S(T_{c,j})$	Virtual start time of transaction $T_{c,j}$
$F(T_{c,j})$	Virtual finish time of transaction $T_{c,j}$

set of m items k, $(1 \leq k \leq m)$. A client c issues a sequence of transactions $T_{c,1}, T_{c,2}, T_{c,3}, \ldots$. The client is *backlogged* during a time interval if it has at least one outstanding request queued at any time. We assume *work-conservation*: as long as there are requests in the queue, the system is not idle. Let $T_{c,j}$ be the j^{th} transaction of client c. Upon $T_{c,j}$'s arrival, the scheduler will assign a virtual start time $S(T_{c,j})$ and finish time $F(T_{c,j})$ to this request. The client's virtual time at real time t is given by $V(c,t)$. For simplicity, we assume each operation takes one unit of time and explain how we measure real time in Section 6. We also support *weighted workloads*: if a client c is given weight w_c, it takes $\frac{1}{w_c}$ virtual time units to execute each of its operations [31]. Weighted workloads are useful in the transactional context because certain applications may have business-critical tasks that require higher priority.

Transaction virtual start time. Under memoryless scheduling, a client's current usage should not depend on its past usage to avoid starving existing clients when new ones enter the system. We calculate virtual times accordingly. Specifically, we determine a transaction's virtual start time based on whether the client that issues the transaction is backlogged (i.e., active). For a non-backlogged client, the start time of its transaction is based on the virtual time of the request currently being executed by the system [41]. As a result, this newly active client will not receive preferential service just because it was not active in the past. For a backlogged client, a transaction's virtual start time is simply the virtual finish time of the previous transaction of this client (we discuss how to compute finish time next) since this request must wait on preceding requests from this client to complete. Formally, the virtual start of transaction $T_{c,j}$ is calculated as:

$$S(T_{c,j}) = \max\Big(A(a_{c,j}), F(T_{c,j-1})\Big)$$

where $a_{c,j}$ is the real arrival time of the transaction and $A(a_{c,j})$ is the virtual start time of the transaction currently being executed by the system at real time $a_{c,j}$. Using real arrival time is a standard choice for fair schedulers [38, 41].

Transaction virtual finish time. A transaction's virtual finish time is determined by its item usage. To ensure fairness, we equalize usage of *dominant* resources (i.e., the resource that each transaction requires the most time on) across clients. Since the dominant resource can change across transactions, we calculate it on a per-request basis, inline with prior work [38]. The primary challenge in our setting is that item usage (and thus, the dominant resource) can change as a function of the schedule. To deal with this, we first account for the *baseline* usage and then retroactively update its *schedule-dependent* usage (potentially multiple times) as subsequent transactions are scheduled. Accordingly, the finish time is determined after transaction execution. Once each transaction has executed, we first calculate the baseline usage, denoted $b_{c,j,k} = |BU_{T_{c,j}}(k)|$, for each item that transaction $T_{c,j}$ accesses. As conflicting transactions are scheduled after transaction $T_{c,j}$ in the schedule H, we calculate the schedule-dependent usage, denoted $d_{c,j,k} = |IU_{T_{c,j}}(k,H)| - b_{c,j,k}$, for each conflicting item.

$$F(T_{c,j}) = S(T_{c,j}) + \max_k \left\{ b_{c,j,k} + \frac{d_{c,j,k}}{l_k} \right\} \times \frac{1}{w_c}$$

where l_k is the number of transactions that conflict on item k.

After the dominant resource is determined for this transaction, we also add $\frac{d_{c,j,k}}{l_k}$ to the virtual times of its conflicting transactions (i.e., requests corresponding to the SCOs of this item). By distributing SDU across all involved requests, DRFT ensures strategy-proofness: even if a client tries to add extraneous operations/transactions to increase the usage of other clients, it will not be better off because it would observe an identical increase in virtual time. We note that since SDU cannot be computed until after conflicting transactions are scheduled, there can be a delay until the usage of a given transaction is fully accounted for. We discuss why this delay is unavoidable and its implications at the end of this section.

Client virtual time. Finally, we define a client's virtual time, denoted $V(c,t)$ for client c as the virtual finish time of its most recently committed transaction at real time t. Formally,

$$V(c,t) = F(\arg\max_j (C(T_{c,j}) < t))$$

where $C(T_{c,j})$ returns the commit time of transaction $T_{c,j}$.

DRFT uses client virtual times to decide which request to schedule: it picks the client with the lowest virtual time to execute next to keep usages across clients as equal as possible.

Scheduling based on virtual time. We discuss the implications of using virtual time in DRFT. The primary advantage of how we compute this value (following prior work [38, 41]) is that the system does *not* need to know the operations of a transaction a priori. The virtual start time of transaction $T_{c,j}$ depends only on the start times of the currently executing transactions and the finish time of the previous transaction, $F(T_{c,j-1})$, of this client. This makes DRFT suitable for interactive transactions (i.e., data accesses are not known beforehand), which are the default in most databases.

The retroactive nature of SDU calculation means that a client's virtual time may not immediately reflect the final usage of some transactions and may be adjusted multiple times as subsequent conflicting requests are scheduled. This delay in fully accounting for usage is unavoidable: unless we know precisely which transactions will arrive and be scheduled in the future, we cannot calculate SDU a priori. While we cannot achieve instantaneous fairness, we ensure the share guarantee as quickly as possible with DRFT.

5.2 Generalized DRFT

While memoryless scheduling achieves fairness quickly, it often does not result in schedules that maximize throughput. To balance between memoryless scheduling and improving overall utilization, we now introduce the full version of DRFT. This refinement enables system operators to control the degree to which we can delay fairness through a Δ parameter. Such customization is important in practice to cater to diverse workload requirements (Section 2).

Algorithm 1: DRFT /* accheng: check this */

```
1  Data structures
2  Δ: delta parameter
3  txn_sched: schedule of transactions
4  client_item_usages: item usages, per client
5
6  procedure SCHED_NEXT_TXN():
7      if GET_MAX_VT_DIFF() ≤ Δ then
8          txn_sched.add(SMF_SCHEDULER())
9      else
10         txn_sched.add(GET_TXN_WITH_MIN_VT())
11
12 procedure UPDATE_BU(txn : transaction, c : client):
13     // Update baseline usage of this txn
14     s_time ← GET_START_TIME(c)
15     f_time ← GET_FINISH_TIME(s_time, txn)
16     client_item_usages[c] = f_time
17
18 procedure UPDATE_SDU(txn : transaction, c : client):
19     // Update usages of txns earlier in the schedule
20     conflict_items ←
        GET_CONFLICT_ITEMS(txn, txn_sched)
21     for k ∈ conflict_items do
22         prev_txn, s_time, client_j ← GET_PREV_TXN(k)
23         sdu ← GET_SDU(prev_txn, txn, txn_sched)
24         client_item_usages[client_j].update_ft(sdu)
25         client_item_usages[c].update_ft(sdu)
26
27 procedure COMMIT_TXN(txn : transaction, c : client):
28     UPDATE_BU(txn, c)
29     UPDATE_SDU(txn, c)
30     success ← DB_COMMIT(ts)
31     return success
```

The Δ parameter bounds the maximum allowable divergence between virtual times of active clients in the system. Specifically, the virtual time of any client can never differ by more than Δ:

$$\max_{c \neq c'} \{V(c) - V(c')\} \leq \Delta$$

This grants the system flexibility in choosing what to schedule while ensuring that client virtual times will be eventually equalized: the scheduler can defer serving a client with the lowest virtual time if scheduling another client's transaction would lead to better throughput. A larger Δ means clients may need to tolerate longer delays between periods of service.

As an example, we consider Figure 4, which shows two schedules from the workload in Figure 2b. We assume the current virtual times of clients C1, C2, and C3 are 0, 1, and 2, respectively. If $\Delta = 0$, then DRFT must schedule T1 first since C1 has the lowest virtual time, leading to a schedule that executes in eight units of time (Figure 4a). However, if $\Delta > 2$, then DRFT can run T3 before T1, leading to a schedule that finishes in only six units of time (Figure 4b). While C1's transaction executes later in Figure 4b, C1 eventually gets its fair share, and the system observes higher throughput since the same number of transactions execute in a shorter period of time.

To select transactions that enhance throughput, DRFT integrates the Shortest Makespan First (SMF) algorithm [18], a state-of-the-art scheduling policy that minimizes the delays caused by conflicts for fast schedules. DRFT uses SMF to schedule transactions unless there is a client with a virtual time Δ less than any other client. In this case, this client's transaction is automatically scheduled next to ensure bounded fairness. We note that memoryless DRFT is a special case of the full policy when $\Delta = 0$ since this algorithm always chooses the client with the smallest virtual time.

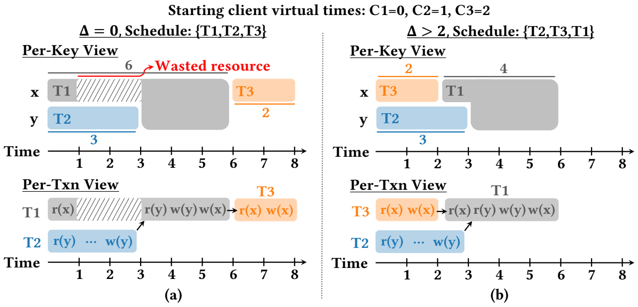


Figure 4: Two schedules of the workload in Figure 2b. The schedule in (b) provides higher throughput compared to (a).

Setting Δ. Determining an appropriate Δ value depends on how much a system administrator prioritizes overall system performance compared to client tail latencies. A higher Δ typically leads to higher throughput but also increased latencies since clients can be delayed for longer periods of time. In general, setting Δ depends mainly on the average transaction duration (e.g., if requests take 10ms on average, then Δ=100ms means that DRFT has the flexibility to choose among roughly 10 different clients at a time). We find that tuning Delta to allow the system to choose among five or more clients enables us to attain high throughput (Section 7.2).

DRFT walkthrough. We provide an overview of the transaction lifecycle under DRFT in Algorithm 1. When a transaction arrives in the system, it is queued to be scheduled. To pick the next transaction to execute (line 6), DRFT checks if there exists a client with a virtual time Δ less than any other client (line 7). If so, this client's transaction is automatically scheduled next (line 10). Otherwise, DRFT uses SMF to choose the next transaction to run (line 8). After each transaction executes, we update usages at commit time (line 27). Specifically, we update the baseline usage of this transaction (line 12) and the schedule-dependent usages of transactions earlier in the schedule that conflict with this transaction (line 18).

5.3 Fairness Properties

We present the formal properties (share guarantee and strategy-proofness) provided by DRFT. We also discuss the bounds on fairness under our policy. Due to space constraints, we defer the proofs to the extended version of our paper [8].

Theorem 1. *DRFT provides the share guarantee (a backlogged client should receive an equal share, i.e., item usage, of one of the resources it uses, regardless of the demand of other clients.).*

The share guarantee is essential for performance isolation. DRFT provides it for transactions, ensuring that every client makes progress.

Theorem 2. *DRFT ensures strategy-proofness (a client cannot increase its dominant resource usage by lying about its demand, i.e., by adding extraneous operations and/or transactions).*

Strategy-proofness prevents clients from "gaming" the scheduler [39]. As detailed in Section 5.1, DRFT achieves this by distributing SDU across all transactions involved in a conflict. As a result,

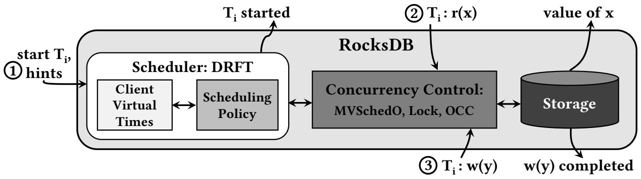


Figure 5: Overview of DRFT implementation in RocksDB.

any increased item usage a client attempts to cause via conflicts also impacts its own virtual time. Thus, it is impossible for a client to improve its own performance at the detriment of others.

DRFT also provides theoretical bounds on (i) the maximum difference in usage between any two backlogged clients and (ii) the maximum delay a client can observe. These bounds are standard for fair schedulers in other domains [38] and prevent starvation. We present them in the extended version of our paper [8]. /* accheng: do we want the tput bound? */

5.4 Discussion

We discuss the implications of applying fairness in the transactional setting for two performance properties.

Sharing incentive. In prior work, fair schedulers typically provide (via the share guarantee) *sharing incentive*: each client is better off sharing the resource rather than taking turns exclusively using the entire resource. For example, each of n clients should receive at least as much service as if it were the only client sending requests to the resource for $\frac{1}{n}$ of the time. However, guaranteeing the sharing incentive is not always possible in the transactional setting because some schedules lead to worse overall throughput due to conflicts. For example, Figure 4a shows a schedule in which C1's transaction is delayed due to its conflict with another client's request, leading to worse performance than if C1 had exclusive access to x and y for one-third of the time. If client workloads are arbitrarily delayed, we can control what share each client will have, so we can asymptotically achieve the sharing incentive with DRFT as Δ goes to infinity. More generally, DRFT's Δ parameter enables system administrators to determine how much delay clients can tolerate to achieve a guaranteed level of service.

Performance predictability. As is true in prior work [39, 55], ensuring fairness does not automatically provide predictable performance, which is often a desirable goal for multi-tenant systems (e.g., database SLAs [22]). To understand why, we consider a simple example: if the current number of clients are receiving a guaranteed level of service and the system has no additional capacity, we cannot admit another client into the system without impacting the performance of existing clients. For predictability, we need not only the share guarantee but also *admission control* (which we discuss further in Section 8) to limit the number of clients in the system.

6 IMPLEMENTATION

We implement DRFT in RocksDB, an open-source transactional key-value store [32] from Meta that is used in a number of production systems [62]. Figure 5 illustrates how DRFT schedules requests before passing them to the concurrency control and storage. We require minimal changes to existing database architectures to enhance DRFT's potential for adoption.

DRFT. DRFT requires several pieces of metadata for scheduling. First, DRFT requires a client identifier for each transaction. We assume that this information is available and note that, in practice, most database requests contain application identifiers (e.g., via metadata, routing information, etc.) even if connection pooling or DB proxies are used [63, 70]. In RocksDB, we modify the transaction `START` function to take in a `clientID`. Second, we must measure item usages for transactions and update client usages with these values. To reduce overheads, we optimize DRFT to only track the usage for items that observe many conflicts (i.e., hot items). We dynamically track hot items with a count min-sketch [26], a probabilistic data structure with bounded error. Most transactional workloads have a small group of hot items [18], so this optimization reduces the overheads of tracking item usages. To quantify item usage in an online system (given variance in the system and workload), we record the real time of operations. In RocksDB, we store this information in each transaction's metadata.

SMF. DRFT integrates SMF [18], which is a scheduler that orders requests to maximize throughput. SMF consists of two main components: a KNN classifier and a scheduling policy. The classifier is trained on a trace of requests for each workload to predict the set of hot items that input transactions will access. For inference, the classifier takes in application metadata (the transaction type and any items available upon instantiation) for each transaction and outputs a label corresponding to a set of hot items (labels are mapped to sets of hot items during training) that this request is likely to access. SMF's scheduling policy estimates the makespan (i.e., overall execution time) impact of different candidate transactions. Specifically, it samples a number of unscheduled transactions and computes the projected incremental makespan increase if each is added to the schedule. This accounts for the cost of potential conflicts that the unscheduled transaction has with the current ordering. DRFT passes transaction metadata to SMF and uses it to choose transactions that improve performance when $\Delta > 0$.

Concurrency control. We demonstrate DRFT's versatility by combining it with three concurrency control protocols. First, we support MVSchedO, a scheduling-optimized concurrency control [18] that augments MVTSO by queuing operations to follow the order established by the scheduler. We also layer our scheduler on top of the Locking and OCC protocols in RocksDB via a "bolt-on" approach with minimal modifications. Transactions are queued upon arrival and begin execution once they are scheduled by DRFT.

7 EVALUATION

In this section, we evaluate DRFT on a range of workloads. Specifically, we aim to answer the following questions:

- What is the impact of fair scheduling on performance?
- What are the tradeoffs between memoryless scheduling and maximizing throughput?
- What are the overheads of fair scheduling?

7.1 Experimental Setup

We implement various baselines in RocksDB (8.5) [34]. We run our database and clients on separate c5ad.16xlarge EC2 instances with 64 vCPUs, 128GB RAM, and local NVMe-based SSDs in the same

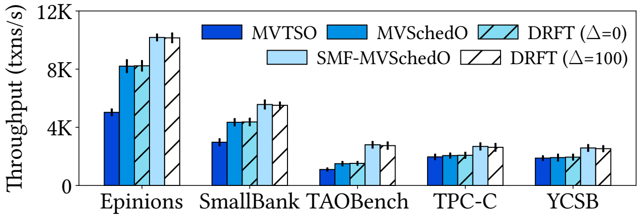


Figure 6: DRFT performance on application benchmarks.

region. Clients run in a closed-loop fashion with exponential backoff (accounting for aborts and retries when measuring latency), and we report the average of three 90 second runs with 30 seconds of warm-up each. For each workload, we tune the number of client and worker threads to ensure system saturation. For RocksDB parameters [5], we change several default values: we set the LRU (read) cache to 16GB, write buffer size to 64MB, level 0 compaction trigger to 4, level 1 buffer size to 256MB, and level 0 stop writes trigger to 36. We evaluate the following baselines (other than SMF-MVSchedO, all baselines use FIFO scheduling):

1. **RocksDB Optimistic Concurrency Control (OCC).** RocksDB's Optimistic transactions [34] provide up to Snapshot Isolation (SI) using optimistic concurrency control.
2. **RocksDB Locking (Lock).** RocksDB's Pessimistic transactions use a locking protocol [34] that only holds write locks and reads from snapshots to provide SI.
3. **RocksDB Multi-Version Timestamp Ordering (MVTSO).** We implement MVTSO [12] in RocksDB to provide serializability.
4. **MVSchedO.** We implement MVSchedO, a state-of-the-art concurrency control protocol that provides serializability [18].
5. **Shortest Makespan First (SMF-MVSchedO).** We implement SMF [18], a high-performance transaction scheduling policy.

Benchmarks. We evaluate the performance of our schedulers on a diverse set of standard benchmarks and real-world workloads. Our benchmarking implementation consists of 7K lines of Java. **Epinions** [30] consists of nine transaction types (78% reads) for a consumer reviews website. We run the benchmark with 2M users and 1M items (total data size of 50GB) with a Zipfian distribution ($\theta = 0.90$). **SmallBank** [71] contains six types of transactions (75% reads) that model a simple banking application. We configure it to run with 1M accounts (total data size of 50GB) with a Zipfian distribution ($\theta = 0.90$). **TPC-C** [27], a standard OLTP benchmark, simulates the business logic of e-commerce suppliers with five types of transactions. We run the workload under high contention with 10 Warehouses (total data size of 2GB, 68% reads). **TAOBench** [19] is a social network benchmark based on Meta's production traces. We run Workload T (>90% reads), which captures the full transactional workload on TAO, Meta's social graph database. We configure this benchmark to run with 10M objects (total data size of 100GB). **YCSB** is a microbenchmarking suite that generates read and write operations, which we group into sets of 16 for transactions following past work [17]. We use Workload B (95% reads, 5% writes) with a Zipfian distribution, and load 1M objects (total data size of 10GB).

7.2 Fairness on Application Benchmarks

We evaluate DRFT on the five application benchmarks (Figure 6) and finds that it provides high throughput while ensuring fairness between clients. For this set of experiments, we assume that each thread is a distinct client, ranging from 12 to 32 clients on these benchmarks to reach saturation due to contention (CPU usage and disk bandwidth remain low for all benchmarks). Each client sends the same workload per benchmark. As observed in prior work [31, 38, 39, 41], ensuring fairness can result in lower performance since fair schedules do not always maximize throughput. The key benefit of DRFT is that it provides the crucial fairness properties (share guarantee and strategy-proofness) with minimal throughput degradation. On all benchmarks, memoryless DRFT ($\Delta = 0$) has less than a 3% throughput difference with MVSchedO, which executes requests under FIFO. This is expected as memoryless DRFT approximates FIFO order since it picks the transaction of the client with lowest virtual start time, which alternates between all clients. DRFT has higher performance than MVTSO since it is implemented on top of MVSchedO, which prevents unnecessary aborts [18]. DRFT with $\Delta = 100$ms has less than a 5% throughput difference with SMF-MVSchedO. With a larger Δ, DRFT has the flexibility to schedule transactions that maximize concurrency. We find that DRFT's performance does not change significantly for Δ values beyond 100ms on these workloads, indicating this bound is sufficient for DRFT to attain throughput gains via SMF.

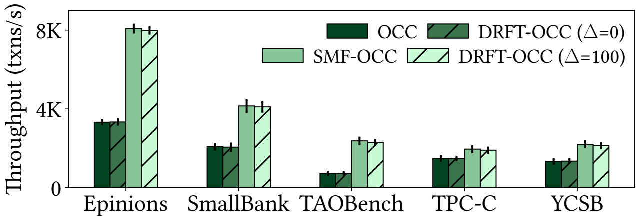
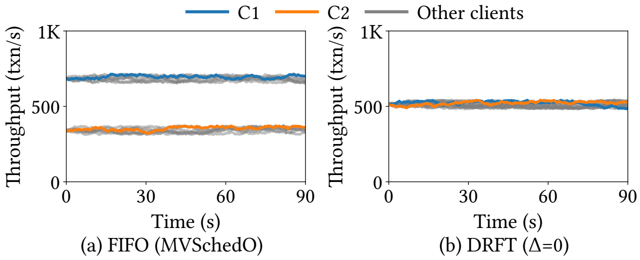


Figure 7: Bolt-on performance on application benchmarks.

7.2.1 Bolt-on Results. To demonstrate DRFT's extensibility, we evaluate it layered on top of RocksDB's OCC (DRFT-OCC) and Lock (DRFT-Lock) protocols (Figure 7). This set of experiments also assumes each thread is a distinct client, ranging from 10 to 28 clients for system saturation. Across all benchmarks, OCC and Lock have the same performance because they encounter similar conflicts when processing requests in FIFO order and abort at nearly equal rates (DRFT-OCC and DRFT-Lock also have equal performance), so we omit Lock results from Figure 7 due to space constraints. Memoryless DRFT-OCC has similar performance to the OCC baseline since OCC follows FIFO order, and DRFT-OCC mostly does as well while ensuring fairness across clients. DRFT-OCC with $\Delta = 100$ms has nearly identical performance with SMF-OCC. Both show throughput improvements over the baseline (ranging from 2.0× for SmallBank to 3.3× for TAOBench) since they prevent many conflicts and aborts. Both DRFT-OCC and SMF-OCC have lower throughput than SMF-MVSchedO because they only delay the transaction start while SMF-MVSchedO utilizes fine-grained operation scheduling to extract bigger wins. Overall, these results confirm that DRFT can be directly applied to existing concurrency control protocols with minimal changes to ensure fairness.

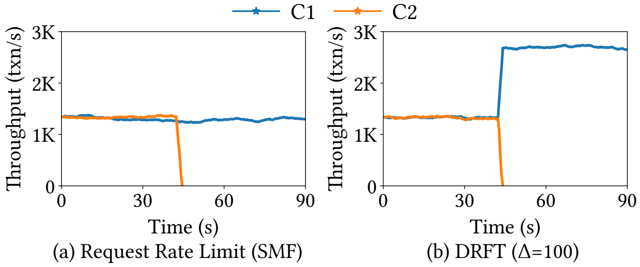
7.3 Fairness Properties

Next, we evaluate DRFT's ability to provide performance isolation and high utilization. We also test its performance under weighted, bursty, and mammoth workloads. Since OLTP benchmarks generally do not specify different clients, we construct client delineations *without* changing workloads. Specifically, we assign different request rates of the same underlying benchmark to clients.



(a) FIFO (MVSchedO)
(b) DRFT (Δ=0)

Figure 8: DRFT ensures equal usage across clients.



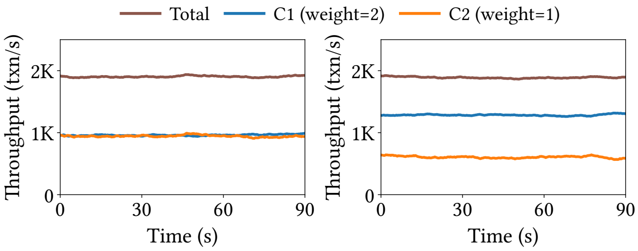
(a) Request Rate Limit (SMF)
(b) DRFT (Δ=100)

Figure 9: DRFT ensures high resource utilization.

Performance isolation. First, we demonstrate that DRFT ensures performance isolation. For this set of experiments, we have each client send a constant stream of requests from the Epinions benchmark, which has contention bottlenecks on popular users and items. We configure 16 clients (one thread per client), and we have half the clients send twice as many requests. Figure 8 shows the throughput of all clients over time (the aggregate throughput is identical to the results in Section 7.2). We highlight the performance of two clients, where C1 sends twice the number of requests as C2. Since more of C1's requests arrive and queue in the system, the baseline protocol (MVSchedO) executes more of C1's requests than C2's under FIFO, leading to C1 having double the throughput of C2. In contrast, the clients have equal throughput under DRFT ($\Delta = 0$); this is fair because the item usage of each client's transactions is equal. These results show that FIFO is unable to ensure performance isolation while DRFT provides the share guarantee.

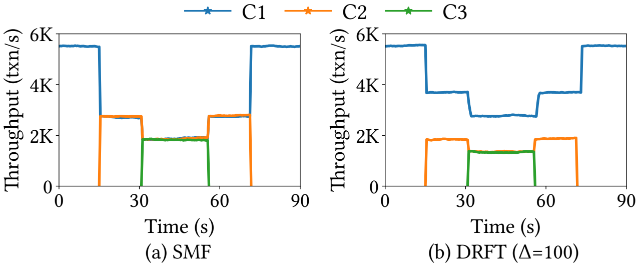
High utilization. Next, we show that DRFT provides high utilization. We perform experiments on the TPC-C workload, which mainly conflicts on the Warehouse and District items in the `NewOrder` and `Payment` transactions. We have two clients, C1 and C2, which are bottlenecked on conflicts to these items. We compare the performance of a request rate limit baseline using SMF, where each client is allocated half of the maximum possible throughput to ensure performance isolation (Figure 9). When C2 stops sending requests after 45s, DRFT ($\Delta = 100$) dynamically adjusts C1's share to maximize usage of the hot items. On the other hand, the request rate limit is brittle: once C2 stops sending requests, C1 still has the same throughput since each client is assigned a static rate limit, leading to 50% lower overall system throughput. These results show that request rate limits can lead to severe resource underutilization while DRFT ensures high utilization while serves requests fairly.

Weighted workloads. In production, multi-tenant databases often need to prioritize certain application requests. DRFT supports this functionality via weighted workloads. We demonstrate this on YCSB with two clients, C1 and C2, sending the same workload. C1 is assigned a weight twice that of C2. Figure 10 shows that DRFT ($\Delta = 0$) respects the weights of each client's workload: since C1 has twice the weight of C2, its throughput is twice as high.



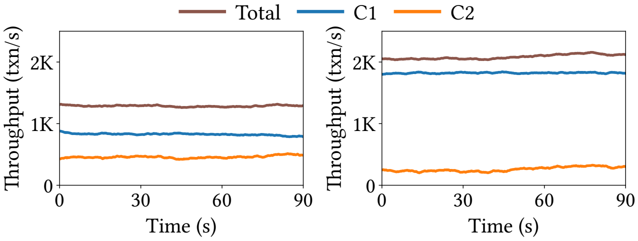
(a) FIFO (MVSchedO)
(b) DRFT (Δ=0)

Figure 10: DRFT respects request weights.



(a) SMF
(b) DRFT (Δ=100)

Figure 11: DRFT dynamically adjusts each client's share.



(a) FIFO (MVSchedO)
(b) DRFT (Δ=0)

Figure 12: DRFT ensures that mammoths execute fairly.

In contrast, MVSchedO executes an equal number of C1 and C2 requests, resulting in identical throughput for the clients, since it uses FIFO and does not account for weighted workloads.

Bursty workloads. Real-world applications frequency exhibit bursty workloads that often lead to logical contention (Section 2). We model this scenario with the SmallBank workload, which has contention on "hot" accounts. In this set of experiments, there is initially one active client (C1). Two other clients, C2 and C3, then join the system but send requests at half the rate of C1. The workload mixture of all clients is identical, and their requests are backlogged on the hot items. Figure 11 shows the results of DRFT ($\Delta = 100$) and SMF. DRFT ensures that C2 and C3 get their fair shares. When C2 becomes active at 15s, the throughput of C1 drops by half; this is fair since each client's workload has equal hot item usage. C3 also gets its fair share when it joins at 30s. In contrast, SMF does not ensure the share guarantee, so C1 has twice the throughput of the other clients when they are active. Notably, DRFT's total throughput is nearly equal to that of SMF. Since $\Delta = 100$, DRFT has sufficient flexibility to schedule transactions that maximize throughput while still providing fairness.

Mammoth transactions. To evaluate DRFT under long-running requests, we measure performance on mammoth transactions (>60 operations) from TAOBench. Specifically, C1 sends the benchmark workload excluding mammoths while C2 sends only mammoths, with both at equal rates. Under FIFO, C2 requests run more frequently (though C1 still has higher throughput since its requests are shorter and conflict less), leading to lower overall system performance since mammoths block the progress of other requests.

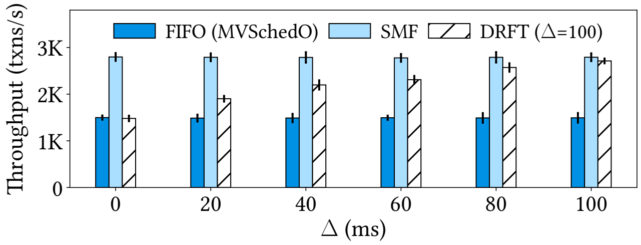


Figure 13: Performance of DRFT increases with Δ.

In contrast, DRFT (Δ = 0) fairly accounts for the usage of mammoths and reduces the rate at which C2's requests execute (by 42%) in equalizing client usage. As a result, the system has 56% higher overall throughput when each client gets its fair share.

7.4 The Impact of Δ

To quantify the tradeoff between fairness and throughput, we vary Δ on the TAOBench benchmark (Figure 13). We assume each thread is a separate client (12 clients total), and client workloads are identical. Since item accesses are drawn from probability distributions in this workload, hot items are not requested in a fixed order, as in many of the other benchmarks. Consequently, naively ordering requests in arrival order (as the baselines do) results in slow schedules and many aborts. However, intelligently scheduling these transactions can substantially increase performance. We compare DRFT with FIFO and SMF; all scheduling policies use MVSchedO as the underlying concurrency control protocol to isolate the impact of scheduling. Under memoryless DRFT (Δ = 0), scheduling proceeds in roughly FIFO order since all clients have the same workload, so we observe a similar throughput to FIFO. As we increase Δ, DRFT's throughput increases until it is less than 5% different from that of SMF. As we allow DRFT more flexibility in scheduling (and longer delays in achieving fairness), we observe higher overall throughput.

7.5 Fairness Overheads

To understand the overheads of ensuring fairness, we measure DRFT's performance under low ($\theta = 0.10$), medium ($\theta = 0.50$), and high ($\theta = 0.90$) contention on the YCSB workload (Figure 14). We assume each thread is a separate client, and we compare a read-dominant workload (95% reads) as well as a write-intensive workload (80% writes). Overall, DRFT has throughput no lower than 4% compared to the MVSchedO baseline across all workloads, regardless of θ. DRFT's overheads are more apparent under low contention (for which scheduling does not materially affect performance), but they are minimal: these overheads result from scheduling based on client usages and SMF computational costs from considering different candidate transactions to find fast schedules [18]. Once there is some contention ($\theta = 0.5$), both DRFT (Δ = 100) and SMF show throughput improvements compared to the baseline. The memory overheads of DRFT are minimal (and bounded) since we maintain only a small amount of metadata per client for item usage. The overheads of our scheduler are mitigated by the fact that we operate in a disk-based system; we view extending DRFT to in-memory databases as an interesting avenue for future work.

8 RELATED WORK

Fair scheduling. This paper builds a long line of work on fair scheduling, including start-time fair queuing (SFQ) [41], weighted fair queuing (WFQ) [31, 55], and many other algorithms [11, 40].

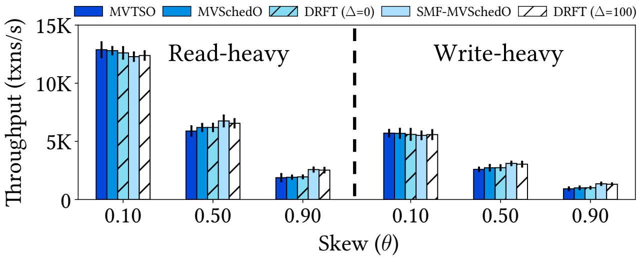


Figure 14: The overheads of DRFT are minimal.

For multi-resource settings, DRF [39] is a widely adopted standard, offering both the share guarantee and strategy-proofness. Other algorithms, such as Competitive Equilibrium from Equal Incomes (CEEI) [75], have been proposed, but they are not strategy-proof. Dominant Resource Fair Queuing (DRFQ) [38] extends DRF principles to ensure fair allocation over multiple (exclusive) resources for network packets and shares several features with DRFT. However, both DRFQ and DRF fall short for transactional workloads because they do not account for incremental resource acquisition.

DB resource allocation. A range of work [7, 42, 44, 51, 52, 59, 64, 72, 77] addresses multi-resource allocation for databases. Prior research concentrates on physical resource isolation [51, 64], intra-application interference [44], and fixed resource reservations for SQL stores [52]. However, these do not address logical resource contention and how transactions acquire resources gradually. Other related efforts aim for performance isolation via latency SLAs [22]/ SLOs [9]. Predictable performance [47, 68], often via admission control techniques [25, 80], is another area of focus that is complementary to the problem we address with fair sharing.

Transaction scheduling. Transaction scheduling has predominantly been tackled through the lens of concurrency control [12, 13, 54, 57] to improve performance. Most of these techniques operate within the (often implicit) constraint of arrival order (FIFO) and react to conflicts as they appear [1, 3, 4, 43, 45, 49, 53, 66, 67, 74, 78, 79, 81]. Some methods address the transaction schedule more explicitly by reordering the schedule after transaction commit [50] and/or abort [16, 33] but do not address performance isolation. Deterministic databases, which assume access to the full read-write sets of transactions, also schedule batches of requests explicitly via FIFO order [35–37, 73] or by partitioning workloads based on hot items [28, 56, 58, 60, 61, 82] for performance rather than fairness. DRFT complements these approaches by ensuring fair sharing for transactions. In particular, DRFT integrates a state-of-the-art scheduling policy, SMF [18] to achieve high throughput.

9 CONCLUSION

In this paper, we address the problem of fair sharing for transactions. We illustrate how ensuring fairness in this setting differs from prior work and highlight the challenges to providing both the share guarantee and strategy-proofness. We subsequently propose the Dominant Resource Fair Transaction (DRFT) scheduler, which accurately accounts for the item usage of each request to provide fairness between clients. DRFT also navigates the tradeoff between fairness and throughput, offering the flexibility for system operators to adapt to diverse workload needs. We evaluate DRFT on a range of workloads, showing that it ensures fairness with high performance and minimal overheads.

ACKNOWLEDGMENTS

We thank Dave Cecere, Shilpa Lawande, members of the Sky Lab, and the VLDB anonymous reviewers for their insightful feedback. This work is supported by gifts from Accenture, AMD, Anyscale, Google, IBM, Intel, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, SAP, and VMware.

REFERENCES

[1] 2020. MySQL Transactional and Locking Statements. https://dev.mysql.com/doc/refman/8.0/en/sql-transactional-statements.html

[2] 2021. Sharding Graph Data with Neo4j Fabric. https://neo4j.com/developer/neo4j-fabric-sharding/

[3] 2024. CockroachDB Transaction Layer. https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer

[4] 2024. PostgreSQL. https://www.postgresql.org/

[5] 2024. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

[6] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized Isolation Level Definitions. (2000), 67–78.

[7] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end performance isolation through virtual datacenters. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 233–248.

[8] Anonymous. 2024. Artifact link. https://github.com/audreyccheng/fair-txn-scheduler

[9] Michael Armbrust, Kristal Curtis, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. 2011. PIQL: success-tolerant query processing in the cloud. *Proc. VLDB Endow.* 5, 3 (nov 2011), 181–192.

[10] AWS. 2024. Quotas and constraints for Amazon RDS. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Limits.html

[11] Jon C. R. Bennett and Hui Zhang. 1996. WF2Q: worst-case fair weighted fair queueing. In *Proceedings of the Fifteenth Annual Joint Conference of the IEEE Computer and Communications Societies Conference on The Conference on Computer Communications - Volume 1* (San Francisco, California) *(INFOCOM'96)*. IEEE Computer Society, USA, 120–128.

[12] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[13] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[14] BigQuery. 2024. BigQuery Multi-statement Transactions. https://cloud.google.com/bigquery/docs/transactions

[15] 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.

[16] Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. 2023. Morty: Scaling Concurrency Control with Re-Execution. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 687–702.

[17] Yang Cao, Wenfei Fan, Weijie Ou, Rui Xie, and Wenyue Zhao. 2023. Transaction Scheduling: From Conflicts to Runtime Conflicts. *Proc. ACM Manag. Data* 1, 1, Article 26 (may 2023), 26 pages.

[18] Audrey Cheng, Aaron Kabcenell, Xiao Shi, Jason Chan, Peter Bailis, Natacha Crooks, and Ion Stoica. 2024. Towards Optimal Transaction Scheduling. *Proc. VLDB Endow.* 17, 4 (jul 2024).

[19] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. 2022. TAOBench: An End-to-End Benchmark for Social Network Workloads. *Proc. VLDB Endow.* 15, 9 (may 2022), 1965–1977.

[20] 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.

[21] Audrey Cheng, Jack Waudby, Hugo Firth, Natacha Crooks, and Ion Stoica. 2024. Mammoths are Slow: The Overlooked Transactions of Graph Data. *Proc. VLDB Endow.* 17, 4 (mar 2024), 904–911.

[22] Yun Chi, Hyun Jin Moon, and Hakan Hacigümüş. 2011. iCBS: incremental cost-based scheduling under piecewise linear SLAs. *Proc. VLDB Endow.* 4, 9 (jun 2011), 563–574.

[23] Google Cloud. 2024. Strategic Cloud Capacity Planning Using the Google Cloud Architecture Framework. https://www.googlecloudcommunity.com/gc/Community-Blogs/Strategic-Cloud-Capacity-Planning-Using-the-Google-Cloud/ba-p/178296

[24] Cloudflare. 2024. Introducing Advanced Rate Limiting. https://blog.cloudflare.com/advanced-rate-limiting/

[25] CockroachDB. 2024. Admission Control in CockroachDB: How It Protects Against Unexpected Overload. https://www.cockroachlabs.com/blog/admission-control-unexpected-overload/

[26] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[27] The Transaction Processing Performance Council. 2021. TPC-C. http://www.tpc.org/tpcc/

[28] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 48–57.

[29] Databricks. 2024. Databricks SQL. https://www.databricks.com/product/databricks-sql

[30] 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.

[31] Danny Dolev, Dror G. Feitelson, Joseph Y. Halpern, Raz Kupferman, and Nathan Linial. 2012. No justified complaints: on fair sharing of multiple resources. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (Cambridge, Massachusetts) *(ITCS '12)*. Association for Computing Machinery, New York, NY, USA, 68–75.

[32] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[33] Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. 2023. Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions. *Proc. VLDB Endow.* 16, 8 (apr 2023), 1930–1943.

[34] Facebook. 2023. RocksDB Github. https://github.com/facebook/rocksdb

[35] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.* 8, 11 (jul 2015), 1190–1201.

[36] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[37] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. 2014. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data.* 15–26.

[38] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. 2012. Multi-resource fair queueing for packet processing *(SIGCOMM '12)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[39] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 323–336.

[40] S Jamaloddin Golestani. 1994. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of INFOCOM'94 Conference on Computer Communications.* IEEE, 636–646.

[41] Pawan Goyal, Harrick M. Vin, and Haichen Chen. 1996. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *SIGCOMM Comput. Commun. Rev.* 26, 4 (aug 1996), 157–168.

[42] Ajay Gulati, Arif Merchant, and Peter J. Varman. 2010. mClock: handling throughput variability for hypervisor IO scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 437–450.

[43] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In *Proceedings of the 2021 International Conference on Management of Data.* 658–670.

[44] Yigong Hu, Gongqi Huang, and Peng Huang. 2023. Pushing Performance Isolation Boundaries into Application with pBox. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 247–263.

[45] 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.

[46] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 73–90.

[47] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Antony Rowstron. 2012. Bazaar: Enabling predictable performance in datacenters. *Microsoft Res., Cambridge, UK, Tech. Rep. MSR-TR-2012-38* (2012).

[48] Delta Lake. 2024. Delta Lake Transactions. https://delta-io.github.io/delta-rs/how-delta-lake-works/delta-lake-acid-transactions/

[49] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. 2017. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 21–35.

[50] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2047–2060.

[51] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted resource management in multi-tenant distributed systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 589–603.

[52] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. 2013. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *CIDR 2013* (cidr 2013 ed.). 6th Biennial Conference on Innovative Data Systems Research.

[53] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.

[54] Christos H Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653.

[55] Abhay K. Parekh and Robert G. Gallager. 1992. A generalized processor sharing approach to flow control in integrated services networks—the single node case. In *Proceedings of the Eleventh Annual Joint Conference of the IEEE Computer and Communications Societies on One World through Communications (Vol. 2)* (Florence, Italy) *(IEEE INFOCOM '92)*. IEEE Computer Society Press, Washington, DC, USA, 915–924.

[56] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 61–72.

[57] Dan R. K. Ports and Kevin Grittner. 2012. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (aug 2012), 1850–1861.

[58] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling highly contended OLTP workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 527–542.

[59] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. 2016. FairRide: near-optimal, fair cache sharing. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI'16)*. USENIX Association, USA, 393–406.

[60] Thamir M Qadah and Mohammad Sadoghi. 2018. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*. 13–25.

[61] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 180–194.

[62] RocksDB. 2025. RocksDB Users and Use Cases. https://github.com/facebook/rocksdb/wiki/RocksDB-Users-and-Use-Cases

[63] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. 2019. Scheduling OLTP Transactions via Learned Abort Prediction. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Amsterdam, Netherlands) *(aiDM '19)*. Association for Computing Machinery, New York, NY, USA, Article 1, 8 pages.

[64] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 349–362.

[65] Snowflake. 2024. Snowflake Unistore. https://www.snowflake.com/en/data-cloud/workloads/unistore/

[66] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. 2017. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 283–297.

[67] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. 2017. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 283–297.

[68] Zilong Tan and Shivnath Babu. 2016. Tempo: robust and self-tuning resource management in multi-tenant parallel databases. *Proc. VLDB Endow.* 9, 10 (jun 2016), 720–731.

[69] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 4–18.

[70] Dixin Tang, Hao Jiang, and Aaron J. Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All. In *CIDR*, Vol. 2.

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

[72] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow: a software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 182–196.

[73] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[74] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 18–32.

[75] Hal R Varian. 1973. Equity, envy, and efficiency. (1973).

[76] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052.

[77] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. 2012. Cake: enabling high-level SLOs on shared storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 14, 14 pages.

[78] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *OSDI*. 198–216.

[79] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*. 1643–1658.

[80] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Calton Pu, and Hakan HacigümüŞ. 2011. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 15, 14 pages.

[81] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*. 1629–1642.

[82] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 511–526.

[83] L. Zhang. 1990. Virtual clock: a new traffic control algorithm for packet switching networks. In *Proceedings of the ACM Symposium on Communications Architectures &amp; Protocols* (Philadelphia, Pennsylvania, USA) *(SIGCOMM '90)*. Association for Computing Machinery, New York, NY, USA, 19–29.