Competitive Consistent Caching for Transactions

Citation for published version:
An, S, Cao, Y & Zhao, W 2022, Competitive Consistent Caching for Transactions. in Proceedings of 38th IEEE International Conference on Data Engineering. International Conference on Data Engineering, Institute of Electrical and Electronics Engineers (IEEE), pp. 2154-2167, 38th International Conference on Data Engineering 2022, Kuala Lumpur, Malaysia, 9/05/22. https://doi.org/10.1109/ICDE53745.2022.00207

Digital Object Identifier (DOI):
10.1109/ICDE53745.2022.00207

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of 38th IEEE International Conference on Data Engineering

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Competitive Consistent Caching for Transactions

Shuai An  
University of Edinburgh  
shuai.an@ed.ac.uk

Yang Cao  
University of Edinburgh  
yang.cao@ed.ac.uk

Wenyue Zhao  
University of Edinburgh  
wenyue.zhao@ed.ac.uk

Abstract—This paper studies cache policies for transactional caches. Different from conventional caches that focus on latency, transactional caches are primarily used to augment database systems and improve their transaction throughput by offloading read load onto the cache. A read transaction commits on the cache only if it is a consistent cache hit, i.e., all of its reads see a consistent view of the database. We prove that conventional cache policies are not competitive for transactions. We then show that for the large class of batching-based transaction systems, one can break the theoretical performance barrier of conventional cache policies via transaction consistency aware cache policies, although it is NP-complete to find the optimal ones. As a proof, we develop a consistent cache policy that is theoretically competitive under common cache schemes. To further exploit batching, we propose to reorder transactions within batches while guaranteeing that each transaction sees data values with bounded staleness. Using benchmarks and real-life workloads, we experimentally verify that our policy improves the transaction throughput of Memcached atop HBase by 126.95% on average, up to 479.27% higher than existing cache policies adopted for transactions.

I. INTRODUCTION

Data caches such as Memcached [1], NCache [2] and Redis [3] have found increasing popularity in large-scale data systems, e.g., TAO [4], [5], LiveJournal [6], and MediaWiki [7]. Different from web caches that bring data closer to computation to reduce latency, data caches are typically lightweight key-value stores that are collocated with databases. They are primarily used to improve the scalability and throughput of databases for large volumes of concurrent requests. By shifting reads from the database, data caches have shown effective in alleviating database load and improving the performance of the combined system for highly concurrent workloads [4], [5], [8].

Driven by emerging applications from social network [4], [5] to online retailers [9] that generate transactional reads [10], [11], [12], [13], [4], [14], [15], [16], [17], [18], [19], [20], [21], [22], there has been recent effort in developing transactional caches which add transactional guarantees to data caches oblivious to transaction consistency, e.g., TxCache [8], Sundial [23], COPS-SNOW [11], Facebook [5], IQ [24] and T-Cache [9]. Transactional cache deals with read transactions that demand consistency: all data items seen by a transaction in the cache are consistent, i.e., they come from a consistent snapshot of the database. As shown in Fig. 1, transactional cache combines cache invalidation and transaction commit protocols, such that a transaction commits only when it is a consistent cache hit.

While transaction commit protocols have been developed to assure that transactions committed on the cache are consistent [8], [23], [9], little attention has been paid to cache policies that maintain transactional caches in response to transaction workloads and database updates. Indeed, existing transactional caches simply adopt conventional policies, e.g., LRU, which completely overlook transaction consistency when updating caches. This raises a number of questions. How does transaction consistency play a role in cache performance for transactions? How should it be accounted in the design of cache policies to improve transaction throughput? How well do conventional cache policies work for transactions? Are they optimal? Indeed, Facebook has even listed the design of cache policies as a key challenge for their Memcached-augmented database [25].

In this study, we answer all these questions.

Competitive analysis. We first model transaction caching with consistent cache schemes, with which we then study how conventional cache policies are adopted for transactional caches and what limitations they have when used for transactions. We consider three cache schemes w.r.t. commonly used cache invalidation protocols, e.g., PURGE, FRESH or BAN [26].

We prove that conventional cache policies, when adopted for transactional caches, do not perform well. In particular, we show that no conventional cache policies are competitive under any of the three cache schemes, where a cache policy is competitive if it is a bounded approximation of the optimal offline policy [27].

Limitation. The analysis tells us that conventional policies like LRU and its variants can perform poorly with transactional caches; moreover, it is theoretically intractable to find better alternatives. The root cause of the impossibility results is their pure online nature, as they were originally developed for CPU or web caches that focus on latency of individual read request and are oblivious to transaction consistency. Instead, transactional caches deal with both read and write, and target transaction throughput. Writes generate multiple versions of the same object, which imposes additional challenge to ensure that a read transaction commits over cache only when it sees a consistent view of the underlying database, while the cache may be inconsistent. The throughput-oriented objective and consistency requirement, intertwined with cache invalidation between database and cache, make conventional cache policies ill-fitted and not competitive for transactional caches.

Fig. 1: Cache-augmented data store
Therefore, to break the theoretical performance barrier of transactional caches, one has to go beyond conventional cache policies and explore characteristics pertaining to transactional database systems. We show that this is attainable for transaction systems that use batching, e.g., partitioning-based systems [28], [29], [30] and deterministic database (cf. [31]), which are gaining increasing popularity in both multicore and distributed transaction processing. Their key characteristic is that transactions are processed in batches [29], [32] and are often in a pre-defined total order [31] for higher throughput. Such a batching-based design has been adopted by systems from both academia [32], [33], [34], [35], [29], [28], [36], [37], [30], [38], [39] and industry [40], [41], [42], [43]. We show that the use of batching in these systems also benefits cache policy design for transactions.

### Batch consistent caching
To demonstrate this, we study consistent cache policies for transactional caches atop batching-based databases. In contrast to conventional policies such as LRU and LRU-k [44], they take into account transaction consistency when evicting cached items upon overflows. Moreover, they explore batching and the pre-determined order of transactions, to defy the non-competitiveness of conventional policies.

We first show that, via batching, there exist cache policies with which transactional caches can gain competitive performance that is beyond the capability of conventional cache policies. However, it is rather non-trivial to fully explore the potential of batching due to the need to uphold transaction consistency. Indeed, we prove that it is NP-complete to find optimal transaction policies that maximize the transaction throughput by offloading reads from the database to the cache.

Nonetheless, we develop characterizations of optimal cache policies for transactions, based on which we design linear to logarithmic-time policies that are theoretically competitive for transactional caches atop batching-based transaction databases with major cache invalidation protocols. Moreover, when reads in the transactions access values of the same size, e.g., integers or fixed-size objects, our policies become optimal.

### Staleness-bounded transaction reordering
To further exploit the benefit of batching, we propose to incorporate transaction reordering into cache policies. The idea is to reorder transactions in the batch so that more transactions can commit on the cache. However, naively reordering transactions would cause transactions to read inordinately stale versions of data values that are not supposed to be seen if they commit in the original order.

To this end, we constrain the scope of reordering with a staleness parameter $s$ such that each read in the reordered sequence is guaranteed to see a version of its requested value that has staleness bounded by $s$. This guarantees that, although we change the ordering of the transactions in the batch, transactions still see a reasonably “live” view of the database via transactional caches. By controlling the staleness bound, we allow flexible trade-off between liveness and cache performance for transactions. We show that it is NP-complete to find an optimal reordering. Nonetheless, we develop effective heuristics that further improve our policies via staleness-bounded reordering.

### Prototype
We develop TCache, a prototype that implements our cache policies and optimizations. Using real-life workload and benchmarks, we found the following. (1) Compared to existing cache policies adopted for transactions, TCache improves the throughput of Memcached atop HBase by 126.95% on average, up to 479.27%. (2) Transaction reordering is effective when the staleness bound $s$ is set to as small as 4, improving TCache by 46.78%; moreover, it still improves TCache by 21.45% even staleness is not allowed, i.e., $s$ is set to 0.

### Contributions & organization
To summarize, we initiate the study of cache policies for transactional caches. Our main contributions are listed as follows:

- We prove that existing cache policies are not competitive for transactions, and make a case for batching in caching (§III).
- We settle down the complexity of batch consistent caching and characterize competitive policies for transactions (§IV).
- We develop competitive and even optimal cache policies for transactional caches under common cache schemes (§V).
- We study staleness-bounded transaction reordering to further improve transactional caches (§VI).
- We develop TCache that improves transaction throughput of Memcached with existing policies by 126.95% (§VII).

See full version [45] for all proofs and additional experiments.

### Related Work
We categorize related work as follows.

Transaction data caches. Data caches [1], [2], [3] have been extensively used to improve the throughput of real world database systems [4], [6], [7], by shifting read load from database to cache layer. They augment conventional database systems with flexible, scalable and efficient auxiliary memory to serve increasing volumes of read workload. Updates in the databases are propagated to the caches via cache invalidation protocols [26]. Due to their lightweight design, they naturally lack many heavy database operations like transaction algorithms. Hence, database systems augmented with data caches cannot maintain transaction consistency. To alleviate this, transactional caches have been proposed [8], [9], [23], [11], [24], which use distributed transaction commit protocols that are aware of cache presence to ensure committed transactions are consistent [9], [11], [24] or nearly consistent [8], [23].

Our work complements existing research on transactional caches. (1) Instead of developing yet another cache-aware transaction protocol, we focus on cache policies for transactional caches. (2) We show that simply adopting conventional cache policies suffers fundamental limitations. We develop policies specifically for transactional caches, with performance guarantees that are beyond the capacity of conventional cache policies. (3) Our study implies that existing systems have not exploited the full potential of transactional caches due to restricted cache schemes and policies, and can benefit from this study.

**Batching.** Batching-based transaction systems have been actively studied in academia [33], [34], [35], [32], [29], [28], [36], [37], [30], [38], [46], [39] and practiced in industry [40], [41], [42], [43]. They execute transactions in batches collected via, e.g., a time-window [29] or an append-only log [32],
and are often with a pre-defined order (cf. [31]). It also allows transaction partitioning to reduce conflicts and simplifies transaction protocols. In this work, we study transactional caches on top of such batching-based systems, which allow us to break the theoretical performance barrier of conventional cache policies for transactions by taking the advantages of batching.

**Cache policies.** Caching has been well-studied for decades [47], [48], [49], [44], [50], [51]. Cache policies decide how to evict cache content upon cache overflows. Conventional policies such as LRU, LFU, FIFO and their variants are online in nature, by deciding which item to evict for requests coming online, one at a time. This makes them easy to use in virtually all cache scenarios. There has also been the work on offline caching [52], [53], [54], [55], [56], [57], which makes eviction decisions with the request sequence known beforehand. When cached items are of unit size (a.k.a. paging), it is well known that the linear time Belady’s rule [55], [56] is the optimal policy. The complexity of generic caching has been an open problem for decades [52] until the NP-completeness result of Chrobak et al. [53], [54].

Our work differs from these as follows. (1) In contrast to conventional cache that deals with singleton reads, we study cache policies for transactions, where cache decisions are made per transaction. (2) Writes are often treated as a minor extension of conventional cache policies. Indeed, we are not aware of any cache analyses that take into account writes. However, writes and cache invalidation play a central role when caching transactions. (3) Cache policies for transactions have to deal with cache-side consistency, which is heavily intertwined with cache invalidation that is not considered by conventional policies. (4) Instead of maximizing cache hit rate, we aim to maximize transaction throughput. This, together with transaction consistency, makes cache policy design a much harder problem. Indeed, it is already NP-complete for uni-size transactions, as opposed to trivially in PTIME for conventional policies [55]. (5) Obsolete reads for transactions do not exist for conventional cache and can be coNP-complete to identify.

**Reordering.** Request reordering has also been studied in web caching, by ordering online requests within a sliding window to improve cache hit rate [58], [59], [60], [61]. Different from the context of web caching, we consider the reordering of transactions instead of read requests. Moreover, we restrict inordinate implications of reordering via a staleness parameter.

## II. CACHING FOR TRANSACTIONS

We start with preliminaries of transactional databases (§II-A) and consistent caching for transactions (§II-B).

### A. Preliminaries

**Cache-augmented databases.** We consider cache-augmented database systems, where an external data cache is added to the database to serve transactions. The cache is typically a lightweight distributed memory that is faster and easier to scale out than a full-fledged database. We focus on look-aside cache as illustrated in Fig. 1, which is adopted by e.g., Facebook [5] and Twitter [62], and is proven effective for read-heavy workloads particularly. For such systems, writes are committed to the database and are propagated to the cache via cache invalidation.

While external caches enable faster reads and higher scalability, their lightweight design brings new challenges when serving transaction workloads. One immediate consequence is that the augmented system may lose transaction correctness guarantees that a database system is supposed to have. To this end, transactional caches [4], [5], [9], [24], [8], [23], [11] have been developed such that database systems extended with caches would retain the desired transaction guarantees.

The research on transactional caches has been primarily focusing on lightweight protocols to ensure that read transactions commit on the cache only when they are guaranteed correct. Instead of developing yet another transaction protocol for caches, in this work we study the design of cache policies for transactional caches. To delve into the problem, we start below with necessary definitions of transaction correctness over cache, referred to as cache-side transaction consistency [23], [8], [11].

**Cache-side transaction consistency.** We abstract a database $D$ as a set of pairs $\{(q_1, v_1), \ldots, (q_n, v_n)\}$, where $q_i$ identifies an item $v_i$ (we also refer to $v_i$ as the value of $q_i$). A read query (or simply read) $r(q_i)$ fetches the value $v_i$ of $q_i$ and a write $w(q_i)$ updates the value $v_i$. When executing write transactions over $D$, we yield a series of different versions (i.e., snapshots) of $D$, say $D[0], \ldots, D[k], \ldots$, where $D[i]$ is updated from $D[i-1]$ when a write transaction commits. In practice, $(q_i, v_i)$ could be a key-value pair for NoSQL or (id, tuple) for relations; a read $r(q_i)$ (or simply $q_i$) retrieves the value pertaining to the item indexed by $q_i$. To remain focused on transactional caches, we do not consider the internal structure of reads $q_i$, i.e., they are atomic operations instead of complex queries which, in practice, can typically be represented as a set of atomic reads.

A read transaction $R$ is simply a set of reads. We consider the general case where $R$ may read values of different sizes, e.g., when they refer to objects of varying sizes. As a special case, when all values read by $R$ have the same size, e.g., integers or tuples from the same relation, $R$ is called an uni-size transaction.

A cache $C$ is a set of pairs of $D$ such that each pair $(q_i, v_i)$ is taken from some version of $D$ (say, $D[k]$), determined by when the item $q_i$ is cached in $C$. That is, $(q_i, v_i) \in D[k]$ but it is possible that $(q_i, v_i)$ is not in the current $D$, i.e., the value $v_i$ of $q_i$ in $C$ may not always be the current version depending on how updates to $q_i$ are propagated to $C$ via cache invalidation. We say that item $q_i$ is cached if $C$ caches its value $v_i$.

**Consistent cache hit.** A read transaction $R$ is a consistent cache hit over cache $C$ if (a) all items read by $R$ are cached in $C$ and (b) there exists version $k$ of $D$ such that the cached values to items of $R$ in $C$ are from snapshot $D[k]$. Intuitively, $R$ is a consistent cache hit over $C$ if $R$ is a cache hit over $C$ as usual but additionally the cached values for the items read by $R$ must form a consistent view of the database at some point of time, i.e., they exist in a snapshot $D[k]$ of $D$.

It is widely adopted by transactional caches that a read transaction commits over the cache only when it is a consistent cache hit [8], [11], [4], [5], [9], [24]. While the cache may be incon-
sistent as it contains values from different database snapshots, cache-side consistency guarantees that the answer to any read transaction computed over cache is sensible in that it was correct at some point of time over the database. Depending on how cache invalidation is implemented, transactions committed over the cache may not always see the current database snapshot.

B. Consistent Cache Schemes for Transactions

We present consistent cache schemes to capture how transactional caches operate and state the consistent caching problem.

As shown in Fig. 1, we consider the case where transactions are coming online to the application server, where read transactions are processed on a fixed-size cache \( C \) which holds data up to a total size of \( b \). Write transactions are executed at the database server, which then propagates committed changes to the cache via cache invalidation protocols.

A read transaction \( R \) may have three possible cases over \( C \):

(a) \( R \) is a consistent cache hit on \( C \);

(b) \( R \) is an inconsistent cache hit, i.e., all reads of \( R \) are cached in \( C \) but are not from the same snapshot of \( D \); or

(c) \( R \) is a cache miss, i.e., \( R \) reads a item \( q \) not in \( C \).

Consistent cache schemes. A consistent cache scheme specifies how \( C \) is maintained and used to process transactions, along with the backend database. We consider three schemes below, depending on how writes are propagated from the database to cache \( C \) and how read transactions are processed over \( C \).

PCC (Pessimistic consistent cache scheme). It employs the PURGE cache invalidation protocol [26]. When the database is updated, a PURGE message is sent to the cache with a list of updated items. Upon receiving the message, \( C \) purges all referenced items immediately. When processing a read transaction \( R \) over \( C \), the system checks whether all the reads of \( R \) are cached in \( C \), if so it answers \( R \). Otherwise, it fetches the missing items from the database, caches them if there is room in \( C \), and answers \( R \). If \( C \) has no room to hold the new items, i.e., a cache overflow occurs, \( C \) has to evict sufficient number of cached items so that it has the free space to cache the new ones.

ACC (Active consistent cache scheme). It is compatible with the REFRESH invalidation protocol [26]. Under ACC, the cache \( C \) works the same as under PCC when processing a read transaction. When the database is updated, a REFRESH message is sent to the cache, which triggers \( C \) to update its outdated items referenced in the message by refetching from the database.

Observe that under both PCC and ACC the cache \( C \) is always consistent as all the cached items are in their latest version because of the cache invalidation protocols they employed.

LCC (Locally consistent cache scheme). It works with the BAN invalidation protocol [26]. Under LCC, when the database is updated, a BAN message is sent to the cache with a list of all updated items. In contrast to PCC and ACC, when the cache receives a BAN, it does not modify \( C \) as PURGE and REFRESH do; instead, it only adds the referenced cached items to a ban list, recording that they have just been updated. Hence, \( C \) may be inconsistent under LCC and one has to ensure cache-side transaction consistency when answering read transactions.

By the turn a read transaction \( R \) is processed over \( C \) under LCC, the cache decides whether \( R \) is a consistent cache hit.

(1) If \( R \) is a consistent cache hit, it answers \( R \) using \( C \) directly.

(2) If \( R \) is an inconsistent cache hit or a cache miss, it selects a subset \( R' \) of \( R \), fetches \( R' \) from the database to \( C \), and answers \( R \) over the updated \( C \) when \( R \) becomes a consistent cache hit.

Example 1: Consider a sequence \( \ell \) of transactions as shown in Fig. 2, where each \( R_i \) (resp. \( W_i \)) is a read (resp. write) transaction. Assume that initially the cache \( C \) holds \( q_0, \ldots, q_4 \), \( C \) has unlimited size, and all items in \( D \) are of unit size. Note that write transactions generate 3 versions of \( D \), say \( D[0] \) before \( W_1 \), \( D[1] \) after \( W_1 \) and \( D[2] \) after \( W_3 \).

(1) Under PCC. \( R_2 \) is a cache miss as \( W_1 \) purges \( q_1 \) from \( C \); similarly for \( R_4 \) and \( R_5 \) due to \( W_3 \). Hence any cache policy has to read at least 4 items \( (q_1, q_3, q_4, q_5) \) to answer them with \( C \).

(2) Under ACC. \( R_2 \) is a consistent cache hit as \( q_1 \) in \( C \) is re-fetched after \( W_1 \); similarly for \( R_5 \). \( R_4 \) is a cache miss. Therefore, it takes 5 reads in total to re-fetch \( q_0, q_1, q_3 \) and \( q_4 \) for \( W_1 \) and \( W_3 \) and fetch \( q_5 \) for \( R_4 \).

(3) Under LCC. When \( R_2 \) is processed, \( q_1 \) is stale in \( C \) since there is a newer version in \( D \) updated by \( W_1 \). However, \( q_1 \) and \( q_2 \) still form a consistent cache hit for \( R_2 \); they are from the same snapshot of \( D \), i.e., \( D[0] \). Hence \( R_2 \) can be answered consistently using \( C \) only without reading \( D \). Instead, \( R_4 \) is a cache miss since \( q_5 \) is not in \( C \) when \( R_4 \) is processed. Hence \( R_4 \) reads \( q_3 \) from \( D \); however, \( q_3 \) is in \( D[0] \) while the newly fetched \( q_5 \) is from \( D[2] \), hence \( q_5 \) will also be re-fetched by \( R_4 \). For \( R_5 \), it looks like a cache hit since both \( q_4 \) and \( q_5 \) are in \( C \) by its turn; however, they are not consistent due to \( W_3 \). Indeed, \( q_4 \) in \( C \) appears in both \( D[0] \) and \( D[1] \), but not in \( D[2] \), while \( q_5 \) in \( C \) appears only in \( D[2] \). Hence, \( R_5 \) is a cache hit but inconsistent, which requires to re-fetch \( q_1 \). Hence, an ideal cache schedule under LCC reads just 3 items from the database in total.

Example 1 shows that the performance of caching varies over different schemes, depending on the cache invalidation methods implemented. While many transaction caches employ PCC by default, e.g., [5, 8], ACC and LCC are more often used in web caches, e.g., Varnish [63]. Example 1 also illustrates that transaction caches may benefit more from ACC or LCC by using an alternative cache invalidation protocol.

Cache policies. A central problem in caching is the design of cache replacement policies. We study consistent cache policies for transactional caches \( C \) that decide, for each read transaction \( R \) in a sequence \( \ell \) of transactions, if \( R \) is a cache miss that causes an overflow over \( C \), which cached items in \( C \) to evict in order to free space for \( R \) and make it a consistent cache hit.
Here a cache overflow happens when \( R \) is a cache miss over \( C \) and \( C \) has no available space to cache the missing reads in \( R \). The sequence of actions decided for each transaction in \( \ell \) forms a consistent cache schedule for \( \ell \) over \( C \).

Intuitively, consistent cache policy are algorithms that operate on transactional caches under one of the cache schemes, to maintain cache data and serve transactions. As remarked in Section I, conventional cache policies do not work well with writes. Worse still, they cannot observe consistency for transactions.

**The TCP problem.** In light of the new challenges, in the sequel we focus on the design and analysis of consistent cache policies, stated as the transactional caching problem (TCP) below.

**INPUT** A cache \( C \) of size \( b \), and a sequence \( \ell \) of transactions, **OUTPUT** A consistent cache schedule \( P \) for \( \ell \).

**OBJ** Minimize \( \text{cost}(P) \), the total number of reads in the transactions of \( \ell \) to be fetched from the database.

Intuitively, TCP is to design a consistent cache policy for transactional caches that maximize transaction throughput by minimizing accesses to the backend database, \( i.e., \text{cost}(P) \). By minimizing \( \text{cost}(P) \), we offload as much read load as possible from the database on to cache so that the entire system could serve more concurrent requests and increase throughput. Here \( \ell \) contains both read and write transactions. In practice write transactions appear as cache invalidation messages to the cache. The quality of cache policies depends on the consistent cache schemes (\( i.e., \text{PCC}, \text{ACC} \) and \( \text{LCC} \)) employed. In addition, it is also related to how the cache policy observes \( \ell \) when generating schedules. To start with, we assume that cache policies are pure online, \( i.e., \) they make cache decisions for each read transaction \( R \) in \( \ell \) without knowing subsequent transactions in \( \ell \). This captures conventional cache policies that are designed for \( i.e., \) web requests coming online one by one.

### III. The Case for Batching

In this section, we prove that conventional cache policies cannot be competitive for transactions and make a case for batching to break the barrier of conventional policies.

**Competitiveness.** Following the study of online algorithms [27], we use competitive ratio to analyze cache policies.

Consider a consistent cache policy \( P \) for problem TCP. We denote by \( \text{cost}(P, \ell) \) the cost of the cache schedule generated by \( P \) for \( \ell \). We say that \( P \) is \( \alpha \)-competitive \( (\alpha \geq 1) \) if for any sequence \( \ell \) of read and write transactions, we have \( \text{cost}(P, \ell) \leq \alpha \cdot \text{cost}(\text{OPT}, \ell) \), where \( \text{OPT} \) is the offline optimal policy, \( i.e., \) \( \text{cost}(\text{OPT}, \ell) \) is the cost of the optimal schedule for \( \ell \) generated by \( \text{OPT} \) that knows the entire \( \ell \) beforehand. A policy is not competitive if it is not \( \alpha \)-competitive for any \( \alpha \). Intuitively, if \( P \) is \( \alpha \)-competitive for some \( \alpha \geq 1 \), it means \( P \) is comparable to the best policy one can hope for.

**Analysis.** We next analyze the competitiveness of conventional cache policies when adopted for transactional cache. We consider online policies captured in the class \( L_M \) of Marker’s algorithms (cf. [47], [27], [64]), which includes popular policies like LRU and its variants. It is well known that all policies in \( L_M \) are competitive in the conventional cache setting [64]. We study them for transactional caches.

We first focus on LRU, which is adopted by virtually all transactional caches, \( e.g., \) [8], [25], [5]. We start with a positive result that shows why LRU is so popular. Consider sequences \( \ell \) with uni-size transactions only, \( i.e., \) each read is of size 1 in every transaction of \( \ell \). Let \( m \) be the size of the cache \( C \).

**Proposition 1:** Under \( \text{PCC} \), if \( \ell \) consists of uni-size transactions only, then LRU is \( m \)-competitive.

Proposition 1 shows why LRU is often used with the \( \text{PURGE} \) protocol. However, for general cases and other cache schemes, all policies in \( L_M \), including LRU, are not competitive.

**Theorem 2:** (1) There exists no cache policy in \( L_M \) that is competitive under \( \text{PCC}, \text{ACC} \) or \( \text{LCC} \).

(2) There exists no cache policy in \( L_M \) that is competitive under \( \text{ACC} \) even for uni-size transactions only.

Theorem 2 tells us that existing cache policies are not competitive in most of the cases and it is impossible to improve the situation if we stay with conventional cache policies.

**Implication.** The impossibility results motivate us to step back and rethink the objective of transactional caches. One distinct characteristic of transactional caches that stands out is the type of workload, \( i.e., \) transactions. Instead of one read request at a time, a read transaction contains multiple reads to be served at the same time. Databases are also updated by write transactions. Moreover, transactions are typically coming concurrently in a large volume and the backend database system aims to maximize the overall throughput with the help of transactional caches. The bottleneck of the database is then often the computation capacity instead of the latency of each read.

**The case for batching.** This naturally gives rise to the idea of designing cache policies that can explore the characteristic of concurrent transactions and the underlying database systems. To this end, we specifically focus on the class of batching-based transaction databases (\( e.g., \) deterministic databases [31]), which batches transactions and executes each batch in a pre-determined order. It has found advantageous in both distributed [32], [33] and multi-core databases [46], [29], by exploring offline transaction workload partitioning.

While conventional cache policies are not competitive, via batching we can have competitive policies for transactions.

**Proposition 3:** There exists consistent cache policies that are competitive for batching-based databases under \( \text{PCC} \) and \( \text{ACC} \); moreover, they become optimal for uni-size transactions.

Proposition 3 justifies the benefits of incorporating transaction batching, a technique that has already been exploited by transaction databases, for caching. Below we characterize consistent cache policies for batched transactions in §IV, based on which we then give a constructive proof of Proposition 3 in §V.
IV. THE FOUNDATION OF BATCH CONSISTENT CACHING

In this section, we lay the foundation of consistent caching for batched transactions. We first settle down its complexity (§IV-A) and then develop characterizations (§IV-B).

A. Complexity

We study the complexity of problem TCP (recall §II-B) when the sequence ℓ is a transaction batch that is part of the input known to the cache policies. This is to some extent similar to the conventional caching in the offline setting where ℓ is known beforehand. However, the presence of transactions and the consistency requirement make it much more challenging.

For example, it is well-known that finding the optimal cache schedule for offline paging is in PTIME via Belady’s rule [55], which evicts the cached item whose next request time is furthest in future. However, it is no longer optimal for transactions.

Example 2: Continue Example 1. Assume that W₃ also includes q₀ and the cache C has a size limit of 5. Following W₁, ..., W₅, assume that there are 4 new transactions W₆ = {q₀}, R₇ = {q₀, q₅}, R₈ = {q₃, q₄} and R₉ = {q₁}. Then by the turn of R₄, it incurs a cache overflow under both ACC and LCC since q₅ is not in C and C is full at the time.

(1) First consider ACC. By Belady’s rule which is to evict from C the item whose next read time is the most distant in the sequence at the time, we need to replace q₁ in C with the new item q₅ for R₄. The total number of reads required is 8: q₅ for R₄, q₁ for R₉, and all items in the write transactions except q₅ of W₃. However, if we replace q₀ in C with q₅ for R₄, we do not need to update q₀ upon W₆, yielding a better cache schedule of 7 reads in total. This shows that the presence of writes and invalidation impairs the optimality of Belady.

(2) LCC is even more intriguing. Belady’s rule would work the same as under ACC. Hence, (a) by R₄₁, it replaces q₁ in C with q₅. Meanwhile, q₃ is re-fetched as q₃ is from D[2] while q₃ is from D[0], causing an inconsistent cache hit; (b) for the same reason, it re-fetches q₄ for R₅ and q₀ for R₇; (c) R₈ is a consistent cache hit; (d) R₉ is a cache miss and it fetches q₁. That is, a total of 5 reads are needed for Belady’s rule.

Now consider the cache schedule that replaces q₀ (instead of q₁) in C with q₅ for R₄. Then for R₄, R₅ and R₇, it acts exactly the same as above. However, both R₈ and R₉ are now consistent cache hits, witnessed by D[2] and D[0], respectively. Hence the schedule incurs 4 reads in total under LCC, better than Belady’s rule. This shows that under LCC transaction consistency further complicates the optimality of cache policies.

This shows that optimal offline cache policy (Belady’s rule) is no longer optimal for transactions, even for uni-size transactions. Indeed, caching transactions is much more challenging.

Complexity. Consider the decision problem of TCP.

Theorem 4: (1) TCP is NP-complete under all three schemes.

(2) When all the reads in the transactions are of unit size,
(a) TCP becomes in PTIME under both PCC and ACC;
(b) however, it remains NP-hard under LCC.

B. Characterizations

We next develop characterizations for optimal consistent cache policies for batched transactions.

Consider a sequence ℓ of transactions in the batch. For convenience, we assume that transactions are mapped to time indices in [1, |ℓ|], where |ℓ| is the number of transactions in ℓ.

Example 3: Continue Example 2. Under LCC, both q₀ and q₄ in the cache C at time 4 (i.e., when answering R₄) will never be used by any transactions after time 4 (i.e., after R₄ in ℓ). Hence, an optimal cache schedule for ℓ would evict or re-fetch q₀ and q₄ after time 4 upon cache overflows.

Similarly, under PCC and ACC, q₀ at time 4 in C is not helpful for any transactions after R₄ since q₀ due to W₆. □

Example 3 shows that, due to intertwined cache invalidation and transaction consistency, a cached read can be useless for any transactions even it appears again. Hence, any optimal cache schedule must evict these cached reads first upon a cache overflow. Below we formally capture such reads, based on which we develop characterizations for optimal cache schedules.

Obsolete reads. Consider a sequence ℓ of batched read and write transactions. Let C be the cache buffer at the current time t. Denote by ℓₜ the suffix of ℓ starting from time t, i.e., the part of ℓ that is yet to be processed.

A read r(q) (or simply q) is obsolete at time t for ℓ if
(a) q is in the cache C at time t; and
(b) for any consistent cache schedule for ℓₜ over C, the cached q will not contribute to the committing of transactions in ℓₜ, i.e., q will have been re-fetched from the database to answer any of the transactions in ℓₜ that contain r(q).

That is, when r(q) is obsolete, by the time the immediate next read transaction R in ℓₜ that contains r(q) is processed, in any consistent cache schedule (i) q must be stale in the cache, (ii) R is an inconsistent cache hit or a cache miss, and (iii) q must be updated in order to make R a consistent cache hit. In other words, the cached copy of q will be replaced when answering R, under any consistent cache schedule for ℓₜ over C. Note that, q can be useful (i.e., not obsolete) even when q is stale when R arrives and R is a cache miss or inconsistent cache hit.

Intuitively, if q is obsolete in cache at the current time, it will for sure be updated by any consistent cache schedule before being used to answer a transaction. Hence, keeping q in the cache is by no means useful for the transactions. Therefore, at all times obsolete reads should be evicted whenever the cache buffer needs to squeeze space for caching new transactions. As will be shown shortly, this observation allows us to characterize competitive and even optimal cache schedules for transactions.

Characterizations. We first show that it is essential to evict obsolete reads at all times. We say that a consistent cache schedule
P is optimal for $\ell$ if for any other schedule $P'$, $\text{cost}(P, \ell) \leq \text{cost}(P', \ell)$. Denote by $P[i]$ the cache replacement decision made by $P$ for read transaction $R$ of $\ell$ that arrives at time $i$.

**Lemma 5:** Under PCC, ACC and LCC, for any consistent cache schedule $P$, if $P$ is optimal for $\ell$, then at any time $i \in [1, |\ell|]$, after applying $P[i]$ the cache will contain no obsolete reads. \hfill $\square$

Lemma 5 shows that the key to the design of a competitive or even optimal cache policy is the capability of identifying obsolete reads for each read transaction in $\ell$ and evicting them with the highest priority when an overflow occurs. This motivates us to study the identification of obsolete reads for any given sequence of batched transactions, under PCC, ACC and LCC.

1. **PCC** and ACC. The following confirms that determining obsolete reads under PCC and ACC can be done in linear time.

Consider cache buffer $C$, sequence $\ell$ of read and write transactions, read $r(q)$ that is cached in $C$ at time $t$.

**Proposition 6:** Under PCC and ACC, $r(q)$ is obsolete for $\ell$ at time $t$ if and only if the next read time of $q$ after $t$ is later than the time of the next write transaction with $w(q)$. \hfill $\square$

By Proposition 6, under both PCC and ACC, it is in $O(K_C)$-time to tell whether a cached read $q$ is obsolete at any given time, where $K_C$ is the number of reads that cache $C$ can hold.

2. **LCC.** Due to the possibility that read transactions can still be well served over cache $C$ even if $C$ is inconsistent, it becomes far more intriguing to decide obsolete reads under LCC.

**Theorem 7:** It is coNP-complete to decide whether $r(q)$ is obsolete under LCC, even $\ell$ consists of uni-size transactions. \hfill $\square$

V. A Competitive Cache Policy for Transactions

In light of Theorem 4, any practical cache policy for TCP has to be approximate. Nonetheless, based on Lemma 5, we develop a unified policy below for TCP that works under all three cache schemes, with provable competitiveness guarantees.

**The OFF policy.** The policy, denoted by OFF (Observe First then First-in-the-furthest), is presented in Algorithm 1. The essential idea is to, upon a cache miss or an inconsistent cache hit, first evict all those cached reads that are obsolete at the time, and then process reads in the current transaction one by one, using an extended Belady’s rule when a cache overflow occurs. Since Belady’s rule only works for conventional cache setting where reads are of unit size and processed one at a time, we need to extend it to cope with the varying sizes of reads in the form of transactions with consistency requirements.

OFF builds upon the techniques of conventional caching with varying sizes [52]. It first classifies all reads in the transactions of $\ell$ by their sizes. For each transaction $R$, if $R$ is an inconsistent cache hit or cache miss, OFF refreshes staled reads in the cache w.r.t. write transactions prior to $R$. If $R$ is a cache miss, OFF fetches the missing new reads and then answers $R$ using cache consistently; if a cache overflow occurs even all obsolete reads have been evicted, OFF evicts one or two cached reads from each class in the cache, which will guarantee that there are sufficient room to cache the new reads that incur overflow.

We next present OFF in details. Denote by $\Sigma$ the set of all distinct items read or written by transactions in $\ell$. OFF first classifies all reads in the transactions into $\lfloor \log k \rfloor + 1$ classes such that class $i \in [1, \lfloor \log k \rfloor + 1]$ contains reads of size in the range $[\min_{q \in \Sigma} |q|, 2^{i-1} \cdot \min_{q \in \Sigma} |q|, 2^i]$, where $k = \max_{q \in \Sigma} |q|$, which is the size of the read item queried by $q$. We say that $q$ is a $i$-read if its size falls in the $i$-th class $[\min_{q \in \Sigma} |q|, 2^{i-1} \cdot \min_{q \in \Sigma} |q|, 2^i]$.

It then processes read and write transactions of $\ell$ one by one as shown in Algorithm 1. When processing transaction $R_i$ at time $t$, it first checks whether it is a consistent cache hit over cache $C$ (via isCCH; see [45] for more); if so, it answers (commits) $R_i$ (lines 1-2). Otherwise, it first identifies all cached reads that are obsolete at time $t$ via findOB (line 4; more below) and removes them from $C$ (line 5). It then processes reads in $R_i$ one by one; if read $q$ in $R_i$ is in $C$ but outdated due to write transactions before $t$, it updates $q$ in $C$ with the latest version from the database (line 7); otherwise if $q$ is not yet cached, it checks whether there is enough room in $C$ to accommodate $q$ (line 8), and caches $q$ if so (line 12); otherwise, it evicts 2 reads (1 read if all reads are of unit size) from each class whose next appearance time in $\ell$ is the furthest in the future (line 9-11).

**Procedure** findOB. We have already sorted out findOB under PCC and ACC (Proposition 6 of §IV-B). We next focus on LCC.

In light of Theorem 7, it is practically infeasible to find exactly all the obsolete reads under LCC. Nonetheless, we present an efficient design of findOB under LCC, shown as Algorithm 2, that warrants each and every read it identifies is obsolete for certain. In other words, it is a sound method to efficiently identify obsolete reads under LCC with certainty: with it OFF will never evict a good read by mistakenly recognizing it as obsolete.

For any given time $t$, findOB identifies reads in $C$ that are obsolete for $\ell$ at $t$. The key idea is to execute $t_i$, the suffix of $\ell$ starting from time $t$, via a modified dry run of OFF over a temporary cache buffer $C'$ of infinite size, during which findOB marks reads in $C$ that can be determined obsolete for certain. More specifically, it first initializes $C'$ the same as $C$ (line 1).
**Algorithm 2:** The findOB Procedure

**Input:** Sequence $\ell$ of transactions, cache $C$ at current time $t$.

1. $C' \leftarrow C$; $\ell_t \leftarrow (\ell_1; +\infty)$; // create a new buffer $C'$ without size limit
2. while $\ell_t \neq \text{nil}$ and $C$ has unmarked reads do
   3. $R_i \leftarrow t, \text{pop}();$ // $R_i$: the transaction at time $t \geq i$

   4. if isCH($R_i, C'$) is true then
      5. foreach $q \in R_i$ do
         6. if $q$ is in $C$ and unmarked then mark $q$ as safe in $C$;
      7. else
         8. if $R_i$ is a cache miss in $C'$ then
            9. foreach $q \in R_i$ and $q \notin C$ do add $q$ to $C'$;
      10. if isCH($R_i, C'$) is true then
       11. foreach $q \in R_i$ do
           12. if $q$ is unmarked in $C$ then mark $q$ as safe in $C$;
       13. continue // life$_Q(q)$: duration at which $q$ cached at $t$ is also in the database.
       14. $H_t \leftarrow$ a max-heap of all $q \in R_t$ by lower endpoints of life$_Q(q)$;
       15. $H_u \leftarrow$ a min-heap of all $q \in R_t$ by upper endpoints of life$_Q(q)$;
       16. $q_1 \leftarrow H_t, \text{pop}();$ $q_2 \leftarrow H_u, \text{pop}();$
       17. /* for life$_Q(q_1) = [l_{b_1}, u_{b_1}],$ life$_Q(q_2) = [l_{b_2}, u_{b_2}]$ */
       18. while $l_{b_1} > u_{b_2}$ do
           19. replace life$_Q(q_2)$ with life$_Q(q_1)$ and update $H_t$ and $H_u$ accordingly;
           20. if $q_2 \in C$ and unmarked then mark $q_2$ as obsolete in $C$;
           21. $q_1 \leftarrow H_t, \text{pop}();$ $q_2 \leftarrow H_u, \text{pop}();$
       22. foreach $q$ in $R_i$ that is also in $C$ do mark $q$ as safe if not marked
       23. return all reads in $C$ that are marked as obsolete;

It then examines transactions in $\ell_t$ one by one against $C'$, where $\ell_t$ is the suffix of $\ell$ starting from time $t$. It marks reads in the transactions as either safe or obsolete if they are also in $C$. The process terminates when all reads cached in $C$ are marked (or all reads in $\ell_t$ are examined; lines 2-21), and findOB returns those marked as obsolete in the end (line 22).

Each time findOB pops the front transaction remained in $\ell_t$, say $R_i$ at time $i$ (line 3). It checks whether $R_i$ is a consistent cache hit over $C'$ (line 4). If so, all reads in $R_i$ that are also in $C$ are marked as safe if they have not been marked yet (lines 5-6). Otherwise, $R_i$ is either a cache miss or an inconsistent cache hit over $C'$. If $R_i$ is a cache miss over $C'$, findOB expands $C'$ by including reads of $R_i$ that are missing in $C'$, so that $R_i$ becomes a consistent or inconsistent cache hit. If $R_i$ is now a consistent cache hit over $C'$, findOB marks all reads of $R_i$ that are also in $C$ but are not yet marked as safe and moves on to the next transaction in $\ell_t$ (lines 10-13).

If $R_i$ remains an inconsistent cache hit over $C'$, it iteratively examines pairs of reads in $R_i$ and see whether they are inconsistent with each other and “refreshes” (dry-run) one of them to make them consistent if they are not (lines 14-20); all refreshed reads are marked obsolete in $C$ if they are also contained in $C$ but not marked (line 19). findOB does this by maintaining two heaps of reads in $R_i$: one is a max-heap $H_t$ that sorts reads by the lower end points of their life$_Q$ (life span in the database snapshots; see Algorithm 2) ranges in descending order and the other is a min-heap $H_u$ that sorts reads by the upper endpoints of their life$_Q$ ranges in increasing order (lines 14-15). Each iteration, findOB pops the top read from $H_t$ and $H_u$, denoted by $q_1$ and $q_2$, respectively. It checks whether $q_1$ and $q_2$ have overlapping lifespans. If so, $q_2$ must have to be “refreshed” in order to make $R_i$ a consistent cache hit (lines 17-18). It marks $q_2$ as obsolete if it is in $C$ and is not yet marked (line 19). The iteration terminates if the pair of the head reads in $H_t$ and $H_u$ become consistent (line 20). findOB marks all those reads in $R_i$ that also in $C$ but are not yet marked as safe (line 21).

**Proposition 8:** Under LCC, for any read $q$ found by findOB($C, \ell, t$), $q$ must be an obsolete read in $C$ at time $t$ for $\ell$. $\blacksquare$

**Complexity.** OFF generates a consistent cache schedule within $O(|K_C| + |\ell| \cdot T_{\text{findOB}})$-time under all three schemes, where (a) $|K_C|$ is the number of classes that $C$ is divided into (i.e., $K_C = |\log k| + 1$; recall that OFF groups reads in $C$ into classes), (b) $|\ell|$ (resp. $|\ell'|$) is the total number of transactions (resp. reads) in $\ell$, and (c) $T_{\text{findOB}}$ is the complexity of findOB($C, \ell, t$).

Under PCC and ACC, $T_{\text{findOB}}$ is in $O(|K_C|)$-time; under LCC, $T_{\text{findOB}}$ is in $O(h \cdot \log c)$-time, where (i) $h$ is the number of transactions in $\ell_t$, which is the shortest sub-sequence of $\ell$ that starts from time $t$ and covers reads in $C$, and (ii) $c$ is the number of reads a transaction may have, (typically small, e.g., 5).

**Optimization.** As an optimization of findOB, we also parameterize findOB for LCC such that one can specify an upper bound $h_0$ for the length of $\ell_t$ (i.e., $h$). We found that $h_0 = 10$ can cover most of the obsolete reads in practice, which makes findOB almost in constant time. Similarly for PCC and ACC.

**Competitiveness.** We next study the guarantees of OFF. Recall the notion of competitiveness in §III. In particular, a policy $P$ is optimal if it is 1-competitive, i.e., it always generates consistent cache schedule of the lowest cost for each and every sequence $\ell$.

**Theorem 9:** Under both PCC and ACC, OFF

1. is $2 \log k$-competitive; and
2. is optimal when the reads are of unit size. $\blacksquare$

Theorem 9 is constructive proof of Proposition 3. It shows that, in contrast to conventional policies that are not competitive for transactions, OFF is competitive and even optimal. As will be shown in §VII, by evicting obsolete reads OFF does consistently achieve higher throughput than conventional ones.

**VI. TRANSACTION REORDERING FOR CACHING**

In this section, we further improve the effectiveness of transaction caching via transaction reordering, which is naturally enabled and supported by batched transactions.

**Implications of transaction reordering.** There are two implications of intra-batch transaction reordering on the performance of the entire system: (a) an improved performance and (b) a less “accurate” results for the read transactions. Here (a) is to some extent natural since one can expect to improve cache-side transaction committing rate by improving data locality across adjacent transactions via reordering. However, (b) is somehow easily to be overlooked when employing batching and reordering.

**Example 4:** Continue Example 2. Assume initially $C$ contains $\{q_0, q_1, q_2, q_3, q_4\}$. Then for the transaction sequence $\ell$ in Example 2, OFF generates an optimal schedule that incurs 5 reads under PCC. Consider a reordering $\ell' = (R_2, R_8, R_9, R_4, R_5,$}
Example 4 shows that reordering can help offload transactions to the cache for better performance. While it may look straightforward in the example to find the best reordering as we have a perfect cache C that can hold almost all items (5 out of 6), it is however nontrivial in the generic case as, e.g., swapping two transactions may improve the data locality of some items while worsening the others. Furthermore, with reordering transactions may see a stale view of the database that is different from what they would observe in the original order. For many applications such as stock trading [65], manufacturing [66] and warehouses [67], transactions are time-sensitive and stale reads are tolerated only when read values have bounded staleness.

Staleness-bounded reordering. This motivates us to study transaction reordering subject to a controlled bound on the “staleness” of the views that the transactions observe, stated (informally) as the staleness-bounded reordering problem (SBRP):

INPUT a sequence ℓ of transactions, a staleness bound s (to be formalized below), a cache C of size b.

OUTPUT a reordering ℓ′ of ℓ.

CONSTRAINT each read in ℓ′ observes a view of the database of at most s-stale w.r.t. what it would observe in ℓ.

OBJECTIVE minimize cost(ℓ′), where P is the optimal cache size.

Intuitively, SBRP is to find a reordering (ℓ′) of ℓ to maximize the benefit of caching for ℓ, while ensuring that the transactions, if committed in the order of ℓ′, would observe a view that has a bounded staleness distance from the one they would observe in ℓ (we will formally define the notion of staleness shortly).

Note that, SBRP is not a restriction of the simpler reordering setting without bounded staleness, since the latter is a special case of SBRP where the bound s is large, e.g., greater than |ℓ|. Instead, it enables the option to apply intra-batch reordering in a controlled way by specifying appropriate staleness parameter s.

Staleness. To complete the statement of SBRP, we define the notion of staleness below. Denote by RVer(R,r[q],ℓ) the number of writes w[q] in transactions prior to ℓ in R. The staleness of read r[q] of R in the reordering ℓ′ of ℓ, denoted by stale(R,r[q],ℓ′), is defined as |RVer(R,r[q],ℓ′) − RVer(R,r[q],ℓ)|. Intuitively, RVer(R,r[q],ℓ′) measures the version of q-values R,r[q] would see if transactions were committed according to the “natural” order of ℓ, and stale(R,r[q],ℓ′) quantifies the difference between the versions that R,r sees in ℓ′ and ℓ. We say that r[q] of R is at most s-stale in ℓ′ if stale(R,r,ℓ′) ≤ s.

Challenges. The problem is challenging. First, the staleness bound imposes nontrivial restrictions on the search space of valid reordering. Indeed, naive heuristics of grouping similar transactions to improve cache locality may end up with reordering that never satisfies the staleness bound specified by the user. Moreover, even for uni-size transactions and constant staleness bounds, e.g., s = 1 is as small as 1, it is already intractable to find best reordering of ℓ for caching, as shown below.

Theorem 10: SBRP is NP-complete under PCC, ACC or LCC. It is NP-hard even s = 1 and ℓ consists of uni-size transactions.

Algorithm ReO. Despite the intractability, we develop an efficient reordering heuristic, denoted by ReO, that always (a) returns a reordering ℓ′ of ℓ satisfying user-specified staleness bound s and (b) improves cache performance for transactions. Below we sketch the idea of ReO (see [45] for more details).

1. It first creates a bipartite graph Gℓ((V1, V2), E), where V1 and V2 are the two vertex sets and E ⊆ V1 × V2. Each vertex v in V1 encodes a write transaction W in ℓ and each v′ ∈ V2 encodes a read transaction R. It orders V1 in the order as in ℓ. An edge (W, R) is in E if putting R next to W will not break the staleness bound s.

   Note that since the order of write transactions (V1) is fixed, the staleness of a read R,r[q] depends on write transactions prior to R, irrelevant to other read transactions.

2. It then computes a matching M ⊆ E of Gℓ by iteratively processing vertices of V1; each iteration it picks v ∈ V1 with the maximum degree and assigns all vertices v′ ∈ V2 connecting to it; once v′ is assigned to v, it also removes all edges from E that connects v′ and other vertices of V1.

3. The match M of step (2) assigns each read transaction of ℓ to exactly one gap between write transactions. ReO then iteratively reorders read transactions in each gap to maximize cache performance. In each iteration, it picks |ℓ|/k transactions with reads that are mostly least recently requested on average. Here k is a tunable constant that determines the ordering granularity.

Complexity. ReO can be implemented in O(|ℓ| log |ℓ|)-time, where |ℓ| is the total transaction size of ℓ (see [45] for details).

VII. IMPLEMENTATION AND EXPERIMENTAL STUDY

We experimentally evaluate the effectiveness of OFF and its optimization for caching transactions. We start with a prototype that implements OFF. We then present our evaluation findings.

Prototype. We have developed TCache, a prototype that implements OFF and its optimizations on top of Memcached for caching transactions. TCache inherits workflow of batching-based transaction systems (e.g., [29], [31], [39]). For each batch Bi of transactions, TCache generates a cache schedule for it using OFF (§V) and the reordering optimization (§VI). It instructs Memcached to use the generated schedule instead of the default LRU to serve read transactions in Bi. TCache also pipelines cache scheduling and transaction execution: when the underlying database is executing transaction batch Bi, it collects and generates cache schedule for the subsequent batch Bi+1. In this way, the entire system can take further advantages of the batching-based execution model of the underlying databases for caching in a non-blocking way, i.e., cache schedule generation does not block transaction execution.

Evaluation Plan. Using benchmark and real-life datasets, we evaluate (1) the effectiveness of OFF in improving transaction throughput, (2) the feasibility of pipelining, (3) the effectiveness...
of transaction reordering optimization, and (4) the robustness of cache performance over transaction batches of varying sizes.

**Experimental Settings.** We use the following settings.

**Datasets.** We used two benchmarks and one real-world trace.

(1) **YCSB benchmark.** We used the built-in core workload B with a 95/5 reads and writes mix of the YCSB benchmark [68], consistent with typical workloads that transactional caches target in practice [4], [25], [69]. It has the below parameters.

- \( \theta \): the Zipfian distribution parameter used by YCSB to emulate skewed access patterns, ranging from 0.4 to 1.2 (0.4 by default). A higher \( \theta \) means more skewed access distribution.

- \( dsize \): the size of YCSB database. We varied \( dsize \) in the range [10GB, 30GB] by varying its number of keys from 10M to 30M, consistent with previous studies [29], [36], [70].

(2) **TCBench.** To further evaluate cache policies for more diverse transactions, we also implemented a micro-benchmark TCBench that generates YCSB-compliant workloads with varying characteristics. It is controlled by the following parameters.

- \( \theta \): TCBench generates items in the transactions using Zipfian, similar to YCSB built-in workloads. It varies the Zipfian parameter \( \theta \) in the range of [0.4, 1.2] (0.6 by default).

- \#-items: number of distinct items in the transaction batch. It varies in the range of [200, 1000] and is set to 600 by default.

- write\%: the percentage of write transactions, which varies from 5\% to 25\% and is set to 5\% by default.

Given a configuration of the parameters, TCBench randomly generates a sequence \( \ell \) of read and write transactions that conforms to the parameters. The size of the items in the transactions follows the Facebook's Memcached distribution [69].

(3) **Real-life dataset (Wiki).** We also used Wiki, a 14-day Wikipedia CDN trace collected in 2018 [71]. We picked a slice of \( 10^8 \) items, grouped them into transactions, each with 8 items. Each item has size specified by its “request object size” property. Writes are randomly distributed in Wiki with probability write\% \( \in [1\%, 20\%] \) (1\% by default).

**Baselines.** We also compared \( \text{OFF} \) with existing methods.

- \( \text{Cache policies.} \) We compared \( \text{OFF} \) with existing cache policies. To do this, we configured TCache with major cache policies adopted for transactions as competitors. Following existing transactional cache protocols (e.g., [8]), a transaction commits over the cache if it is a consistent cache hit; if it is a cache miss, it fetches missing items from the database and retries; if it is an inconsistent cache hit, it will abort; aborted transactions will retry by re-fetching its requested items. Upon cache overflows when updating the cache, cached items are replaced according to the specific cache policies used by TCache.

- \( \text{LRU: the default cache policy of Memcached [1].} \)

- \( \text{LRU-k: a modern variant of LRU based on LRU-k [44].} \)

- \( \text{Belady: the optimal offline policy for uni-size reads [55].} \)

- \( \text{LRU-txn: a variant of LRU that evicts the least recently used cached transaction (instead of item) upon cache overflows.} \)

- \( \text{Belady-txn: a variant of Belady that evicts transactions at a time upon cache overflows, similar to LRU-txn.} \)

- \( \text{OFF}^{-}\): a variant of \( \text{OFF} \) that does not evict obsolete items first upon overflows, following the paging policy in [52].

- \( \text{OFF}^{0\%}: \) a plain version of \( \text{OFF} \) that does not employ ReO.

(b) **Reordering policies.** We also compared the reordering optimization of \( \text{OFF} \) (ReO in §VI) with the below baselines:

- Random: transactions are randomly ordered;

- Readfirst: read transactions first then write transactions; and

- Writefirst: write transaction first then read transactions.

Note that these reordering policies do not comply with the staleness bound \( s \) that \( \text{OFF} \) (ReO) is subject to (recall §VI). This is in favour of the baselines as they have more room to exploit transaction reordering for better throughput than ReO does. By default, ReO is enabled for \( \text{OFF} \) with staleness bound \( s = 0 \), i.e., the most restrictive setting with no staleness allowed.

**Configuration.** The experiments were run on AWS EC2 [72]. We used HBase v2.2.4 on a m5.24xlarge EC2 instance as the database server and Memcached v1.5.6 on 60 m5.8xlarge instances as cache nodes with TCache deployed; each cache node also serves as an application server node that receives/generates transactions and gathers results. The cache size accounts for an \( \alpha_{\text{size}} \) fraction of all the read/write items in the transaction workloads, where \( \alpha_{\text{size}} \) varies from 20\% to 40\% (40\% by default). To measure the impact of parallelism, we also varied the total number of transaction threads (\( \#-\text{txn} \)) on the cache nodes from 600 to 1400 (1000 by default). All nodes are in the same EC2 region connected by 10 Gigabit intranet.

Following the practice of deterministic databases and batch-based transaction systems, we process transaction workloads in batches, each consists of 500 to 5000 transactions per thread (1000 by default). To accurately evaluate the effectiveness of cache policies via transaction throughput, we keep the system saturated with a steady stream of transaction batches; each test was run for at least 1 hour and was repeated for 3 times.

**Experimental Results.** We next report our main findings.

**Exp-1: Throughput.** We first evaluated the effectiveness of all cache policies in improving transaction throughput. We compared the throughput of the entire system with different cache policies over all three datasets. When varying a parameter, all the other parameters were set to the default.

(1) **Overall performance.** We first compared the throughput of all methods with the baseline that does not use cache (nocache). As shown in Fig. 3a, caching does improve the overall throughput, for all cache policies, e.g., over YCSB, \( \text{OFF}, \text{OFF}^{-}\), Belady and LRU improve nocache by 7.01, 4.79, 4.84 and 2.80 times, respectively. This also confirms previous studies on the benefit of cache for transactions.

We also compared the average throughput of all cache policies with varying workload parameters (\( \theta \) and write\%). Key results are reported in Figures 3b-3f; see [45] for more). We found that, with \( \text{OFF} \) the overall throughput is consistently the best among all. Over YCSB under LCC, the throughput with
(b) Belady-txn
\(s_{32}/s_{76}/s_{82}/s_{85}\)

To understand why \(\text{OFF}\) allows consistent cache hit over possibly stale items. This improvement of \(\text{LCC}\) are effective in identifying them. This is reflected by the larger findOB #-read of read load is shifted to Memcached nodes due to more out at HBase. We found that with \(\text{OFF}\), the number of obsolete items accounts for most of the improvement for \(\text{OFF}\) over the baselines. However, for workloads with higher write\%, the effectiveness of ReO reduces noticeably and evicting obsolete items accounts for most of the improvement for \(\text{OFF}\).

(4) ReO optimization. We found that \(\text{OFF}\) (\(\text{OFF}\) without ReO) also consistently outperforms all other baselines, e.g., over TCache its throughput is 103.81%, 114.23%, 28.50%, 126.83%, 129.08% and 29.80% higher than LRU, LRU-k, Belady, LRU-txn, Belady-txn and \(\text{OFF}\), respectively. The improvement under PCC is 132.07%, 109.89%, 25.35%, 164.62%, 156.88% and 17.31%, respectively; similarly for ACC.

(5) Impact of configurations. We also tested the impact of system configurations (\(\alpha_{\text{size}}, d_{\text{size}}, \text{and } \#-\text{thds}\)) on throughput.

(a) Varying cache & database size. The throughput of all cache policies increases with larger cache size, e.g., over YCSB under LCC, \(\text{OFF}\), \(\text{OFF}\), Belady and LRU improve by 63.94%, 41.09%, 45.93% and 27.43% when \(\alpha_{\text{size}}\) increases from 20% to 40% (Fig. 3i; see more in [45]). By contrast, all policies are not quite sensitive to database size (\(d_{\text{size}}\)) as shown in Fig. 3i, partially due to that cache hit rate is determined by the cache size, transaction workloads and cache policies only, and the cost of read operations on the database side (HBase) is also insensitive to database size because of the key-value design.

(b) Varying threads (\(\#-\text{thds}\)). Surprisingly, we found that not all cache policies benefit from increased threads. For instance, on Wiki under LCC, with LRU and LRU-k the throughput initially increases with more threads until \(\#-\text{thds}\) reaches 1000, after which their performance even degrades (Fig. 3k; more in [45]). This is because, when compared to \(\text{OFF}\), LRU and LRU-k have higher rate of cache miss or inconsistent cache hit; with larger
We further tested the impact of staleness bound by 10.53 times with 600 threads while the gap increases to 49.63 times with 1400 threads (see [45] for a detailed report). This further justifies the benefit of caching with Off.

**Exp-2: Pipelining** To justify the feasibility of pipelining across transaction batches, we evaluated the overhead of cache scheduling. More specifically, we tested the ratio of the runtime of cache scheduling to the transaction execution time per batch for Off, denoted by OverheadR. The results over TCBench with varying batch sizes are shown in Fig. 4a (see [45] for more). The average OverheadR of Off under ACC, PCC and LCC is 33.09%, 36.76% and 34.90%, respectively, and is consistently below 50% in all cases when batch size varies from 500 to 5000. This validates that, via pipelining cache scheduling does not block transaction execution. We remark that pipelining naturally requires dedicated cores for scheduling, which is typically not a problem for applications in the cloud e.g., EC2.

**Exp-3: Staleness-bounded transaction reordering** We next examined the effectiveness of ReO for Off in more detail.

1. **Overall performance.** We first evaluated (a) the throughput with each reordering method and (b) the maximum staleness that a read observes after reordering. To favor competitors, we restrict that the staleness bound of ReO is 2, while all the competitors have unrestricted staleness. In all tests, Off is set as the default cache policy and each batch has 5000 transactions.

(a) As shown in Fig. 4b, although ReO is subject to bounded staleness, it still gives Off the highest throughput, e.g., on average 39.20%, 34.89%, 36.46% and 42.45% higher than Random, Readfirst, Writefirst, and no reordering, respectively.

(b) While having higher throughput, as shown in Fig. 4c, ReO strictly complies with the specified staleness bound (i.e., 2) and is much smaller than the observed staleness by all competitors.

2. **Impact of staleness bound.** We further tested the impact of the staleness bound \( s \) on the effectiveness of ReO, by varying \( s \) from 0 to 4. The results over YCSB are reported in Fig. 4d (see [45] for Wiki and TCBench, which are similar). Over Wiki, on average ReO improves the throughput of Off (without reordering) by 24.11% with \( s = 0 \), i.e., no staleness is allowed; and this increases to 51.39% with \( s = 4 \). Indeed, with larger \( s \), ReO is given more room to increase cache-side transaction commits via reordering, yielding better throughput. It demonstrates that ReO enables flexible trade-offs between the performance and the “quality” of transaction execution over cache.

**Exp-4: Robustness against transaction batch size.** Finally, we evaluated the impact of transaction batch size on the performance of cache policies. In particular, we want to know whether the throughput is robust against transaction batches of varying sizes. To this end, we evaluated the average throughput over batches with 500 to 5000 transactions, using the same setting as in Exp-1. The results over YCSB are shown in Fig. 3l (see [45] for similar results over Wiki and TCBench) We found that Off is quite robust and stable with transaction batches of varying sizes. For instance, over YCSB, its average throughput is 0.42 M/s over batches of size 5000, while it is 0.41 M/s when the batches are of size 1000; similarly for other datasets.

**Summary.** We find the following on average. (1) Off consistently performs better than other policies in all case. (2) Using Off the transaction throughput of HBase and Memcached is improved by 155.79%, 103.23%, and 115.78% over existing cache policies under LCC, PCC and ACC, respectively. (3) Off has moderate overhead which makes pipelining feasible. (4) The reordering method of Off achieves 38.71% higher transaction throughput than baselines while ensuring staleness bound that others cannot comply with. (5) The performance of Off is robust against transaction batches of varying sizes.

**VIII. Conclusion**

We have made a first attempt to study consistent cache policies for transactional caches. In contrast to conventional caching, consistent caching aims to answer read transactions consistently over caches. We have proved that existing cache policies are not competitive for transactions. Instead, we have proposed batch consistent cache policies for batching-based transaction systems, characterized and settled down their complexity, and developed a consistent cache policy that works with common cache invalidation protocols with provable guarantees. We have also developed reordering optimization to further improve cache performance, with bounded staleness. Our experimental study has shown that the policy is effective in improving transaction throughput of systems extended with caches.

This work aims to initiate the study of consistent caching. We are currently extending the study from batch-based transaction systems to databases that directly use CC without batching.
