# Take Out the TraChe: Maximizing (Tra)nsactional Ca(che) Hit Rate

*Audrey Cheng*

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-270
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-270.html

December 16, 2022

**Take Out the TraChe: Maximizing (Tra)nsactional Ca(che) Hit Rate**

by Audrey Cheng

## Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

**Committee:**

Professor Ion Stoica
Research Advisor

12/15/2022
(Date)

* * * * * * *

Professor Natacha Crooks
Research Advisor

12/15/2022
(Date)

12/15/2022

# Take Out the TraChe:
# Maximizing (Tra)nsactional Ca(che) Hit Rate

*Paper #761*


## Abstract

Most caching policies focus on increasing object hit rate to improve overall system performance. However, these algorithms are insufficient for transactions. In this work, we define a new metric, transactional hit rate, to capture when caching reduces latency for transactions. We present DeToX, a caching system that leverages transactional dependencies to make eviction and prefetching decisions. DeToX is able to significantly outperform single-object alternatives on real-world workloads and popular OLTP benchmarks, providing up to a 130% increase in transaction hit rate and 3.4x improvement in cache efficiency.


## 1 Introduction

To improve latency at scale, application developers often layer caching systems, such as Memcached [74] and Redis [2], over standard data stores. These systems traditionally optimize for *object hit rate*, or how often requested objects can be served from cache. Unfortunately, current caching policies fail to capture the transactional nature of many application workloads. Since transactions access groups of objects atomically, caching will not improve latency unless *all* objects requested in parallel are present in the cache. For example, on a production workload from Meta's TAOBench [28] benchmark, we find that up to 90% of objects cached by LRU and LFU, two popular caching algorithms, do not have any impact on latency despite high object hit rates. Existing policies fail to capture the *all-or-nothing* property of transactions: if even a single key accessed in parallel is not in cache, it will harm performance.

Accordingly, object hit rate is the wrong objective for transactional workloads. Instead, we propose a new metric, *transactional hit rate*, which captures the latency reduction of transactions from accessing the cache. A policy that maximizes transactional hit rate should cache all objects requested in parallel within a transaction.

In this paper, we present DeToX, the first high-performance caching system that optimizes for transactional hit rate. In accordance with standard caching algorithms, DeToX assigns scores to objects and evicts those with the lowest values. As such, its policy is easily adaptable to existing caching systems. To rank objects in the transactional context, DeToX leverages the following insight: objects accessed in parallel within the same transaction should be scored together since they must all be cached to reduce transactional latency.

While scoring keys together might seem simple, the structure of transactional workloads complicates matters. Unlike previous work on caching for parallel jobs [12] and web applications [8, 11, 19, 96, 97], transactions need to be modeled as non-trivial directed acyclic graphs (DAGs) of read and write operations [23, 100]. Crucially, some keys within a transaction are accessed in parallel, but others are not. Consequently, a transaction's latency is determined by its *residual length*, or the number of sequential accesses on its longest, non-cached path. Rather than considering all keys in a transaction, we should focus on caching the *groups* of keys that reduce residual length.

Implementing a caching policy based on grouping presents several significant challenges. (1) For an arbitrary transaction, there can be an exponential number of groups, making scoring prohibitively expensive. (2) Identifying groups requires access to transactional DAGs, which are not always available. (3) Objects that are accessed by different transactions can belong to different groups, which have varying latency benefits if cached, and these disparities must be captured.

We address each of these issues in DeToX. (1) To avoid the overhead of an exponential number of groups, we introduce the notion of *interchangeable* keys: if two keys can replace each other in any group and still reduce residual length, then they can be represented by the same group. Interchangeable keys drastically curb the number of groups that need to be scored. (2) When transactional DAGs are not accessible, we propose a simplified policy that dynamically infers groups based on which requests are executed in parallel (termed *levels*). (3) Finally, we take group membership into account when scoring keys to ensure these values precisely reflect each object's contribution to future transactional hits.

Moreover, while our approach is primarily targeted at eviction, it also enables prefetching (Section 6). DeToX's prefetching policy tracks dependencies within transactions and brings items into cache preemptively when specific items are accessed earlier in the transaction.

Our eviction and prefetching algorithms are implemented in DeToX, which presents a key-value API that supports drivers for Redis [2], Postgres [3], and TiKV [4]. We evaluate our system on real-world workloads from TAOBench [28], a social network benchmark that models Meta's production workloads, as well as standard OLTP benchmarks (Epinions [39], SmallBank [93], and TPC-C [35]). Compared to single-object caching algorithms and systems, including LRU, LFU, GDSF [29], LIFE [12], and ChronoCache [48], our algorithm

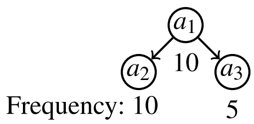



Figure 1: `GetLinkedAccounts` transaction.

can achieve up to a 130% increase in *transactional hit rate*, leading to a 3.4x improvement in cache efficiency (defined as the least amount of cache space required to achieve a particular transactional hit rate). For a Redis-Postgres setup, this translates into 31% higher throughput and 30% lower latency.

We note that our transactional hit rate metric prioritizes *latency* and exposes a new trade-off in caching enabled by the cloud's elastic resources: optimizing for latency versus shedding load off the system. Single-object policies provide high *object* hit rates but low transactional hit rates; they reduce load on the data store but do not always improve transaction request times. This work instead focuses exclusively on improving latency. In summary, we make the following contributions:

- We define a new metric, transactional hit rate, to evaluate the latency reduction of caching for transactions (Section 3).
- We provide the first formalization of transactional caching, and we prove that it is NP-Hard (Section 3.4).
- We present a new caching system, DeToX that leverages transactional dependency information to optimize for transactional hit rate and significantly improve performance on popular workloads (Sections 4 and 4.2).

## 2 Motivation

In this section, we illustrate why single-object eviction algorithms perform poorly for transactional workloads. Specifically, we show that a well-known optimality result in caching does not hold for transactions and that popular caching algorithms achieve low transactional hit rates.

### 2.1 Object Hit Rate is Insufficient

Most existing cache eviction algorithms focus on maximizing object hit rate, or the fraction of single object requests served by the cache. However, this approach fails to capture the inter-object dependencies that transactions introduce. Consider for example a simple transaction `GetLinkedAccounts` that returns secondary bank accounts $a_2, a_3$ linked with a primary account $a_1$ (Figure 1). This transaction must first read $a_1$ before accessing both secondary accounts $a_2$, $a_3$ in parallel. Thus, $a_1$, $a_2$ and $a_3$ are all on the longest path of the transaction. If we cache $a_1$, we can reduce the end-to-end latency of the transaction. However, if we additionally cache $a_2$, the overall latency does not improve because we still need to access $a_3$ from disk. In fact, caching $a_2$ or $a_3$ individually does not improve performance; transaction latency remains equivalent to the case in which *neither* key was cached. On the other hand, caching *both* $a_2$ and $a_3$ does improve latency.

Transactions have an implicit *all-or-nothing* property between groups of objects that traditional caching algorithms

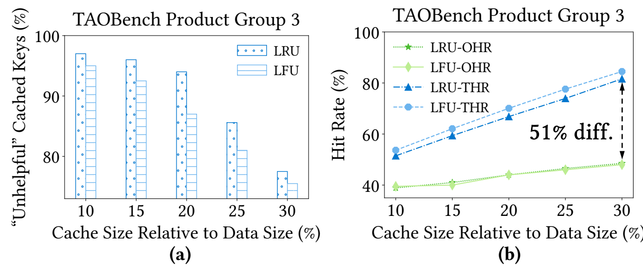



Figure 2: Single-object policy performance.

fail to capture. This can lead popular eviction algorithms, such as least recently used (LRU) or least frequently used (LFU), to make poor caching decisions. Consider a situation in which $a_2$ is more frequently accessed than $a_1$ while $a_3$ is almost never accessed. LFU and LRU would choose to evict $a_1$ and $a_3$ over $a_2$, resulting in no latency improvement for this transaction. Effectively, the low frequency of $a_3$ *contaminates* popular key $a_2$ since both need to be present to achieve a transactional hit.

**Real-world workloads.** This observation is not limited to our simple example. We find that single-object eviction algorithms also perform poorly for complex, real-world workloads. Figure 2a illustrates that over 90% of cached keys do not have any impact on latency ("unhelpful" keys) for the Product Group 3 workload of TAOBench [28]. The root cause is simple: these algorithms optimize for *object* hit rate (OHR) rather than transactional hit rate (THR). As we see in Figure 2b, LRU and LFU achieve high object hit rates but up to 51% lower transactional hit rates. Transactions in this workload access either a combination of hot keys and warm keys, or hot keys and cold keys. Single-object algorithms, which use only individual object features to score keys, retain hot keys from transactions in both categories but evict most warm keys and all cold keys. As a result, they achieve few transactional hits. A transactionally-aware policy would instead recognize that cold keys contaminate their associated hot keys and prioritize caching only the hot and warm keys that are accessed together.

### 2.2 Optimality

Our observations also have theoretical implications. We find that Belady [17], the offline, optimal eviction algorithm for uniformly-sized objects does not make the best decisions for maximizing THR. This policy evicts keys that are accessed furthest in the future but fails to take into account whether these keys generate transactional hits.

We prove that Belady is *not* optimal even for the simplest case of uniformly-sized transactions with uniformly-sized objects (Figure 3). In this example, we have four transactions with a cache size of 3. $T_1$ and $T_4$ access keys $a_1$,$a_2$,$a_3$, while $T_2$ accesses $a_4$,$a_5$,$a_6$ and $T_3$ accesses $a_4$,$a_5$,$a_7$. Belady chooses to first cache $a_1, a_2, a_3$ and then replace the last two keys with $a_4$,$a_5$ since these keys give object hits (but no latency reduction) for $T_3$. However, keeping $a_2$,$a_3$ in the cache would lead to a transactional hit (and latency improvement) for $T_4$.

| T | Keys accessed | Cache state | Optimal cache state |
|---|---|---|---|
| 1 | $a_1,a_2,a_3$ | - | - |
| 2 | $a_4,a_5,a_6$ | $a_1,a_2,a_3$ | $a_1,a_2,a_3$ |
| 3 | $a_4,a_5,a_7$ | $a_1,a_4,a_5$ | $a_1,a_2,a_3$ |
| 4 | $a_1,a_2,a_3$ | $a_1,a_4,a_5$ | $a_1,a_2,a_3$ |

Figure 3: Non-optimality of Belady.

## 2.3 Towards a new approach

Our results highlight how single-object caching strategies yield low transactional hit rates by storing many unhelpful objects. Web caching algorithms suggest a potential way forward: they acknowledge the need to cache multiple objects together (e.g., page-level hit ratio) but only consider flat dependencies [12, 97]. In contrast, transactions can have convoluted topologies with multiple levels of dependencies.

To develop a new transactionally-aware caching system, we must address three challenges: (1) formalizing caching in the transactional context, including optimality analysis (Sections 3 and 3.4), (2) efficiently identifying which groups of objects lead to transaction hits, given the potentially complex structure of transactions (Section 4), and (3) scoring the individual objects in these groups to determine which objects to store in the cache (Section 4.2). In our design, we are careful to emphasize compatibility with existing caching systems, such as Redis and Memcached, so that our approach can be easily implemented in these systems for greater applicability.

# 3 Transactional Caching

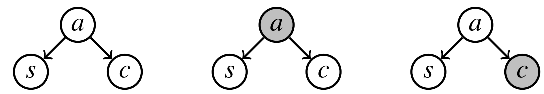
In this section, we formalize the transactional caching problem. We define a new metric, *transactional hit rate*, to capture the latency reduction of caching transactions.

## 3.1 Transactions

Transactions consist of read and write requests that must be applied atomically [23]. Some of these operations are independent and can execute in parallel, while others are *dependent* on the result of preceding operations. For instance, a read operation may query a key determined by the return value of a previous operation. As a result, these operations must be run sequentially. In effect, transaction execution can be captured by a DAG of operations. More formally, we apply the notion of a logical dependency, generalizing the model from [100]:

**Definition 1** *(Logical dependency). Given two operations $t$ and $p$ of a transaction, an operation $t$ is logically dependent on operation $p$ if $p$ determines the key or value accessed by $t$.*

Traditionally, these dependencies are not captured by the system, which observes only sequences of reads and writes. In practice, these relationships can be captured statically through program analysis or specified at run-time by the developer. Together, operations and logical dependencies define a transaction execution graph:




(a) Cache state: {} (b) Cache state: $\{a\}$ (c) Cache state: $\{c\}$

Figure 4: Smallbank `Balance` transactions.

```
1 id = SELECT cId FROM ACCOUNTS WHERE name = cName
2 s = SELECT savings FROM SAVINGS WHERE cId = id
3 c = SELECT checking FROM CHECKING WHERE cId = id
4 return s + c
```

Listing 1: Code corresponding to Figure 4. The dependency of Line 2 and Line 3 on the output of Line 1 is highlighted in red.

**Definition 2** *(Transaction execution graph). A transaction execution graph $G = (V,E)$ for transaction $T$ is a DAG, where each vertex in $V$ represents a pair $(x,X)$ of a read or write operation to key $x$ in table $X$, and edges in $E$ represent logical dependencies between the operations.*

Each transaction execution graph, disregarding the exact keys accessed, corresponds to a *transaction type*:

**Definition 3** *(Transaction type). A transaction type $\mathcal{T}_i$ is a class of transactions with execution graphs that are identical when projected onto tables.*

Transaction execution graphs can be extracted from application code [38, 100]. Within a type, each transaction accesses the same tables through the same logical dependencies but does not necessarily access the same keys. For example, the Smallbank workload [93] contains the transaction types: Amalgamate, Balance, DepositChecking, TransactSavings, SendPayment, and WriteCheck. For the Balance transaction (Listing 1), requests to both the Savings and Checkings tables are dependent on the result of the read to the Accounts table. The corresponding execution graph (Figure 4a) thus consists of three nodes, one for each operation, and logical dependencies $r[A] \rightarrow r[S]$ and $r[A] \rightarrow r[C]$. While the reads to $S$ and $C$ are independent and can be executed in parallel, they cannot proceed until after the read to $A$ finishes.

## 3.2 Cache

The previous section formalizes the notion of a transaction, including the logical dependencies that constrain a transaction's execution. We now formalize how *caching* affects transactions, drawing from [51] for notation.

**Definition 4** *(Cache state). A cache state is a set of keys $C$, where $|C| \leq n$, where $n$ is the capacity of the cache.*

By assumption, objects are served with lower latency from the cache than from the underlying system. We make the simplifying assumption that requests served from the cache have *zero* latency. Under this model, transaction latency is defined by the number of sequential, *non-cached* accesses. This corresponds to the longest path in the transaction's execution graph $G$, ignoring vertices with cached keys.

We formalize this notion as the residual transaction length:

**Definition 5** *(Residual transaction length). Given a transaction $T$ with transaction execution graph $G$, $K$ number of keys, and cache state $C$, the residual transaction length is the length of the longest path from any source vertex (no incoming edges) to any sink vertex (no outgoing edges) excluding vertices corresponding to keys in $C$. We define the function $L: G \times 2^K \rightarrow \mathbb{N}$ for which $2^K$ is the powerset of all keys, such that $L(G,C)$ is the residual transaction length.*

Given a transaction $T$ with execution graph $G$, $L(G,\{\})$ represents the length of the longest path in $G$ when the cache is empty. For example, Figure 4 has longest paths $\{r[a],r[c]\}$ and $\{r[a],r[s]\}$ with transaction length $L(G,\{\}) = 2$. Caching key $a$ (Figure 4b) would shorten the residual transaction length to $L(G,\{a\}) = 1$, as the longest paths are reduced to $\{r[c]\}$ and $\{r[s]\}$. However, caching key $c$ (Figure 4c) does not change the residual transactional length, since $\{r[a], r[s]\}$ remains the longest path with $L(G,\{c\}) = 2$. Informally, we refer to a *transactional hit* when we have a length reduction of one.

## 3.3 Transactional Hit Rate (THR)

Having defined the necessary formalisms for transaction latency and caching, we can now introduce transactional hit rate. Informally, this metric captures how much latency improves when caching for transactions, much like how its single-object counterpart, object hit rate, does so for individual requests.

We first present THR in the context of a single transaction:

**Definition 6** *(Individual transactional hit rate). Given transaction $T$ with execution graph $G$ and cache state $C$, the individual transactional hit rate is $\frac{L(G,\{\})-L(G,C)}{L(G,\{\})}$.*

The difference in transaction length represents the reduction in sequential, non-cached accesses after caching. We normalize this difference by dividing by the total transaction length. This metric captures the impact of caching for the execution of a single transaction. We can easily extend this definition to a sequence of transactions:

**Definition 7** *(Transactional hit rate). Given a sequence of transactions $T_1$, $T_2$, ... , $T_m$ with execution graphs $G_1, G_2, \ldots, G_m$ and the respective cache states at the time of execution $C_1$, $C_2$, ... , $C_m$, the transactional hit rate is $\frac{\Sigma_{i=1}^{m}(L(G_i,\{\})-L(G_i,C_i))}{\Sigma_{i=1}^{m} L(G_i,\{\})}$.*

## 3.4 Optimality Analysis

Single-object caching is a well-studied problem: past work has shown that Belady's algorithm is provably optimal for eviction when considering uniformly-sized objects, and this problem is NP-Hard in the general case [31]. However, Belady does not provide the best possible performance for transactional caching (proof in Appendix). We further show in this appendix that the optimal problem in this setting is NP-Hard by reducing this problem to variable-sized caching.

# 4 Group Identification and Scoring

Designing an optimal caching policy is impractical for transactional caching, since it would run in exponential time. Unfortunately, traditional heuristics perform poorly for transaction hit rate (Section 2) because they fail to identify the keys that must be cached as a group to yield a transactional hit. This notion of *grouping* is central to our new transactionally-aware caching policy. We proceed in two steps: first, we identify which groups of keys lead to transactional hits when cached together (*group identification*). Second, we determine what scores should be assigned to each key within a group (*group scoring*).

## 4.1 Group Identification

Intuitively, a group is a set of keys that, if cached together, reduce residual transaction length. Specifically, we define the notion of a *complete group* from which one cannot remove any key without increasing residual transactional length. Completeness optimizes cache efficiency by storing the minimal subset of keys necessary to reduce latency. Formally:

**Definition 8** *(Complete group). Given a transaction $T$ and its execution graph $G$, a complete group is a subset of keys $g$ accessed in $T$ such that $\forall g' \subset g$, $L(G,g) < L(G,g')$.*

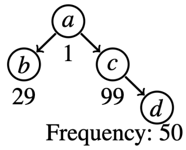
We use static analysis to identify complete groups at compile time, applying previous work to capture logical dependencies [100]. A simple algorithm would be to iterate through the powerset of all possible table accesses for each generated execution graph to identify complete groups and compute their resulting reductions in transactional length.

Consider Figure 5a, which has a transaction length of three (serial accesses $a$,$c$,$d$) and seven complete groups ($\{a\}$, $\{c\}$, $\{d\}$, $\{a,c\}$, $\{a,d\}$, $\{b,c,d\}$, $\{a,b,c,d\}$). Note that $\{c,d\}$ is not a complete group: it generates only one transactional hit (reducing the length from three to two), but caching $c$ or $d$ individually also generates one transactional hit. Similarly, $\{a,b\}$ is not a complete group because it generates a transactional hit regardless of whether $b$ is cached.

In the worst case, the number of complete groups can be exponential in the size of the transaction, even for simple transaction topologies. Fortunately, many of these groups are in fact *equivalent*. We describe this notion more precisely in Section 5.1 and presents an optimization that drastically reduces the number of groups that need to be considered.

## 4.2 Scoring

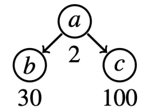
Caching policies typically assign scores to keys and evict objects with lower scores. We adopt the same strategy and carefully map complete groups to scores for individual keys. This approach has two benefits: 1) we can draw from prior work on single-object caching algorithms, 2) we minimize implementation changes to real caching systems.




**(a) Transaction 1**

| Groups | $\{a\}$ | $\{c\}$ | $\{d\}$ | $\{a,c\}$ | $\{a,d\}$ | $\{b,c,d\}$ | $\{a,b,c,d\}$ |
|---|---|---|---|---|---|---|---|
| SCORE_G | 1 | 99 | 50 | 1 | 1 | 19.3 | 0.75 |
| SCORE_I | - | $c$ | - | - | - | $b,d$ | $a$ |

**(b) Scores for Transaction 1**




**(c) Transaction 2**

| Groups | $\{a\}$ | $\{b,c\}$ | $\{a,b,c\}$ |
|---|---|---|---|
| SCORE_G | 2 | 15 | 1.3 |
| SCORE_I | - | $b,c$ | $a$ |

**(d) Scores for Transaction 2**

$$\texttt{SCORE\_K}(key) = \frac{TS_{key}}{F_{key}} + A_{global}$$

**Key Score Formula**

| Keys | $a$ | $b$ | $c$ | $d$ |
|---|---|---|---|---|
| Previous $TS_{key}$ | 0 | 30 | 200 | 70 |
| SCORE_K | 0.75 | 1.7 | 3.02 | 1.8 |

**(e) Agg. Key Scores after Txn 1**

| Keys | $a$ | $b$ | $c$ |
|---|---|---|---|
| Previous $TS_{key}$ | 0.75 | 49.3 | 299 |
| SCORE_K | 2.13 | 2.9 | 3.89 |

**(f) Agg. Key Scores after Txn 2**

| T | Cache state | $A_{global}$ |
|---|---|---|
| - | $\{a,b,c\}$ | 0 |
| 1 | $\{b,c,d\}$ | 0.75 |
| 2 | $\{b,c,a\}$ | 1.8 |

**(g) Cache State**

Figure 5: Example transactions and scores. Key sizes are 1, cache size is 3, and $A_{global}$ starts at 0. Cache initially stores $\{a,b,c\}$. $a$ is evicted after T1, and $d$ is evicted after T2, with $A_{global}$ updated on each eviction.

### 4.2.1 Scoring a Group in a Single Transaction

We begin by assigning individual numerical scores to each group (a *group score*), with higher values representing groups that are more beneficial to cache. We draw inspiration from GDSF, a high-performing web caching algorithm [29]. GDSF considers three metrics to score keys: frequency (access count), recency (time since last access), and size. We leverage frequency and size to score each group as follows (and incorporate recency into key scores in Section 4.2.3) :

$$\texttt{SCORE\_G}(group) = \frac{\texttt{min}(F_{group}) \times L_{group}}{S_{group}}$$

$F_{group}$ is a list of all key frequencies in the group. $L_{group}$ is the number of transactional hits generated if this group is cached. $S_{group}$ is the sum of all key sizes in the group. All scoring parameters can be found in Table 1. We consider the transactions in Figure 5 as running examples. The group scores of each complete group for these transactions are shown in Figures 5b and 5d. For instance, Figure 5a has keys $a,b,c,d$ with frequencies of 1, 29, 99, and 50, respectively and sizes of 1. The score of group $\{a,b,c,d\}$ is thus $\frac{\texttt{min}(1,29,99,50)\times 3}{4} = 0.75$.

**Frequency ($F_{group}$).** Keys within a complete group may vary in frequency but must all be cached to yield a hit. For example, if a high-frequency key $x$ is only associated with a group of keys $\{y\_1...y\_k\}$ (each with much lower frequency than $x$), then it is not beneficial to cache $x$. In effect, the key with the minimum frequency determines the cacheability of the entire group since it contaminates the other keys. Thus, we take the minimum of all key frequencies in calculating the score of a group. Consider for instance Figure 5c: key $c$ is more frequently accessed than key $b$. As a result, $b$ drives down the frequency of the group $\{b,c\}$ to $\texttt{min}(F_{group}) = \texttt{min}(30,100) = 30$.

**Transactional length reduction ($L_{group}$).** This parameter captures the reduction in transactional length when caching a group ($L_{group} = L(G,\{\}) - L(G,group)$). Other factors being equal, groups with higher latency reductions are better choices for caching and should thus be assigned a higher score.

**Size ($S_{group}$).** All keys must be present in the cache to generate a transactional hit. $S_{group}$ represents the cache space needed to store the group. THR is maximized by retaining groups of smaller sizes as more of these groups can be cached.

| Parameter | Description |
|---|---|
| $\texttt{SCORE\_G}(group)$ | Score of a group |
| $F_{group}$ | Set of all key frequencies in a group |
| $L_{group}$ | Transactional hits for a group |
| $S_{group}$ | Sum of key sizes in a group |
| $\texttt{SCORE\_K}(key)$ | Score of a key |
| $TS_{key}$ | Sum of instance scores for a key |
| $F_{key}$ | Frequency of a key |
| $A_{global}$ | Global aging factor |

Table 1: Scoring parameters.

Next, we describe how to go from group scores to key scores.

### 4.2.2 Scoring Across Groups in a Single Transaction

Mapping group scores to keys is challenging: keys can belong to multiple groups and their contribution to reducing transaction length depends on which other keys are present in the cache. For example, consider the groups $\{c\}$ and $\{d\}$ in Figure 5a. Individually, they can both reduce transactional length by one. However, $\{c,d\}$ is not a complete group, so caching both $c$ and $d$ is wasteful unless $b$ is also cached. Consequently, one should not give both $c$ and $d$ a high score unless $b$ also is given a high score.

To address this issue, we adopt an iterative approach to condition each key's *instance score* (SCORE_I) on all previous scoring decisions for this transaction. Our algorithm first computes group scores with SCORE_G for all complete groups in the transaction. At each iteration, it then finds the highest scoring complete group that is a *superset* of all previously scored groups and assigns each unscored key in that group the corresponding group score. The protocol terminates when all keys have been scored (note that all keys will eventually be scored since they are part of the trivial complete group that contains all keys of the transaction). This algorithm greedily favors caching groups of keys with higher scores and accounts for these decisions in all subsequent scoring choices.

Consider again Figure 5a. We use the group scores in Figure 5b to find instance scores. The first group selected is $\{c\}$ since it has the highest group score of 99, and the

SCORE_I row indicates that $c$ is assigned the corresponding SCORE_G. The next group with the highest score that contains $c$ is $\{b,c,d\}$, so $b$ and $d$ are assigned the group score of 19.3. Intuitively, our algorithm captures the fact that, once $c$ is cached, $d$ should only be cached when $b$ is cached. The low score of $b$ contaminates $d$ but should not contaminate $c$ (since $c$ by itself can lead to a transactional hit).

#### 4.2.3 Scoring Across Transactions

Finally, we describe how to integrate instance key scores across multiple transactions into an *aggregate* value. This final score will be used by the system to decide which keys to evict from the cache. We adopt the following formula:

$$\texttt{SCORE\_K}(key) = \frac{TS_{key}}{F_{key}} + A_{global}$$

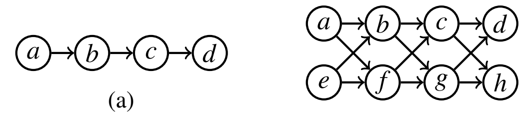
$TS_{key}$ is the sum of all instance scores from Section 4.2.2 across all transactions accessing this key. $F_{key}$ is the frequency of this key. $A_{global}$ is the global aging factor.

**Averaging instance scores.** To map instance key scores to a single value for a given key, we take the running average of these scores. Each time a key is accessed, we add its instance score to the total score $TS_{key}$ and increment $F_{key}$ before calculating a new aggregate score. Figure 5e gives the key scores of $a$, $b$, $c$, and $d$ after executing the transaction in Figure 5a, assuming that the aging factor is initialized to 0, key size is 1, and the previous $TS_{key}$ values are 0, 30, 200, 70 respectively. For example, $c$ has an instance score of 99 (Figure 5b) for the transaction in Figure 5a, a previous $TS_{key}$ of 200, and frequency of 99, giving $\texttt{SCORE\_K}(c) = \frac{200+99}{99} + 0 = 3.02$ in Figure 5e.

**Recency.** GDSF, along with many other algorithms [9], uses an aging factor to account for recency. Since object access distributions can shift over time, previously popular objects can remain in the cache for extended periods since their frequencies are high, preventing newly popular objects from being cached. $A_{global}$ is a global value added to the score of a key upon each access to increase the scores of more recently accessed objects and age older objects out of cache. It is updated each time an object is evicted and set as that object's score. $A_{global}$ acts as a "reset" on scores and ensures that all accesses after this eviction will have scores higher than the last evicted key. In Figure 5, $a$ is evicted after the transaction in Figure 5a, and $A_{global}$ is set to its score (0.75). This value is then added to SCORE_K for each key accessed the subsequent transaction in Figure 5c. For example, $c$ has an instance score of 15 (Figure 5d) for the transaction in Figure 5c, a previous $TS_{key}$ of 299, frequency of 100, and $A_{global}$ of 0.75, giving $\texttt{SCORE\_K}(c) = \frac{299+15}{100} + 0.75 = 3.89$ in Figure 5f.

## 5 Optimizations

Our current approach to grouping and scoring can be prohibitively expensive, since the number of complete groups




(a) (b)

Figure 6: Transactions to demonstrate interchangeability.

is exponential for some topologies. We address this problem in two ways. (1) We observe that many complete groups capture redundant information and introduce *interchangeable groups* to avoid scoring all complete groups and dramatically reduce run-time overhead. We compute *interchangeable groups* offline by static analysis. (2) Assuming no access to code (i.e., we do not know the transaction's DAG), we present a restricted form of grouping, *levels*, that dynamically approximates groups at run-time.

### 5.1 Interchangeability

The number of complete groups can be exponential with respect to transaction size. For example, in Figure 6a, all possible groups in the powerset are complete (e.g., caching $\{a\}$ gives a transactional hit, so does $\{a,b\}$, $\{a,b,c\}$, etc).

We observe that transactions often contain complete groups that differ by only a single key. For instance, for every group in which $b$ is present in Figure 6a, there exists an identical group in which $a$ replaces $b$ (and vice-versa). In effect, these keys can be "swapped" with each other and still produce a complete group. This interchangeability property is powerful: if two keys can be exchanged in any complete group, then deciding to cache one key over the other is entirely dependent on the *individual* scores of these keys, as all other parameters are shared. Consequently, we do not need to calculate the scores of all their complete groups in order to score each key. Consider the groups $\{a,c\}$ and $\{b,c\}$ where $a$ has a higher individual score than $b$. If $a$ and $b$ are interchangeable, then we know that $\{a,c\}$ must have a higher group score than $\{b,c\}$. Since our scoring algorithm favors caching groups with higher scores, we can avoid calculating the score of $\{b,c\}$ at run-time.

We can further generalize the idea of interchangeability to *sets* of keys that can also be "swapped" with each other. We call such sets *interchangeable groups*:

**Definition 9** *(Interchangeable groups). Let $s_1$ and $s_2$ be distinct sets of keys in a transaction with execution graph G. We say that $s_1$ and $s_2$ are interchangeable if*
*(1) $\forall$ complete groups $g_1$ such that $s_1 \subseteq g_1$ and $s_2 \cap g_1 = \varnothing$, $g_1' = g_1 \setminus s_1 \cup s_2$ is also a group and $L(G,g_1) = L(G,g_1')$, and*
*(2) $\forall$ complete groups $g_2$ such that $s_2 \subseteq g_2$ and $s_1 \cap g_2 = \varnothing$, $g_2' = g_2 \setminus s_2 \cup s_1$ is also a group and $L(G,g_2) = L(G,g_2')$.*

Computationally, interchangeability allows us to reduce the number of complete groups scored at run-time. We compress the representation of complete groups and reduce run-time complexity of the scoring algorithm as follows, using Figure 6b as a running example:

• (Compile-time) **Find** all interchangeable groups in the set of complete groups. The complete groups are: $\{a,e\}$, $\{b,f\}$, $\{c,g\}$, $\{d,h\}$, $\{a,e,b,f\}$, $\{c,g,b,f\}$, $\{d,h,b,f\}$, $\{a,e,c,g\}$, $\{a,e,d,h\}$, $\{c,g,d,h\}$, $\{a,e,b,f,c,g\}$, $\{a,e,b,f,d,h\}$, $\{a,e,c,g,d,h\}$, $\{c,g,b,f,d,h\}$, $\{a,e,b,f,c,g,d,h\}$. Consider replacing $\{a,e\}$ with $\{d,h\}$ in any complete group; the resulting group is still complete. Thus, $\{a,e\}$ and $\{d,h\}$ are interchangeable. Using the same logic, we find that $\{a,e\}$,$\{b,f\}$,$\{c,g\}$,$\{d,h\}$ are all mutually interchangeable.

• (Compile-time) **Compress** complete groups. Denote an access to any one of the mutually interchangeable groups—$\{a,e\}$, $\{b,f\}$, $\{c,g\}$, $\{d,h\}$—as $[C]$. For example, $\{a,e,b,f,d,h\}$ becomes $[C,C,C]$. In this particular example, all groups of size four can be written as $[C,C]$, groups of size six as $[C,C,C]$, and groups of size eight as $[C,C,C,C]$. We call these representations compressed groups.

• (Run-time) **Score** compressed groups. Recall from Section 4.2.2 that our instance scoring algorithm scores all complete groups before greedily selecting the highest-scoring ones. With interchangeability, we no longer need to score all complete groups. Assume the minimum scores of the following interchangeable groups are: $\{a,e\}:1$, $\{b,f\}:10$, $\{c,g\}:30$, $\{d,h\}:50$. Since we know that $\{a,e\}$ and $\{d,h\}$ are interchangeable and that $\{d,h\}$ has a higher score, for any complete group containing $\{a,e\}$, there must be another complete group containing $\{d,h\}$ that has the same (or higher) score. Applying this intuition, the highest-scoring complete group corresponding to the compressed group $[C,C]$ must be composed of the highest and second-highest scoring interchangeable groups, $\{d,h\}$ and $\{c,g\}$ respectively.

## 5.2 Levels

For transactions in which code is unavailable, we design a simplified protocol to dynamically infer groups. We define a *level* to be a set of keys in a transaction that are sent to the data store in parallel; similar definitions are used to group tasks to optimize caching for parallel job execution [12]. We assume that applications send requests as soon as their logical dependencies are fulfilled. For instance, the transaction in Figure 5a has levels $\{a\}$, $\{b,c\}$, and $\{d\}$. We have $d$ as a standalone level since it can only be requested once $c$ has finished executing.

We note that in transactions where all keys and groups are interchangeable, as in Figures 6a and 6b, all levels are complete groups. In these cases, using either levels or complete groups is equivalent. We find that many real workloads have such topologies (including all the ones we evaluate in Section 8). However, levels consider only a subset of all possible complete groups, so it can miss out on performance opportunities for unbalanced topologies. For example, in Figure 5a, $b$ and $c$ are scored together under levels, lowering $c$'s score. To maximize transactional hits, $b$ should instead be scored with $d$ since both are colder keys, and $c$ should be given a high score because caching this key by itself is likely to lead to a transactional hit. We measure this tradeoffin Section 8.

# 6 Prefetching

Prefetching is a popular technique to reduce the client-perceived latency of requests by caching items before they are requested [11, 26, 47, 48, 96]. We revisit this strategy in the context of transactions and design a new prefetching algorithm that uses logical dependencies to minimize latency.

Prefetching leverages conditional probabilities: once key $a$ is accessed, it may be very likely that key $b$ will also be requested in the same transaction. Consider for example `GetLinkedAccounts` in Figure 1: the access to a primary account is almost always followed by requests to the same subsidiary accounts. Our prefetching algorithm tracks these correlations and brings such objects in the cache. Specifically, DeToX uses logical dependencies and identifies, for every request $r$, sets of keys in subsequent accesses that are logically dependent on $r$. The policy then tracks which of these sets is most frequently accessed and preemptively fetches in the most popular set into cache alongside $r$.

# 7 Implementation

In this section, we describe our implementation of DeToX, which consists for 7K lines of Java. We adopt a standard two-tier architecture in which we layer a Redis (7.0) cache on top of Postgres (12.10) or TiKV (5.4.3). A shim layer routes requests, manages concurrency control, and enables prefetching.

## 7.1 Shim Layer

All client requests are directed to the shim layer, which mediates accesses to the cache and data store to support serializable transactions. Read requests go first to Redis. In the absence of a cache hit, the shim sends the request to the data store and updates the cache with the result. All writes are forwarded to the data store. While our shim layer currently supports a key-value API, we can convert SQL queries to this format, as previous systems have done [36, 37, 50, 62–66, 70–72, 84, 87–90, 98, 104]. We choose to implement a separate shim layer since there is limited open-source support for concurrency control between caching and data store systems [10, 46, 48, 49, 80, 81, 91, 94]. Furthermore, our shim layer allows us to easily plug in different caches and data stores. We will explore integrating transactional caching directly into systems in future work.

**Concurrency control.** We implement two-phase locking [24] with timeout-based deadlock detection in the shim layer to maintain serializability. The system maintains the following invariant: values in the cache will either 1) reflect the value committed in the (serializable) data store or 2) be protected by an exclusive write lock.

To achieve this, the shim acquires locks on individual objects before sending requests to either storage system. Writes are buffered at the shim layer until commit. All requests to the data store are sent as a part of the same transaction. Once values are committed in the data store, they are updated in the cache before write locks are released. To handle crashes, we rely on the data store as the source of truth, similar to previous work [49, 80, 81], and we clear the cache after failure to prevent stale reads.

**Extracting transaction types and execution graphs.** We extract logical dependencies and transaction types from application code in line with prior work [38, 100]. The widespread adoption of JDBC-style drivers presents a common interface for extracting transactions across applications.

## 7.2 Eviction

Our eviction strategy scores keys as a function of their groups as well as their frequency, size, and recency. The latter three are all features that are readily available in Redis, which natively supports LRU and LFU. We reuse many of these metrics to minimize code changes when implementing our heuristics. We make two primary modifications to Redis: we add 1) a global aging factor that is updated during eviction (as detailed in Section 4.2.3), and 2) support for scoring groups of keys. Our changes involve less than 100 lines of code and suggest that DeToX can be easily integrated into any caching system. We also implement a trace-driven simulator in Python to evaluate offline algorithms Belady and Transactional Belady.

# 8 Evaluation

In this section, we answer the following questions:

- How does DeToX compare to single-object algorithms in terms of transactional hit rate and cache efficiency?
- What is the impact of our grouping optimizations?
- What is the tradeoff between optimizing for object hit rate and transactional hit rate?

## 8.1 Experimental Setup

We run our shim layer and Postgres on separate c5a.4xlarge Amazon EC2 instances (16 CPUs, 32GB RAM) and use a memory-optimized r5.4xlarge machine (16 CPUs, 128GB RAM) for Redis. Clients run on c5a.16xlarge instances (64 CPUs, 128GB RAM). We host all machines in the same region with low network latency (0.2ms). For our experiments, we report the average of three 5-minute runs with 60 seconds of warm-up time. When an eviction is needed, we score 10 random samples and choose one to evict among these candidates. This strategy removes the overhead of maintaining a sorted list of keys without degrading performance and is popular in many caching systems [2, 85], including Redis.

**Benchmarks.** We evaluate DeToX and several single-object baselines against a range of workloads. **TAOBench** [28] is an open-source social network benchmark based on Meta's production traces. We run the Product Group 1, 2, and 3 workloads, which represent distinct sets of (anonymized) applications at Meta that share data and use the same product infrastructure. All workloads are read-heavy and skewed, typical of most social networks. They contain point reads and writes (inserts, updates, and deletes) as well as read-only and write-only transactions. Since transaction code is not available for this benchmark, we use levels to score groups for eviction. [1] We run experiments with 100M objects for a total data size of around 1 TB. **Epinions** [39] consists of nine transaction types that represent behavior of a consumer reviews website. We run the benchmark with 2M user and 1M items for a total data size of roughly 1 TB. **SmallBank** [93] contains six types of transactions that model a simple banking application. We configure it to run with 500M (uniformly accessed) accounts (total size of 1TB). **TPC-C** [35], a standard e-commerce OLTP benchmark, simulates the business logic of e-commerce suppliers with five types of transactions. We configure TPC-C to run with 100 warehouses (total size of 8GB. In line with prior transactional key-value stores [36, 87], we use a separate table as a secondary index on the Order table to locate a customer's latest order in the Order-Status transaction, and on the Customer table to look up customers by their last names (for the Order-Status and Payment transactions).

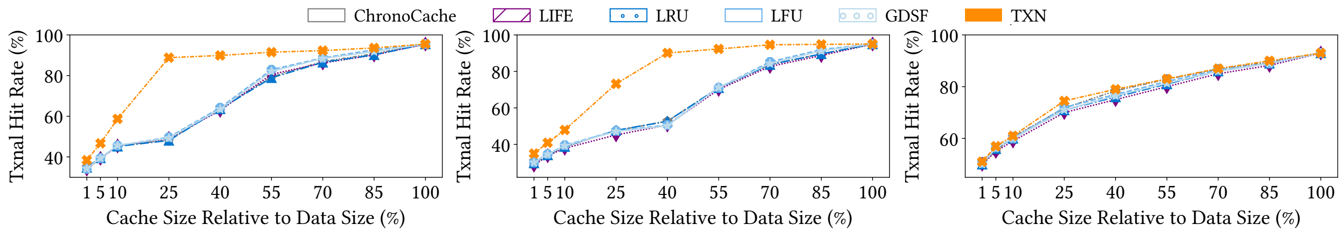
## 8.2 Application Benchmark Results

We show THR over different cache sizes for all benchmark workloads in Figure 7. We omit some throughput / latency graphs for space but describe results in text.

**TAOBench.** DeToX obtains up to 76% higher transactional hit rates on the TAOBench PG2 and PG3 workloads compared to single-object caching algorithms (Figures 7a and 7b). DeToX achieves this with better cache efficiency: at the 25% cache size relative to data size (a common setup following the "80-20 rule"), the protocol achieves an 88% transactional hit rate while the best single-object algorithm requires 3.4x more cache space to attain the same result on PG2. Results are similar for PG3 for which the system requires a 2.2x smaller cache. Throughput increases by 31% (from 18K txns/s to 24 txns/s) for PG2 and 30% for PG3 (from 31K txns/s to 40 txns/s), while latency reduces by 30% (4.6ms to 3.2ms) for PG2 (Figure 8b) and 29% for PG3 (2.3ms to 1.6ms).

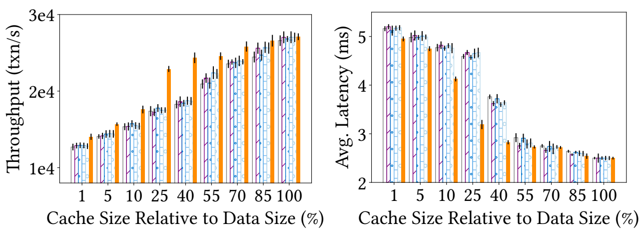
PG2 is read-dominant (>96%) with a mix of point reads, short transactions (<10 operations), and larger read transactions that span up to 40 keys. The point reads and shorter transactions make up 60% of the workload and largely access a small group of hot keys. Consequently, all algorithms achieve a THR of over 45% for small cache sizes (10% relative size). The longer read transactions follow one of two patterns: transactions access either a combination of hot and warm keys (25%), or hot and cold keys (11%). Transactions from the first category are more beneficial to cache since their keys are more

[1] TAOBench [28] chooses to model workloads using probability distributions rather than fixed query types for adaptability.




**(a) Product Group 2** **(b) Product Group 3** **(c) Product Group 1**

Figure 7: TAOBench results.




**(a)** **(b)**

Figure 8: TAOBench PG2 results.

frequently accessed and more likely to lead to transactional hits. There is little benefit in caching any of the keys in the second category (including hot keys) since transactional hits are unlikely given the presence of cold keys in each group.

Under DeToX, the cache initially chooses to cache keys that belong to transactions in the first category (with higher scores). Thus, transactional hit rate improves as the cache size increases from 10% to 40% (Figure 7a). Past this point, the cache begins to retain more keys from transactions in the second category, but the performance benefit is limited since these requests rarely lead to transactional hits. In contrast, single-object algorithms use only individual object features to score keys, so they retain hot keys from transactions in both categories. Transactional hit rate increases slowly up to the 55% cache size at which point the cache becomes large enough to begin storing the warm keys from the first transaction category. Since the TAOBench workloads have no temporal patterns, GDSF and LFU provide slightly higher hit rates compared to LRU for all cache sizes.

Similarly, in PG3, DeToX achieves better cache efficiency by not retaining contaminated keys. This workload has a smaller portion of point reads and shorter transactions (50%), so hit rates at smaller cache sizes are lower for all policies. Longer read transactions span up to 60 items and also fall into two categories. There are more transactions in the first category (33% compared to 25% in Product Group 2), so transactional hit rates grow more slowly with respect to cache size since more warm keys need to be cached.

In contrast to the other workloads, PG1 (Figure 7c) consists mainly of point reads and some short read transactions (of size four or smaller), which together make up over 97% of all requests. Our algorithm does not improve transactional hit rate over single-object policies because most hits result from standalone requests and short read transactions to a set of highly popular keys, which single-object algorithms already cache effectively. Throughput increases by 2% (from 82K txn/s to 84K txn/s), and latency decreases by 2% (from 0.61ms to 0.60ms).

ChronoCache [48] has similar hit rates to single-object algorithms since there are no dependencies within transactions for this benchmark; Chronocache simply uses LRU. The middleware layer, which does dependency analysis at run-time, quickly becomes the bottleneck.

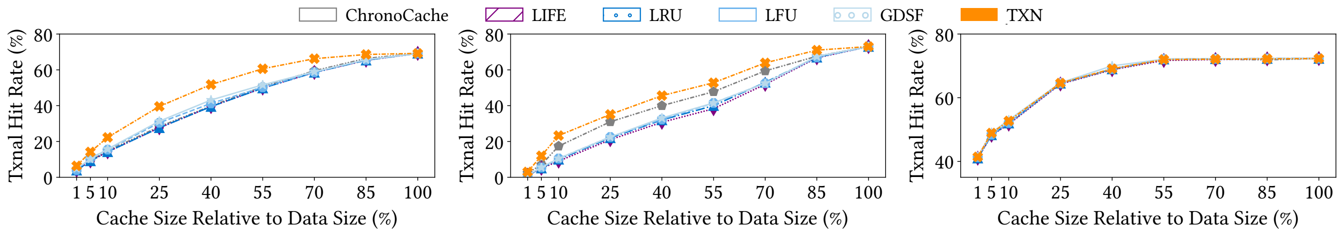
**Epinions.** Epinions centers around user interactions and item reviews. It contains five read-only transactions and four update transactions. Users have both an n-to-m relationship with items (i.e., representing user reviews and ratings of items) and an n-to-m relationship with other users.

DeToX's algorithm provides up to 41% increase in transactional hit rate (Figure 9a), translating into 29% improvement in throughput (from 12K txn/s to 17K txn/s) and 25% decrease in latency (from 6.9ms to 5.5ms). The transactions in Epinions request some group of objects related to a particular user or item (e.g., get all the reviews from one user), so our policy is able to successfully capture the n-to-m relationships of the data with its scoring mechanism. In contrast, the single object policies focus on caching individually popular keys without taking into account correlation between accesses. Since there are no dependencies between or within transactions for this workload, ChronoCache is unable to successfully prefetch objects and the results reflect its eviction policy, LRU.

**SmallBank.** SmallBank consists of requests to the Accounts, Checking, and Savings tables with six transaction types. Its transactions are relatively small, involving four distinct keys at most. Roughly two-thirds of operations are reads. Each customer account is materialized as three separate entries in each table and is accessed with a uniform distribution. There is high correlation between accesses to a customer's row in the Accounts table and the other two rows in the other two tables.

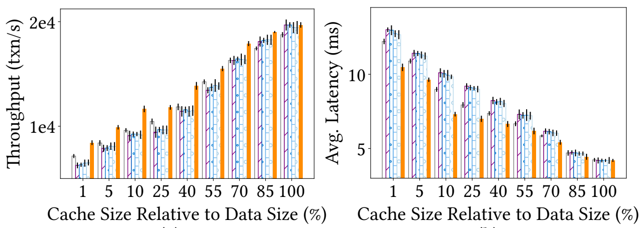
Our algorithm provides up to a 130% increase in transactional hit rate (Figure 9b). The absolute hit rates remain relatively low for smaller cache sizes because of the uniform access distribution to customer accounts. Transactional hit rate increases linearly for all algorithms since more cache space directly results in more hits. DeToX is 1.6x more efficient than the next best-performing algorithm at the 25% cache size.

We observe up to a 28% increase in throughput (from 12K txn/s to 16K txn/s) and 26% decrease in latency (from 6.8ms to 5.4ms) on this workload (Figures 10a and 10b). The long tail in access patterns and short transactions of this workload limit the benefits of our eviction algorithm over single-object alternatives, which all have similar performance.



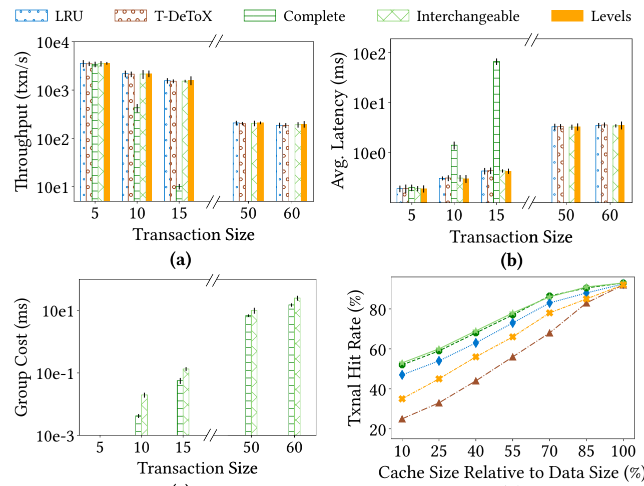
**(a) Epinions** **(b) SmallBank** **(c) TPC-C**

Figure 9: OLTP benchmark results.



**(a)** **(b)**

Figure 10: SmallBank results.



**(a)** **(b)** **(c)** **(d)**

Figure 11: (a),(b),(c). Microbench. 1 (d) Microbench. 2.

**TPC-C.** TPC-C is notably write-heavy and has transactions that can span over 50 items. Its requests tend to fall into two categories: either they access a small set of popular keys (i.e., those in the Warehouse and District tables) or a larger range of keys from a distribution with a long tail (Customer, Item, Stock). Single-object caching algorithms are designed to cache the former while the latter almost always results in transactional misses. For instance, New-Order accesses a key in each of the Warehouse, District, and Customer tables before requesting 10 to 15 items from the Item and Stock tables, which are chosen from a skewed distribution.

Consequently, TPC-C cannot benefit from transactional caching: most transactions access a small set of hot keys that are already in the cache (the object hit rate is >50% with a 10% cache size in Figure 9c) along with a larger set of cold keys that are unlikely to be cached (hit rate grows slowly as cache size increases). Moreover, transactions tend to access keys in quick succession (e.g., once an order is placed, it is then processed, paid for, and delivered), so recency is especially important in this workload. All algorithms incorporate recency in some form, so performance is similar across these policies, with up to 9K transactions per second and 27ms avg latency. DeToX performs on par with single-object policies.

## 8.3 The Need for Dependency Analysis

In this section, we investigate the relative merits of our grouping optimizations. The dependency analysis required for complete groups can impose overheads in two ways: 1) the cost of updating the scores of each key in each group and 2) metadata overhead associated with scoring. Interchangeability can reduce the number of groups that need to be scored, leading to better performance. On the other hand, levels discount unbalanced topologies while T-DeToX, a baseline that scores all keys of a transaction together, ignores dependencies. These simpler policies reduce overhead in some cases but restrict the groups that keys can belong to, leading to worse performance.

**Performance impact**. Microbenchmark 1 intentionally captures the worst-case scenario for grouping. We run a single transaction type with the topology in Figure 5a, and we extend the right branch of the graph for larger transaction sizes. Each read accesses keys uniformly at random among 10M objects. We measure throughput and latency as we increase transaction size up to 60 (equivalent to the largest transactions in the TAOBench workloads). Figures 11a and 11b show that performance for complete groups decreases dramatically as transaction size increases due to the exponential number of complete groups: for a transaction of size 15, over 16K groups have to be scored. In contrast, performance degradation is minimal with interchangeable groups (<5% difference compared to LRU at size 60). There are only a linear number of groups that must be scored with respect to transaction size since all keys in the right branch of this topology are interchangeable. Finally, levels offer similar performance to LRU. Each key can only belong to one level per transaction, so larger transaction sizes do not increase overhead. The run-time CPU overhead of both interchangeable groups and levels is also minimal compared to single-object algorithms for all microbenchmark and previous benchmark workloads.

Moreover, the one-off cost of finding complete and interchangeable groups remains low: transactions of size 60 (with 100K+ groups due to worst-case topologies) require less than five minutes to process (Figure 11c). All benchmark workloads required less than 30 seconds for dependency analysis.

While dependency analysis incurs a static cost, it can lead

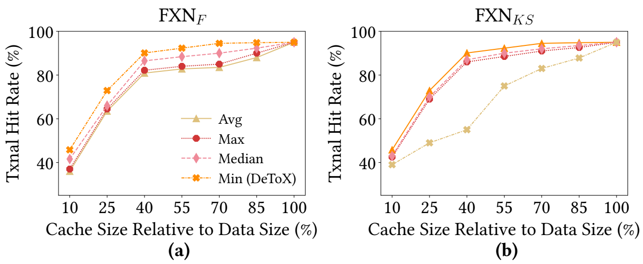



Figure 12: Scoring heuristics

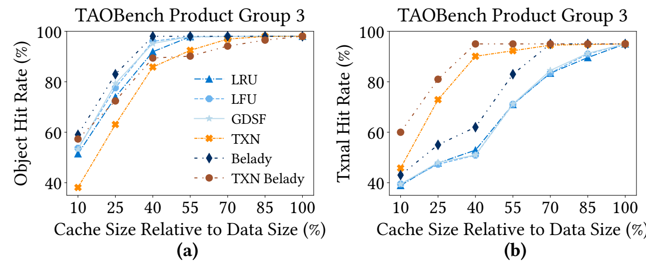



Figure 13: Object (a) and transactional (b) hit rates.

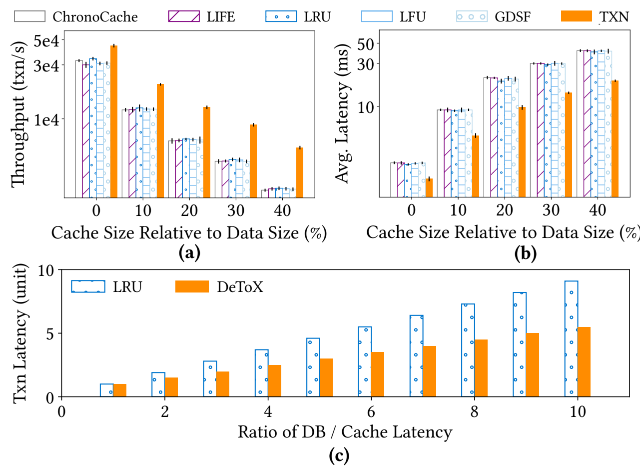



Figure 14: (a), (b). Network latency (c) Simulation results.

to significant benefits compared to more basic forms of grouping (levels and T-DeToX), which ignore some or all dependency information. Microbenchmark 2 quantifies the worst-case scenarios for levels and T-DeToX. We run a single transaction type with the topology in Figure 5a in which the keys in vertices *a* and *c* are hot keys chosen from a Zipfian distribution while keys in *b* and *d* are cold keys chosen from a uniform distribution over 10M objects. Using levels causes keys in *b* and *c* to be scored together. However, keys in *b* are rarely accessed, so the score for *c* is lowered. T-DeToX makes even worse eviction decisions since it scores all keys in *a*, *b*, *c*, and *d* together. Using complete and interchangeable groups would instead cause keys in *b* and *d* to be scored together, enabling the algorithm to capture the fact that caching *c* individually reduces transactional length. We find that complete and interchangeable groups significantly outperform levels (53% increase) and T-DeToX (139% increase) for THR (Figure 11d). Complete and interchangeable groups offer similar performance to LRU since these policies cache keys in *c*, which are frequently accessed.

**Storage overheads.** Metadata overhead in DeToX is low. Our algorithm stores two additional counters (total group score, individual score) per key and a global aging factor for eviction. While prefetching, DeToX stores dependency sets. On TAOBench, additional metadata takes up less than 1% of the cache space. For workloads in which prefetching is more prevalent, metadata overheads increase slightly. For example, in SmallBank, additional metadata grows to 2%. DeToX must store the dependency set associated with each transaction (1.5 keys on average).

## 8.4 Scoring Heuristics

We evaluate different heuristics for calculating instance ($\texttt{FXN}_F$) and aggregate scores ($\texttt{FXN}_{KS}$). DeToX uses the *minimum* frequency of keys in a group for the instance score, and *averages* instance scores to compute an aggregate score (Section 4.2). We measure transactional hit rates for simple functions (average, maximum, median, minimum) in Figure 12 for the PG3 workload (results are similar across workloads).

For assigning key instance scores, we find that, as expected, Min provides the best performance (Figure 12a). Since we only get a transactional hit if *all* keys of a group are cached, the key with the smallest frequency should have outsized impact on the group score. The other functions discount this information and thus perform worse. However, these functions still encode the all-or-nothing property of transactions to some extent since they assign the same instance scores to all keys in a particular group. As a result, we still observe higher hit rates than single-object policies.

Average and Median are the most effective functions for calculating aggregate key score (Figure 12b). Max yields a lower hit rate since it assigns each key the score of its highest-scoring group, but this may not be the most frequent group this key is a member of. Min provides markedly lower performance (up to 64% lower hit rates). Each key is assigned the score of its lowest-scoring group, so most scores converge to the lowest group score (the smallest frequency of any key). As a result, most scores are low and do not differ much.

## 8.5 OHR versus THR

There is a tradeoff between optimizing for latency and for system load. Figure 13 shows the OHR and THR of online algorithms as well as Belady and Transactional Belady (see Appendix). As expected, Belady outperforms other algorithms for object hit rate. Conversely, DeToX and Transactional Belady give some of the lowest object hit rates. However, these two algorithms significantly outperform the other policies for transactional hit rate (and result in better throughput and latency as shown in Section 8.2). While we focus on PG3 here, we find similar results on the other workloads (omitted for lack of space).

The difference between OHR and THR illustrates a tradeoff between reducing I/O bandwidth and optimizing for latency. OHR prioritizes the absolute number of requests that can be

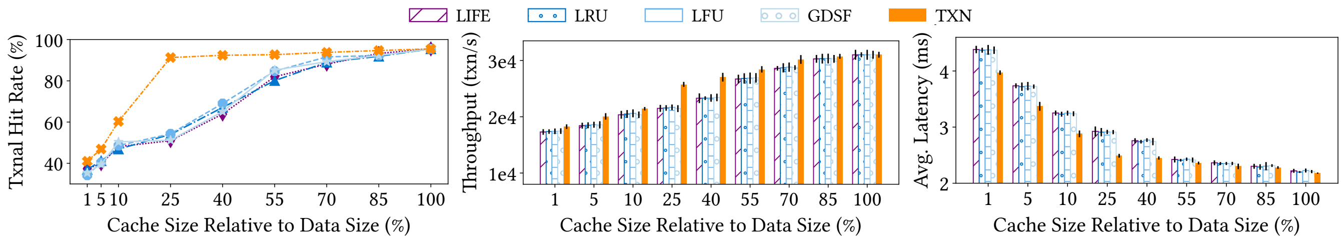


Figure 15: TAOBench Product Group 2 results on TiKV.

served from cache, minimizing requests to disk. In contrast, THR focuses on the number of latency reductions for transactions, leading to lower latency and higher throughput. There are practical motivations for choosing THR as the caching objective: with increasing elasticity from cloud resources, applications often focus on latency optimization for which large wins are possible with DeToX.

### 8.6 Transactional Hit Rate

Transactional hit rate is independent of system specifics; only relative throughput and latency gains differ when cache / system latency changes. We confirm this by 1) varying this ratio (both experimentally and through simulation) and 2) evaluating DeToX with an alternative key-value store, TiKV [4].

**Network latency.** We inject latency between the shim layer and data store to simulate scenarios in which the latter is hosted in a remote cloud region. Figure 14 shows that the performance improvement with DeToX grows as network latency increases. With no additional network latency (0ms), there is a 30% increase in throughput and 29% decrease in latency between DeToX and the best single-object policy for PG3. With a WAN delay of 10ms, there is a 61% increase in throughput and 47% decrease in latency.

**Simulation results.** To illustrate the impact of cache and data store request times, we provide results for the TAOBench PG2 workload. At the 25% cache size, the THR for this workload is around 90% for DeToX and 50% for the other policies (Section 8.2). We vary request times for the cache and the data store (DB), using arbitrary units to represent latency. As we increase the ratio of DB to cache latency in Figure 14c, we find that the difference in request latency between LRU and DeToX increases from 0% to 65% as request times to the data store lengthen.

**Transactional key-value store.** We confirm that both the difference in transactional hit rate and gains in cache efficiency (3.4x) remains identical when executing atop TiKV, demonstrating that these metrics are independent of the setup chosen (Figure 15). In contrast, as TiKV exhibits higher throughput and lower latency than Postgres, throughput and latency gains fall to 19% and 15% respectively.

## 9 Related Work

**Eviction.** There is a wide range of research on efficient caching policies that consider recency or age [34, 45, 55, 75], access frequency [25, 42, 44, 57, 68], the number of unique keys between accesses [15, 53, 61, 69, 78], the variable sizes of objects or pages [5, 27], or combinations of the above [6–8, 13, 14, 16, 19, 21, 30, 52, 54, 60, 83, 86, 105]. Some specialized eviction policies optimize for flash storage [79], adapt to changing workloads [18, 20, 22, 32, 33, 41, 95], or consider network bandwidth and download time for proxy caches [99].

PACMan [12] presents eviction algorithms based on the all-or-nothing property: for jobs that issue tasks in parallel, latency only improves if all parallel tasks are cached. Similarly, existing literature on web caching [8, 19, 97] shows that latency is reduced only when all parts of a page are cached. Transactional hits in DeToX are based on a similar insight. However, DeToX addresses the issue of complex, unbalanced dependency graphs and recognizes keys can be shared across many transactions.

**Prefetching.** Prefetching has been applied extensively for web caching [11, 96]. Past work has focuses on web page analysis [40, 59, 73, 76, 92, 101, 102], which most stand-alone caches do not support [2, 74]. Other research [26, 47, 48, 77] focuses on reducing the latency of query execution using dependency analysis. These works assume that each client issues queries sequentially, so any cache hit can improve latency. Instead, DeToX caches in order to maximize transactional hit. None of these systems provide isolation guarantees or consider how eviction policies should be modified to handle transactions.

**Admission algorithms.** In contrast to eviction algorithms, admission policies decide what is allowed into the cache by enforcing a threshold based on object scores. These algorithms have often been applied alongside eviction policies [8, 22, 43, 56, 67]. While we focus on eviction and prefetching in this work, our grouping and scoring strategies could feasibly extend to admission, which we will explore in future work.

**Cache coherence.** Previous work combining transactions and caching focuses on maintaining isolation guarantees [1, 58, 82, 103]. DeToX ensures serializability while focusing on cache eviction and prefetching.

## 10 Conclusion

We present DeToX, a novel caching system targeting at transactional workloads. DeToX chooses to maximize THR over the traditional object hit rate, using the notion of groups to score keys. Our algorithm improves THR by up to 130% and cache efficiency by up to 3.4x.

## References


[1] Amazon elasticache, November 2021.

[2] Redis, February 2021.

[3] Postgresql, 2022.

[4] Tikv, 2022.

[5] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Edward A. Fox, and Stephen Williams. Removal policies in network caches for world-wide web documents. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, page 293–305, New York, NY, USA, 1996. Association for Computing Machinery.

[6] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Edward A. Fox, and Stephen Williams. Removal policies in network caches for world-wide web documents. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, page 293–305, New York, NY, USA, 1996. Association for Computing Machinery.

[7] Marc Abrams, Charles R Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A Fox. Caching proxies: Limitations and potentials. 1995.

[8] Charu Aggarwal, Joel L. Wolf, and Philip S. Yu. Caching on the world wide web. 11(1):94–107, jan 1999.

[9] Jose Aguilar and Ernst L. Leiss. A web proxy cache coherency and replacement approach. In *Proceedings of the First Asia-Pacific Conference on Web Intelligence: Research and Development*, WI '01, page 75–84, Berlin, Heidelberg, 2001. Springer-Verlag.

[10] Marcos K. Aguilera, Joshua B. Leners, and Michael Walfish. Yesquel: Scalable sql storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 245–262, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Waleed Ali, Siti Mariyam Shamsuddin, Abdul Samad Ismail, et al. A survey of web caching and prefetching. *Int. J. Advance. Soft Comput. Appl*, 3(1):18–44, 2011.

[12] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 20, USA, 2012. USENIX Association.

[13] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *SIGMETRICS Perform. Eval. Rev.*, 27(4):3–11, mar 2000.

[14] Hyokyung Bahn, Kern Koh, S.H. Noh, and S.M. Lyul. Efficient replacement of nonuniform objects in web caches. *Computer*, 35(6):65–73, 2002.

[15] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. USENIX Association.

[16] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, Renton, WA, April 2018. USENIX Association.

[17] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[18] Daniel S Berger. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 134–140, 2018.

[19] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2), jun 2018.

[20] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, Carlsbad, CA, October 2018. USENIX Association.

[21] Daniel S Berger, Sebastian Henningsen, Florin Ciucu, and Jens B Schmitt. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review*, 43(2):57–59, 2015.

[22] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, Boston, MA, March 2017. USENIX Association.

[23] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., USA, 1986.

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

[25] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, Santa Clara, CA, July 2017. USENIX Association.

[26] Ivan T. Bowman and Kenneth Salem. Optimization of query streams using semantic prefetching. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04*. ACM Press, 2004.

[27] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems (USITS 97)*, 1997.

[28] 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. Taobench: An end-to-end benchmark for social network workloads. *Proceedings of the VLDB Endowment*, 15(12):1965–1977, 2022.

[29] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories Palo Alto, CA, 1998.

[30] Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *International Conference on High-Performance Computing and Networking*, pages 114–123. Springer, 2001.

[31] Marek Chrobak, Gerhard J. Woeginger, Kazuhisa Makino, and Haifeng Xu. Caching is hard—even in the fault model. *Algorithmica*, 63(4):781–794, March 2011.

[32] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association.

[33] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, March 2016. USENIX Association.

[34] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[35] The Transaction Processing Performance Council. Tpc-c, 2021.

[36] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 727–743, USA, 2018. USENIX Association.

[37] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. TARDiS. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, June 2016.

[38] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. Transaction repair for multi-version concurrency control. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 235–250, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. volume 7, pages 277–288. VLDB Endowment, 2013.

[40] Josep Domenech, Jose A. Gil, Julio Sahuquillo, and Ana Pont. Using current web page structure to improve prefetching performance. *Comput. Netw.*, 54(9):1404–1417, jun 2010.

[41] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018.

[42] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[43] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Trans. Storage*, 13(4), nov 2017.

[44] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[45] Nicolas Gast and Benny Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. *SIGMETRICS Perform. Eval. Rev.*, 43(1):123–136, jun 2015.

[46] Shahram Ghandeharizadeh, Jason Yap, and Hieu Nguyen. Strong consistency in cache augmented SQL systems. In *Proceedings of the 15th International Middleware Conference on - Middleware '14*. ACM Press, 2014.

[47] Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, Scott Foggo, and Anil Pacaci. Apollo: Learning query correlations for predictive caching in geo-distributed systems, 2018.

[48] Brad Glasbergen, Kyle Langendoen, Michael Abebe, and Khuzaima Daudjee. Chronocache: Predictive and adaptive mid-tier query result caching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2391–2406, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Priya Gupta, Nickolai Zeldovich, and Samuel Madden. A trigger-based middleware cache for orms. In *Proceedings of the 12th International Middleware Conference*, Middleware '11, page 320–339, Laxenburg, AUT, 2011. International Federation for Information Processing.

[50] Jeffrey Helt, Matthew Burke, Amit Levy, and Wyatt Lloyd. Regular sequential serializability and regular sequential consistency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. ACM, October 2021.

[51] Saied Hosseini-Khayat. *Investigation of Generalized Caching*. PhD thesis, USA, 1998. UMI Order No. GAX98-07761.

[52] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, Santa Clara, CA, July 2015. USENIX Association.

[53] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2005.

[54] Shudong Jin and Azer Bestavros. Greedydual⋆ web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications*, 24(2):174–183, 2001.

[55] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[56] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[57] George Karakostas and Dimitrios N Serpanos. Exploitation of different types of locality for web caches. In *Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications*, pages 207–212. IEEE, 2002.

[58] Bryan Kate, Eddie Kohler, Michael S. Kester, Neha Narula, Yandong Mao, and Robert Morris. Easy freshness with pequod cache joins. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 415–428, USA, 2014. USENIX Association.

[59] Bin Lan, Stephane Bressan, Beng Chin Ooi, and Kian-Lee Tan. Rule-assisted prefetching in web-server caching. In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, CIKM '00, page 504–511, New York, NY, USA, 2000. Association for Computing Machinery.

[60] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 134–143, 1999.

[61] Cong Li. Dlirs: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, pages 59–64, 2018.

[62] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, October 2017.

[63] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. nsdi'13, page 313–328, USA, 2013. USENIX Association.

[64] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 313–328, Lombard, IL, April 2013. USENIX Association.

[65] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW theorem and Latency-Optimal Read-Only transactions. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 135–150, Savannah, GA, November 2016. USENIX Association.

[66] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. Performance-Optimal Read-Only transactions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 333–349. USENIX Association, November 2020.

[67] Bruce M. Maggs and Ramesh K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM Comput. Commun. Rev.*, 45(3):52–66, jul 2015.

[68] Dhruv Matani, Ketan Shah, and Anirban Mitra. An o (1) algorithm for implementing the lfu cache eviction scheme. *arXiv preprint arXiv:2110.11602*, 2021.

[69] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association.

[70] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 453–468, Boston, MA, March 2017. USENIX Association.

[71] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, Broomfield, CO, October 2014. USENIX Association.

[72] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 517–532, Savannah, GA, November 2016. USENIX Association.

[73] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1155–1169, 2003.

[74] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[75] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[76] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, jul 1996.

[77] Mark Palmer and Stanley B Zdonik. *Fido: A cache that learns to fetch*. Brown University, Department of Computer Science, 1991.

[78] Sejin Park and Chanik Park. Frd: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[79] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cflru: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, page 234–241, New York, NY, USA, 2006. Association for Computing Machinery.

[80] Francisco Perez-Sorrosal, Marta Patiño Martinez, Ricardo Jimenez-Peris, and Bettina Kemme. Elastic si-cache: Consistent and scalable caching in multi-tier architectures. *The VLDB Journal*, 20(6):841–865, dec 2011.

[81] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 279–292, USA, 2010. USENIX Association.

[82] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 279–292, USA, 2010. USENIX Association.

[83] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions on networking*, 8(2):158–170, 2000.

[84] Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. Rolis. In *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, March 2022.

[85] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[86] David Starobinski and David Tse. Probabilistic methods for web caching. *Performance evaluation*, 46(2-3):125–137, 2001.

[87] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 283–297, New York, NY, USA, 2017. Association for Computing Machinery.

[88] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. ACM, October 2021.

[89] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr. Sharma, Arvind Krishnamurthy, Dan R. K. Ports, and Irene Zhang. Meerkat. In *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, April 2020.

[90] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Cobra: Making transactional Key-Value stores verifiably serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 63–80. USENIX Association, November 2020.

[91] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. Ad hoc transactions in web applications: The good, the bad, and the ugly. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 4–18, New York, NY, USA, 2022. Association for Computing Machinery.

[92] Na Tang and V. Rao Vemuri. An artificial immune system approach to document clustering. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, page 918–922, New York, NY, USA, 2005. Association for Computing Machinery.

[93] The H-Store team. Smallbank benchmark, 2013.

[94] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. Contention-aware lock scheduling for transactional databases. *Proc. VLDB Endow.*, 11(5):648–662, jan 2018.

[95] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with {ML-based}{LeCaR}. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[96] Jia Wang. A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, oct 1999.

[97] Justin Wang, Benjamin Berg, Daniel S. Berger, and Siddhartha Sen. Maximizing page-level cache hit ratios in largeweb services. *SIGMETRICS Perform. Eval. Rev.*, 46(2):91–92, jan 2019.

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

[99] Roland P Wooster and Marc Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 29(8-13):977–986, 1997.

[100] Yingjun Wu, Chee-Yong Chan, and Kian-Lee Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1689–1704, New York, NY, USA, 2016. Association for Computing Machinery.

[101] Lifang Xu, Hongwei Mo, Kejun Wang, and Na Tang. Document clustering based on modified artificial immune network. In Guo-Ying Wang, James F. Peters, Andrzej Skowron, and Yiyu Yao, editors, *Rough Sets and Knowledge Technology*, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[102] Qiang Yang, Haining Henry Zhang, and Tianyi Li. Mining web logs for prediction models in www caching and prefetching. KDD '01, page 473–478, New York, NY, USA, 2001. Association for Computing Machinery.

[103] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system. *Proc. VLDB Endow.*, 11(10):1289–1302, jun 2018.

[104] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. volume 35, New York, NY, USA, December 2018. Association for Computing Machinery.

[105] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue replacement algorithm for second level buffer caches. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, Boston, MA, June 2001. USENIX Association.

## A Appendix

We describe the NP-Hardness of the optimal offline transactional caching problem through a reduction. We begin by providing intuition for how and why traditional optimal offline caching policies fail to translate to transactional caching.

### A.1 Transactional Belady

We straightforwardly adapt Belady's optimal caching policy [17] to the transactional context by defining *Transactional Belady*, a caching policy that evicts keys that result in *transactional hits* furthest in the future. While this extension is intuitive, it does not offer optimal performance even for flat, uniformly-sized transactions that access equally-sized objects, as we prove below.

Consider the execution trace in Figure 16 with cache capacity of 5. All transactions access three keys, either all from set $S_1$: $\{t,u,v,t',u'\}$ or set $S_2$: $\{x,y,z,x',y'\}$. $T_1$ and $T_2$ access only keys from the former group, while $T_3$ and $T_4$ access only keys from the latter. $T_5$ and $T_6$ access keys from $S_1$ and overlap in $v$, while $T_7$,$T_8$,$T_9$ overlap in $x'$,$y$,$z$ from $S_2$. Transactional Belady evicts keys that will yield a transactional hit furthest in the future. After $T_3$'s execution, the algorithm thus evicts $x$,$y$,$z$ as they would first yield a hit at $T_7$ while the other keys would lead a hit at $T_5$ and $T_6$. A similar reasoning leads the algorithm to evict $x',y',z'$ after $T_4$ executes. This strategy yields two transactional hits (for $T_5$ and $T_6$). Unfortunately, evicting $x$,$y$,$z$ after $T_3$ is the wrong decision. Keeping all keys of set $S_2$ in the cache yields three transactional hits $T_7$, $T_8$, $T_9$. As a result, Transactional Belady achieves only two transactional hits, while an optimal caching policy would achieve three.

| T | Keys accessed | Cache state | Optimal cache state |
|---|---|---|---|
| 1 | $t,u,v$ | - | - |
| 2 | $t',u',v$ | $t,u,v$ | - |
| 3 | $x,y,z$ | $t,u,v,t',u'$ | - |
| 4 | $x',y',z$ | $t,u,v,t',u'$ | $x,y,z$ |
| 5 | $t,u,v$ | $t,u,v$, $t',u'$ | $x,y,z,x',y'$ |
| 6 | $t',u',v$ | $t,u$, $v,t',u'$ | $x,y,z,x',y'$ |
| 7 | $x,y,z$ | $t,u,v,t',u'$ | $x,y,z$, $x',y'$ |
| 8 | $x',y,z$ | $t,u,v,t',u'$ | $x$, $y,z,x'$, $y'$ |
| 9 | $x',y',z$ | $t,u,v,t',u'$ | $x,y$, $z,x',y'$ |

Figure 16: Non-optimality of Transactional Belady

Transactional Belady does not account for shared keys across transactions. It caches $S_1$, which is shared across two transactions, instead of keys in set $S_2$, which is shared across three transactions. Belady assumes that a cache hit closer in the future is always as valuable as a cache hit further out. This assumption holds when a single cached object provides a single object hit but breaks down when keys are shared across transactions. In these cases, an equal number of cached objects can produce varying numbers of transactional hits.

### A.2 Optimal Offline Transactional Caching is NP-Hard

We demonstrate that the optimal offline transactional caching problem (TxPolicy) is NP-Hard through a reduction to the variable-sized caching problem, CACHING(FAULT, OPTIONAL), introduced in [31].

We first provide intuition for our reduction. A page hit is only possible if the entire page is present in the cache, regardless of its size. The objective of CACHING(FAULT, OPTIONAL) is to minimize the number of page faults, or the number of pages accessed and missed. We convert each page of size X into a transaction without dependencies that accesses X operations. Therefore, there is only a transaction hit when the entire transaction is in the cache. This transforms CACHING(FAULT, OPTIONAL) into an easier version of TxPolicy with two simplifying assumptions: 1) all transactions will use unique keys, so that retaining a key in the cache from any single transaction provides no benefit to any other transaction and 2) there are no logical dependencies. If an optimal offline transactional caching policy exists, then through this reduction, we have the optimal policy for CACHING(FAULT, OPTIONAL).

We now formally describe CACHING(FAULT, OPTIONAL) from [31]. CACHING(FAULT, OPTIONAL) asks,

> Given a set of pages $p_1,\ldots,p_k$ with sizes SIZE($p_1$),...,SIZE($p_k$), request sequence $r_1,\ldots,r_m$ $\in \{p_1,\ldots,p_k\}$, cache size $C$, and cost bound $F$, is there a replacement policy that serves $r_1,\ldots,r_m$ with cache size $C$ and incurs a total fault cost at most $F$?

A *fault* is incurred when $r_i \notin C_i$, where the FAULT parameter states that each fault has cost 1. The OPTIONAL parameter requires that $\forall i > 1, C_i \subseteq \{C_{i-1} \cup r_i\}$; informally, the caching policy does not have to admit the most recent page.

We formally define the offline transactional caching problem, based on our formalisms from Section 3.

**Definition 10** *(Offline transactional caching policy). An offline transactional caching policy is a function P that takes a sequence of transactions $T_1, T_2, \ldots, T_m$, cache size n, and outputs a sequence of cache states $C_1, C_2, \ldots, C_m$, with the following restrictions:*

1. $C_1 = \varnothing$.
2. $\forall i > 1$, $C_i \subseteq \{C_{i-1} \cup T_{i-1}\}$.

TxPolicy asks,

> Given a set of transactions $T_1$, ... , $T_m$, cache size $C$, is there an offline transactional caching policy that serves $T_1,\ldots,T_m$ with cache size $C$ and incurs at most $F$ transactional misses? We define *transactional misses* as the number of $i$ where $T_i \not\subseteq C_i$, or the number of transactions that cannot be served from cache.

**Theorem 1.** *The optimal offline transactional caching problem is NP-Hard.*

*Proof.* We reduce CACHING(FAULT, OPTIONAL) to TxPolicy through the following polynomial-time reductions. Each page $p_i$ is reduced to a transaction $T_i$. SIZE$(p_i)$ new tables are created per transaction, each with only one key. Let $X$ be one such table. A read operation on the sole key of that table $x \in X$ is inserted into the transaction $T_i$. There are no logical dependencies. Cache size $C$ is preserved. The maximum fault cost $F$ is converted to the maximum number of transactional misses. If there exists a policy solving the offline transactional caching problem, run it with these parameters. Its output is the output to the CACHING(FAULT, OPTIONAL) problem. CACHING(FAULT, OPTIONAL) is NP-Hard; therefore, the offline transactional caching problem is NP-Hard. □