Competitive Consistent Caching for Transactions

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

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*

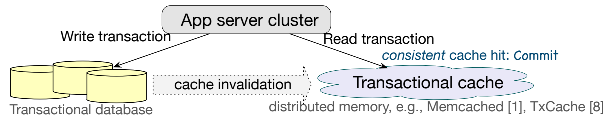


Fig. 1: Cache-augmented data store

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.

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 on to the cache.

Nonetheless, we develop characterizations of optimal cache policies for transactions, based on which we design linear to linearithmic-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.

Transactional 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 transaction 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 $D[0]$, …, $D[k]$, …, 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 $\mathcal{C}$ 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 $\mathcal{C}$. 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 $\mathcal{C}$ may not always be the current version depending on how updates to q_i are propagated to $\mathcal{C}$ via cache invalidation. We say that item q_i is cached if $\mathcal{C}$ caches its value v_i.

Consistent cache hit. A read transaction R is a *consistent* cache hit over cache $\mathcal{C}$ if (a) all items read by R are cached in $\mathcal{C}$ and (b) there exists version k of D such that the cached values to items of R in $\mathcal{C}$ are from snapshot $D[k]$. Intuitively, R is a consistent cache hit over $\mathcal{C}$ if R is a cache hit over $\mathcal{C}$ 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 inconsistent 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 $\mathcal{C}$ 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 $\mathcal{C}$:

(a) R is a consistent cache hit on $\mathcal{C}$;
(b) R is an *inconsistent* cache hit, *i.e.,* all reads of R are cached in $\mathcal{C}$ 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 $\mathcal{C}$.

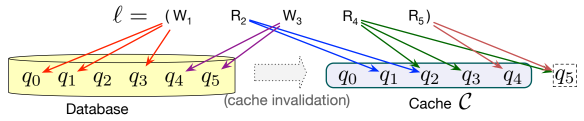
Consistent cache schemes. A consistent cache scheme specifies how $\mathcal{C}$ is maintained and used to process transactions, along with the backend database. We consider three schemes below, depending on how writes are propagated from the database to cache $\mathcal{C}$ and how read transactions are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

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, $\mathcal{C}$ purges all referenced items immediately. When processing a read transaction R over $\mathcal{C}$, the system checks whether all the reads of R are cached in $\mathcal{C}$, if so it answers R. Otherwise, it fetches the missing items from the database, caches them if there is room in $\mathcal{C}$, and answers R. If $\mathcal{C}$ has no room to hold the new items, *i.e.,* a *cache overflow* occurs, $\mathcal{C}$ 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 $\mathcal{C}$ 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 $\mathcal{C}$ to update its outdated items referenced in the message by refetching from the database.

Observe that under both PCC and ACC the cache $\mathcal{C}$ 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 $\mathcal{C}$ 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, $\mathcal{C}$ may


$\ell =$ (W$_1$ R$_2$ W$_3$ R$_4$ R$_5$)
q_0 q_1 q_2 q_3 q_4 q_5
Database
(cache invalidation)
q_0 q_1 q_2 q_3 q_4 q_5
Cache $\mathcal{C}$
Fig. 2: Transactions and cache in Example 1

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 $\mathcal{C}$ under LCC, the cache decides whether R is a consistent cache hit.

(1) If R is a consistent cache hit, it answers R using $\mathcal{C}$ 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 $\mathcal{C}$, and answers R over the updated $\mathcal{C}$ when R becomes a consistent cache hit.

Example 1: Consider a sequence ℓ 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 $\mathcal{C}$ holds $q_0, \ldots, q_4$, $\mathcal{C}$ 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 $\mathcal{C}$; 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 $\mathcal{C}$.

(2) Under ACC. R_2 is a consistent cache hit as q_1 in $\mathcal{C}$ 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 $\mathcal{C}$ 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 $\mathcal{C}$ only without reading D. Instead, R_4 is a cache miss since q_5 is not in $\mathcal{C}$ when R_4 is processed. Hence R_4 reads q_5 from D; however, q_3 is in $D[0]$ while the newly fetched q_5 is from $D[2]$, hence q_3 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 $\mathcal{C}$ by its turn; however, they are not consistent due to W_3. Indeed, q_4 in $\mathcal{C}$ appears in both $D[0]$ and $D[1]$, but not in $D[2]$, while q_5 in $\mathcal{C}$ appears only in $D[2]$. Hence, R_5 is a cache hit but inconsistent, which requires to re-fetch q_4. 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 $\mathcal{C}$ that decide, for each read transaction R in a sequence ℓ of transactions, if R is a cache miss that causes an overflow over $\mathcal{C}$, which cached items in $\mathcal{C}$ 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 $\mathcal{C}$ and $\mathcal{C}$ has no available space to cache the missing reads in R. The sequence of actions decided for each transaction in ℓ forms a *consistent cache schedule* for ℓ over $\mathcal{C}$.

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 $\mathcal{C}$ of size b, and a sequence ℓ of transactions,
OUTPUT	A consistent cache schedule P for ℓ.
OBJ	Minimize cost(P), the total number of reads in the transactions of ℓ 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.,* cost(P). By minimizing 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 ℓ 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.,*PCC, ACC and LCC) employed. In addition, it is also related to how the cache policy observes ℓ 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 ℓ without knowing subsequent transactions in ℓ. This captures conventional cache policies that are designed for *e.g.,* 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 $\mathcal{P}$ for problem TCP. We denote by $\mathsf{cost}(\mathcal{P}, \ell)$ the cost of the cache schedule generated by $\mathcal{P}$ for ℓ. We say that $\mathcal{P}$ is *α-competitive* ($\alpha \geq 1$) if for any sequence ℓ of read and write transactions, we have $\mathsf{cost}(\mathcal{P}, \ell)$ $\leq \alpha \cdot \mathsf{cost}(\mathsf{OPT}, \ell)$, where OPT is the *offline* optimal policy, *i.e.,* $\mathsf{cost}(\mathsf{OPT}, \ell)$ is the cost of the optimal schedule for ℓ generated by OPT that knows the entire ℓ beforehand. A policy is not competitive if it is not α-competitive for any α. Intuitively, if $\mathcal{P}$ is α-competitive for some $\alpha \geq 1$, it means $\mathcal{P}$ 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 $\mathcal{L}_M$ 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 $\mathcal{L}_M$ 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 ℓ with uni-size transactions only, *i.e.,* each read is of size 1 in every transaction of ℓ. Let m be the size of the cache $\mathcal{C}$.

Proposition 1: *Under* PCC*, if ℓ consists of uni-size transactions only, then LRU is m-competitive.* □

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

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

(2) There exists no cache policy in $\mathcal{L}_M$ that is competitive under 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* PCC *and* 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 W_3 also includes q_0 and the cache $\mathcal{C}$ has a size limit of 5. Following $W_1, \ldots, R_5$, assume that there are 4 new transactions $W_6 = \{q_0\}$, $R_7 = \{q_0, q_5\}$, $R_8 = \{q_3, q_4\}$ and $R_9 = \{q_1\}$. Then by the turn of R_4, it incurs a cache overflow under both ACC and LCC since q_5 is not in $\mathcal{C}$ and $\mathcal{C}$ is full at the time.

(1) First consider ACC. By Belady's rule which is to evict from $\mathcal{C}$ the item whose next read time is the most distant in the sequence at the time, we need to replace q_1 in $\mathcal{C}$ with the new item q_5 for R_4. The total number of reads required is 8: q_5 for R_4, q_1 for R_9, and all items in the write transactions except q_5 of W_3. However, if we replace q_0 in $\mathcal{C}$ with q_5 for R_4, we do not need to update q_0 upon W_6, 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 R_4, it replaces q_1 in $\mathcal{C}$ with q_5. Meanwhile, q_3 is re-fetched as q_5 is from $D[2]$ while q_3 is from $D[0]$, causing an inconsistent cache hit; (b) for the same reason, it re-fetches q_4 for R_5 and q_0 for R_7; (c) R_8 is a consistent cache hit; (d) R_9 is a cache miss and it fetches q_1. That is, a total of 5 reads are needed for Belady's rule.

Now consider the cache schedule that replaces q_0 (instead of q_1) in $\mathcal{C}$ with q_5 for R_4. Then for R_4, R_5 and R_7, it acts exactly the same as above. However, both R_8 and R_9 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. □

Theorem 4 shows that caching transactions is much harder than conventional caching. In contrast to TCP, we are not aware of any existing variants of uni-size caching that are NP-hard. We will constructively prove Theorem 4(2a) in §V.

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 $[1, |\ell|]$, where $|\ell|$ is the number of transactions in ℓ.

Example 3: Continue Example 2. Under LCC, both q_0 and q_4 in the cache $\mathcal{C}$ at time 4 (*i.e.,* when answering R_4) will never be used by any transactions after time 4 (*i.e.,* after R_4 in ℓ). Hence, an optimal cache schedule for ℓ would evict or re-fetch q_0 and q_4 after time 4 upon cache overflows.

Similarly, under PCC and ACC, q_0 at time 4 in $\mathcal{C}$ is not helpful for any transactions after R_4 since q_0 due to W_6. □

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 $\mathcal{C}$ be the cache buffer at the current time t. Denote by ℓ_t 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 $\mathcal{C}$ at time t; and

(b) for *any* consistent cache schedule for ℓ_t over $\mathcal{C}$, the cached q will not contribute to the committing of transactions in ℓ_t, *i.e.,*q will have been re-fetched from the database to answer any of the transactions in ℓ_t that contain $r(q)$.

That is, when $r(q)$ is obsolete, by the time the immediate next read transaction R in ℓ_t 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 ℓ_t over $\mathcal{C}$. 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 ℓ if for any other schedule P′, $\text{cost}(\mathsf{P},\ell) \leq \text{cost}(\mathsf{P}',\ell)$. Denote by $\mathsf{P}[i]$ the cache replacement decision made by P for read transaction R of ℓ that arrives at time i.

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

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 ℓ 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 $\mathcal{C}$, sequence ℓ of read and write transactions, read $r(q)$ that is cached in $\mathcal{C}$ at time t.

Proposition 6: *Under* PCC *and* ACC, $r(q)$ *is obsolete for* ℓ *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* $w(q)$. □

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

(2) LCC. Due to the possibility that read transactions can still be well served over cache $\mathcal{C}$ even $\mathcal{C}$ 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* ℓ *consists of uni-size transactions.* □

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 $\mathbb{OFF}$ policy. The policy, denoted by $\mathbb{OFF}$ (Obsolete 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.

$\mathbb{OFF}$ builds upon the techniques of conventional caching with varying sizes [52]. It first classifies all reads in the transactions of ℓ by their sizes. For each transaction R, if R is an inconsistent cache hit or cache miss, $\mathbb{OFF}$ refreshes staled reads in the cache *w.r.t.* write transactions prior to R. If R is a cache miss, $\mathbb{OFF}$ fetches the missing new reads and then answers R using cache consistently; if a cache overflow occurs even all obsolete reads have been evicted, $\mathbb{OFF}$ 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.

ALGORITHM 1: The $\mathbb{OFF}$ policy

```
Input: Cache $\mathcal{C}$ and transaction sequence $\ell$.
Upon processing a read transaction $R_t$ in $\ell$ (at time $t$):
1  if isCCH$(R_t, \mathcal{C})$ = true then          // $R_t$ is a consistent cache hit in $\mathcal{C}$
2      answer $R_t$ using $\mathcal{C}$ directly;
3  else                          // $R_t$ is a cache miss or inconsistent cache hit
4      $S \leftarrow$ findOB$(\mathcal{C}, \ell, t)$;          // find reads in $\mathcal{C}$ obsolete at time $t$
5      $\mathcal{C} \leftarrow \mathcal{C} \setminus S$ ;          // evict obsolete reads currently in $\mathcal{C}$
6      foreach $q \in R_t$ that is either not in $\mathcal{C}$ or outdated in $\mathcal{C}$ do
7          if $q$ is in $\mathcal{C}$ but outdated then update $q$ in $\mathcal{C}$; continue;
8          if $\mathcal{C}$ has no room for $q$ then
9              repeat $\lceil \log r \rceil + 1$ times  // $r = \max_{i \in [1, \lfloor \log k \rfloor + 1]} r_i$,
                   where $r_i$ is the ratio of the maximum $i$-read size over the
                   minimum $i$-read size
10                 foreach $i \in [1, \lfloor \log k \rfloor + 1]$ do
11                     evict the most distant $i$-read in $\mathcal{C}$
12         fetch and cache $q$ in $\mathcal{C}$;
```

We next present $\mathbb{OFF}$ in details. Denote by Σ the set of all distinct items read or written by transactions in ℓ. $\mathbb{OFF}$ 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 $[\min_{q \in \Sigma} |q| \cdot 2^{i-1}, \min_{q \in \Sigma} |q| \cdot 2^{i})$, where $k = \frac{\max_{q \in \Sigma} |q|}{\min_{q \in \Sigma} |q|}$ in which $|q|$ 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 $[\min_{q \in \Sigma} |q| \cdot 2^{i-1}, \min_{q \in \Sigma} |q| \cdot 2^{i})$.

It then processes read and write transactions of ℓ one by one as shown in Algorithm 1. When processing transaction R_t at time t, it first checks whether it is a consistent cache hit over cache $\mathcal{C}$ (via isCCH; see [45] for more); if so, it answers (commits) R_t (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 $\mathcal{C}$ (line 5). It then processes reads in R_t one by one: if read q in R_t is in $\mathcal{C}$ but outdated due to write transactions before t, it updates q in $\mathcal{C}$ with the latest version from the database (line 7); otherwise if q is not yet cached, it checks whether there is enough room in $\mathcal{C}$ 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 ℓ 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 $\mathbb{OFF}$ will never evict a good read by mistakenly recognizing it as obsolete.

For any given time t, findOB identifies reads in $\mathcal{C}$ that are obsolete for ℓ at t. The key idea is to execute ℓ_t, the suffix of ℓ starting from time t, via a modified dry run of $\mathbb{OFF}$ over a temporary cache buffer $\mathcal{C}'$ of infinite size, during which findOB marks reads in $\mathcal{C}$ that can be determined obsolete for certain.

More specifically, it first initializes $\mathcal{C}'$ the same as $\mathcal{C}$ (line 1).

ALGORITHM 2: The findOB Procedure

```
Input: Sequence $\ell$ of transactions, cache $\mathcal{C}$ at current time $t$.
1  $\mathcal{C}' \leftarrow \mathcal{C}$; $\ell_t \leftarrow \ell[t, +\infty]$;        // create a new buffer $\mathcal{C}'$ without size limit
2  while $\ell_t \neq nil$ and $\mathcal{C}$ has unmarked reads do
3      $R_i \leftarrow \ell_t$.pop();                    // $R_i$: the transaction at time $i \geq t$
4      if isCCH ($R_i, \mathcal{C}'$) = true then
5          foreach $q \in R_i$ do
6              if $q$ is in $\mathcal{C}$ and unmarked then mark $q$ as safe in $\mathcal{C}$;
7      else
8          if $R_i$ is a cache miss in $\mathcal{C}'$ then
9              foreach $q \in R_i$ and $q \notin \mathcal{C}$ do add $q$ to $\mathcal{C}'$ ;
10         if isCCH($R_i, \mathcal{C}'$) = true then
11             foreach $q \in R_i$ do
12                 if $q$ is unmarked in $\mathcal{C}$ then mark $q$ as safe in $\mathcal{C}$;
13             continue
           // life$_t(q)$: duration at which $q$ cached at $t$ is also in the database.
14         $\mathcal{H}_l \leftarrow$ a max-heap of all $q \in R_i$ by lower endpoints of life$_t(q)$;
15         $\mathcal{H}_u \leftarrow$ a min-heap of all $q \in R_i$ by upper endpoints of life$_t(q)$;
16         $q_1 \leftarrow \mathcal{H}_l$.pop(); $q_2 \leftarrow \mathcal{H}_u$.pop();
           /* let life$_t(q_1) = [lb_1, ub_1]$, life$_t(q_2) = [lb_2, ub_2]$ */
17         while $lb_1 > ub_2$ do
18             replace life$_t(q_2)$ with life$_i(q_2)$ and update $\mathcal{H}_l$ and $\mathcal{H}_u$
                 accordingly;
19             if $q_2 \in \mathcal{C}$ and unmarked then mark $q_2$ as obsolete in $\mathcal{C}$;
20             $q_1 \leftarrow \mathcal{H}_l$.pop(); $q_2 \leftarrow \mathcal{H}_u$.pop();
21         foreach $q$ in $R_i$ that is also in $\mathcal{C}$ do mark $q$ as safe if not marked;
22 return all reads in $\mathcal{C}$ that are marked as obsolete;
```

It then examines transactions in ℓ_t one by one against $\mathcal{C}'$, where ℓ_t is the suffix of ℓ starting from time t. It marks reads in the transactions as either *safe* or *obsolete* if they are also in $\mathcal{C}$. The process terminates when all reads cached in $\mathcal{C}$ are marked (or all reads in ℓ_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 ℓ_t, say R_i at time i (line 3). It checks whether R_i is a consistent cache hit over $\mathcal{C}'$ (line 4). If so, all reads in R_i that are also in $\mathcal{C}$ 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 $\mathcal{C}'$. If R_i is a cache miss over $\mathcal{C}'$, findOB expands $\mathcal{C}'$ by including reads of R_i that are missing in $\mathcal{C}'$, so that R_i becomes a consistent or inconsistent cache hit. If R_i is now a consistent cache hit over $\mathcal{C}'$, findOB marks all reads of R_i that are also in $\mathcal{C}$ but are not yet marked as *safe* and moves on to the next transaction in ℓ_t (lines 10-13).

If R_i remains an inconsistent cache hit over $\mathcal{C}'$, 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 $\mathcal{C}$ if they are also contained in $\mathcal{C}$ but not marked (line 19). findOB does this by maintaining two heaps of reads in R_i: one is a max-heap $\mathcal{H}_l$ that sorts reads by the lower end points of their life$_t$ (life span in the database snapshots; see Algorithm 2) ranges in descending order and the other is a min-heap $\mathcal{H}_u$ that sorts reads by the upper endpoints of their life$_t$ ranges in increasing order (lines 14-15). Each iteration, findOB pops the top read from $\mathcal{H}_l$ and $\mathcal{H}_u$, 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 $\mathcal{C}$ and is not yet marked (line 19). The iteration terminates if the pair of the head reads in $\mathcal{H}_l$ and $\mathcal{H}_u$ become consistent (line 20). findOB marks all those reads in R_i that also in $\mathcal{C}$ but are not yet marked as *safe* (line 21).

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

Complexity. $\mathbb{OFF}$ generates a consistent cache schedule within $O(K_{\mathcal{C}} * \|\ell\| + |\ell| * T_{\text{findOB}})$-time under all three schemes, where (a) $K_{\mathcal{C}}$ is the number of classes that $\mathcal{C}$ is divided into (*i.e.,* $K_{\mathcal{C}} = \lfloor \log k \rfloor + 1$; recall that $\mathbb{OFF}$ groups reads in $\mathcal{C}$ into classes), (b) $|\ell|$ (resp. $\|\ell\|$) is the total number of transactions (resp. reads) in ℓ, and (c) T_{findOB} is the complexity of findOB($\mathcal{C}, \ell, t$). Under PCC and ACC, T_{findOB} is in $O(K_{\mathcal{C}})$-time; under LCC, T_{findOB} is in $O(h * \log c)$-time, where (i) h is the number of transactions in $\ell_t^{\mathcal{C}}$, which is the shortest sub-sequence of ℓ that starts from time t and covers reads in $\mathcal{C}$, 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 $\ell_t^{\mathcal{C}}$ (*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 $\mathbb{OFF}$. Recall the notion of competitiveness in §III. In particular, a policy $\mathcal{P}$ is *optimal* if it is 1-competitive, *i.e.,* it always generates consistent cache schedule of the lowest cost for each and every sequence ℓ.

Theorem 9: *Under both* PCC *and* ACC*,* $\mathbb{OFF}$

(1) is $2 \log k$-competitive; and

(2) is optimal when the reads are of unit size. □

Theorem 9 is constructive proof of Proposition 3. It shows that, in contrast to conventional policies that are not competitive for transactions, $\mathbb{OFF}$ is competitive and even optimal. As will be shown in §VII, by evicting obsolete reads $\mathbb{OFF}$ 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 $\mathcal{C}$ contains $\{q_0, q_1, q_2, q_3, q_4\}$. Then for the transaction sequence ℓ in Example 2, $\mathbb{OFF}$ generates an optimal schedule that incurs 5 reads under PCC. Consider a reordering $\ell' = (R_2, R_8, R_9, R_4, R_5,$

R_7, W_1, W_3, W_6) of ℓ. One can verify that (a) an optimal schedule for ℓ' under PCC incurs only 1 read from database D, *i.e.,* fetch q_5 for R_4; (b) ℓ' is the best reordering one can find for ℓ; however, (c) in ℓ R_8 reads $D[2]$ while it reads $D[0]$ with ℓ'. □

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 $\mathcal{C}$ 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 $\mathcal{C}$ 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 $\mathsf{cost}(\mathsf{P})$, where P is the optimal cache schedule for ℓ' (recall $\mathsf{cost}()$ in §II-B).

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 $|\ell|$. 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 $\mathsf{RVer}(R.r[q], \ell)$ the number of writes $w[q]$ in transactions prior to R in ℓ. The *staleness* of read $r[q]$ of R in the reordering ℓ' of ℓ, denoted by $\mathsf{stale}(R.r[q], \ell')$, is defined as $|\mathsf{RVer}(R.r[q], \ell') - \mathsf{RVer}(R.r[q], \ell)|$. Intuitively, $\mathsf{RVer}(R.r[q], \ell)$ measures the version of q-values $R.r[q]$ would see if transactions were committed according to the "natural" order of ℓ, and $\mathsf{stale}(R.r[q], \ell')$ 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 $\mathsf{stale}(R.r, \ell') \leq s$.

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 $G_\ell(V_1, V_2, E)$, where V_1 and V_2 are the two vertex sets and $E \subseteq V_1 \times V_2$. Each vertex v in V_1 encodes a write transaction W in ℓ and each $v' \in V_2$ encodes a read transaction R. It orders V_1 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 (V_1) 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 bipartite matching $M \subseteq E$ of G_ℓ by iteratively processing vertices of V_1: each iteration it picks $v \in V_1$ with the maximum degree and assigns all vertices $v' \in V_2$ connecting to it; once v' is assigned to v, it also removes all edges from E that connects v' and other vertices of V_1.

(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 $|\ell|/k$ 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 $O(|\ell| \log |\ell|)$-time, where $|\ell|$ is the total transaction size of ℓ (see [45] for details).

VII. Implementation and Experimental Study

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

Prototype. We have developed TCache, a prototype that implements $\mathbb{OFF}$ 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 B_i of transactions, TCache generates a cache schedule for it using $\mathbb{OFF}$ (§V) and the reordering optimization (§VI). It instructs Memcached to use the generated schedule instead of the default LRU to serve read transactions in B_i. TCache also pipelines cache scheduling and transaction execution: when the underlying database is executing transaction batch B_i, it collects and generates cache schedule for the subsequent batch B_{i+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 $\mathbb{OFF}$ 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.

(a) θ: 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 θ means more skewed access distribution.

(b) `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.

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

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

(c) 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 ℓ 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 $\mathbb{OFF}$ with existing methods.

(a) Cache policies. We compared $\mathbb{OFF}$ 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. We compared $\mathbb{OFF}$ (with ReO) with the following methods:

- LRU: the default cache policy of Memcached [1].
- LRU-k: a modern variant of LRU based on LRU-k [44].
- Belady: the optimal offline policy for uni-size reads [55].
- LRU-txn: a variant of LRU that evicts the least recently used cached transaction (instead of item) upon cache overflows.
- Belady-txn: a variant of Belady that evicts transactions at a time upon cache overflows, similar to LRU-txn.
- $\mathbb{OFF}^-$: a variant of $\mathbb{OFF}$ that does not evict obsolete items first upon overflows, following the paging policy in [52].
- $\mathbb{OFF}^0$: a plain version of $\mathbb{OFF}$ that does not employ ReO.

(b) Reordering policies. We also compared the reordering optimization of $\mathbb{OFF}$ (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 $\mathbb{OFF}$ (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 $\mathbb{OFF}$ 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 α_{csize} fraction of all the read/write items in the transaction workloads, where α_{csize} varies from 20% to 40% (40% by default). To measure the impact of parallelism, we also varied the total number of transaction threads (`#-thds`) 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 polices, *e.g.,* over YCSB, $\mathbb{OFF}$, $\mathbb{OFF}^-$, 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 (θ and write%). Key results are reported in Figures 3b-3f; see [45] for more). We found that, with $\mathbb{OFF}$ the overall throughput is consistently the best among all. Over YCSB under LCC, the throughput with

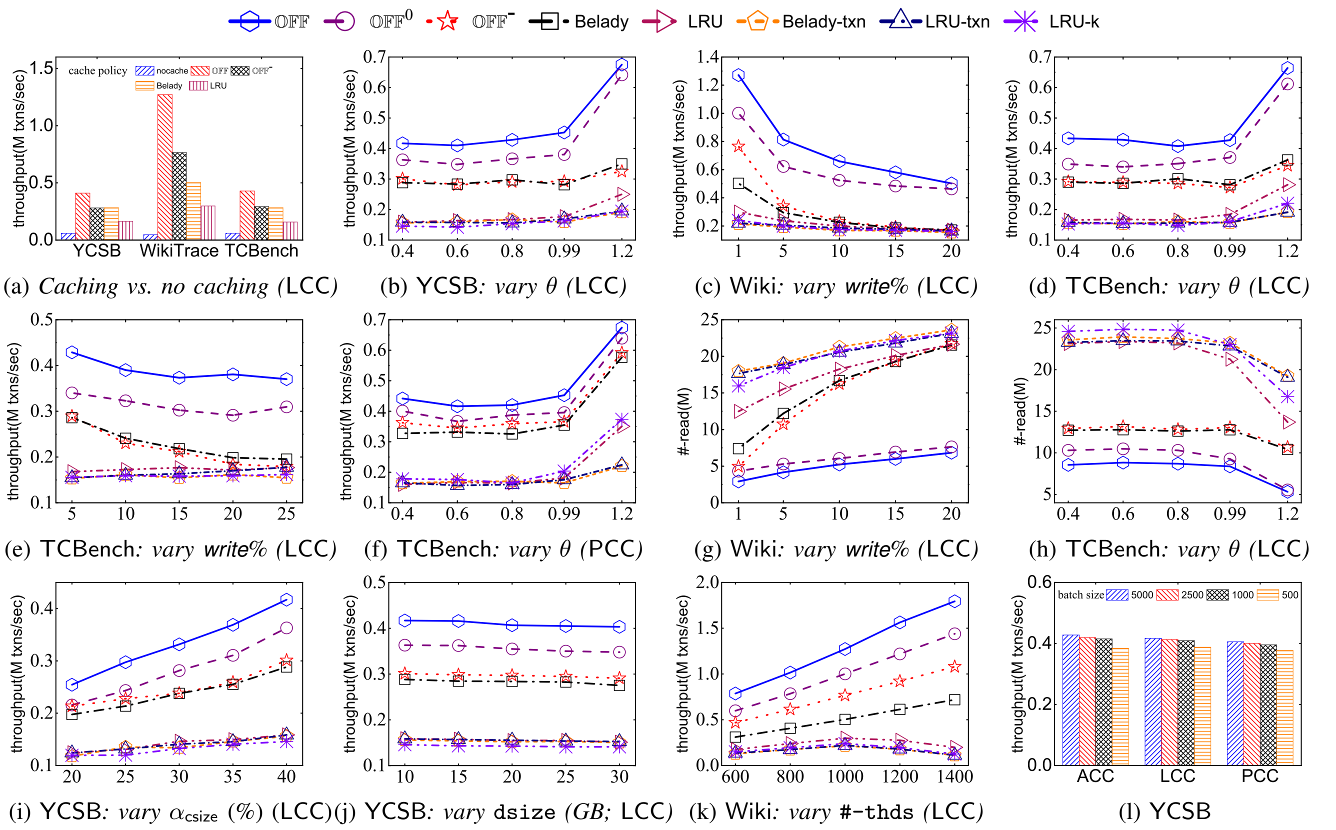


Fig. 3: Experimental results for Exp-1 and Exp-4

𝕆𝔽𝔽 is on average 159.26%, 195.71%, 57.51%, 182.20%, 183.71% and 58.92% higher than with LRU, LRU-k, Belady, LRU-txn, Belady-txn and $\mathbb{OFF}^-$, respectively. The improvement under PCC is 132.07%, 109.89%, 25.35%, 164.62%, 156.88% and 17.31%, respectively; similarly for ACC.

(2) Read load. To understand why 𝕆𝔽𝔽 allows higher transaction throughput, we tested the database load with different cache policies measured as #-read, the number of read operations (per 1K transactions) that are carried out at HBase. We found that with 𝕆𝔽𝔽 higher percentage of read load is shifted to Memcached nodes due to more cache-side transaction commits. For instance, under LCC, over Wiki, on average 𝕆𝔽𝔽 reduces 66.47% and 71.97% of the #-read of Belady and LRU, respectively; similarly over TCBench (Figures 3g-3h; more in [45]). This is because #-read is heavily related to cache-side transaction aborts due to inconsistencies, which in turn depend on how well obsolete queries are dealt with by the cache policies. With findOB, 𝕆𝔽𝔽 eliminates most of obsolete queries while others cannot.

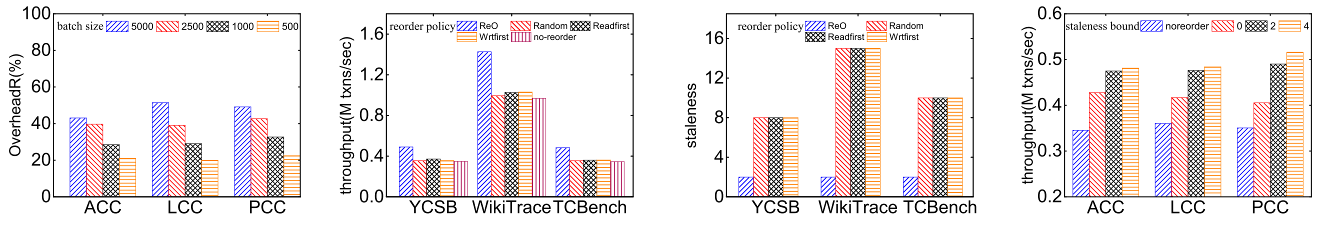
(3) Obsolete reads. Obsolete items have an evident impact on the performance of TCache and findOB of 𝕆𝔽𝔽 and $\mathbb{OFF}^0$ are effective in identifying them. This is reflected by the larger improvement of 𝕆𝔽𝔽 and $\mathbb{OFF}^0$ over other cache policies under LCC than under PCC and ACC (see Figures 3d and 3f). Indeed, under LCC, transactions can make higher use of cached items by allowing consistent cache hit over possibly stale items. This can lead to higher throughput as long as the cache can identify and evict obsolete items as many and early as possible, for which 𝕆𝔽𝔽 does much better than the other cache policies.

(4) ReO optimization. We found that $\mathbb{OFF}^0$ (𝕆𝔽𝔽 without ReO) also consistently outperforms all other baselines, *e.g.*, over TCBench 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 $\mathbb{OFF}^-$, respectively. On average, ReO contributes to nearly half of the speedup that 𝕆𝔽𝔽 has 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 𝕆𝔽𝔽.

(5) Impact of configurations. We also tested the impact of system configurations (α_{csize}, `dsize`, and `#-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, 𝕆𝔽𝔽, $\mathbb{OFF}^-$, Belady and LRU improve by 63.94%, 41.09%, 45.93% and 27.43% when α_{csize} increases from 20% to 40% (Fig. 3i; see more in [45]). By contrast, all policies are not quite sensitive to database size (`dsize`) as shown in Fig. 3j, 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 (`#-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 `#-thds` reaches 1000, after which their performance even degrades (Fig. 3k; more in [45]). This is because, when compared to 𝕆𝔽𝔽, LRU and LRU-k have higher rate of cache miss or inconsistent cache hit; with larger



(a) TCBench (b) *Throughput* (PCC) (c) *Staleness* (PCC) (d) YCSB

Fig. 4: Experimental results for Exp-2 and Exp-3

`#-thds` the increasing amount of reads executed at database causes higher contention that outweighs the increased cache hit. By contrast, $\mathbb{OFF}$ benefits most from increased threads consistently, with highest throughput in all cases.

Additionally, we also found that the gap between $\mathbb{OFF}$ and nocache (without caching) increases noticeably with added threads. For instance, under LCC over Wiki, $\mathbb{OFF}$ improves nocache 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 $\mathbb{OFF}$.

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 $\mathbb{OFF}$, denoted by OverheadR. The results over TCBench with varying batch sizes are shown in Fig. 4a (see [45] for more). The average OverheadR of $\mathbb{OFF}$ 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 $\mathbb{OFF}$ 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 favour competitors, we restrict that the staleness bound of ReO is 2, while all the competitors have unrestricted staleness. In all tests, $\mathbb{OFF}$ 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 $\mathbb{OFF}$ 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 $\mathbb{OFF}$ (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 $\mathbb{OFF}$ 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) $\mathbb{OFF}$ consistently performs better than other policies in all case. (2) Using $\mathbb{OFF}$ 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) $\mathbb{OFF}$ has moderate overhead which makes pipelining feasible. (4) The reordering method of $\mathbb{OFF}$ achieves 38.71% higher transaction throughput than baselines while ensuring staleness bound that others cannot comply with. (5) The performance of $\mathbb{OFF}$ 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.

References

[1] "Memcached," https://memcached.org/, 2021.

[2] "Ncache," https://www.alachisoft.com/ncache/, 2021.

[3] "Redis," https://redis.io/, 2021.

[4] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: facebook's distributed data store for the social graph," in *ATC*, A. Birrell and E. G. Sirer, Eds., 2013, pp. 49–60.

[5] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 385–398.

[6] L. Phillips and B. Fitzpatrick, "Livejournal's backend and memcached: Past, present, and future," in *LISA*, L. Damon, Ed. USENIX, 2004.

[7] "Mediawiki," https://www.mediawiki.org/, 2021.

[8] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov, "Transactional consistency and automatic management in an application data cache," in *OSDI*, R. H. Arpaci-Dusseau and B. Chen, Eds. USENIX Association, 2010, pp. 279–292.

[9] I. Eyal, K. Birman, and R. van Renesse, "Cache serializability: Reducing inconsistency in edge transactions," in *ICDCS*, 2015, pp. 686–695.

[10] "Mysql cluster cge," https://www.mysql.com/products/cluster/, 2016.

[11] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd, "The SNOW theorem and latency-optimal read-only transactions," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, K. Keeton and T. Roscoe, Eds., 2016, pp. 135–150.

[12] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen, "Pisces: A scalable and efficient persistent transactional memory," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, Jul. 2019, pp. 913–928.

[13] S. Almeida, J. a. Leitão, and L. Rodrigues, "Chainreaction: A causal+ consistent datastore based on chain replication," in *EuroSys*, 2013, p. 85–98.

[14] A. Z. Tomsic, M. Bravo, and M. Shapiro, "Distributed transactional reads: the strong, the quick, the fresh & the impossible," in *Middleware*, P. Ferreira and L. Shrira, Eds., 2018, pp. 120–133.

[15] P. Bailis, A. D. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica, "Scalable atomic visibility with RAMP transactions," in *SIGMOD*, 2014.

[16] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford, "Spanner: Becoming a SQL system," in *SIGMOD*, 2017, pp. 331–343.

[17] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *NSDI*, N. Feamster and J. C. Mogul, Eds., 2013, pp. 313–328.

[18] K. Spirovska, D. Didona, and W. Zwaenepoel, "Wren: Nonblocking reads in a partitioned transactional causally consistent data store," in *DSN*, 2018, pp. 1–12.

[19] H. Lu, S. Sen, and W. Lloyd, "Performance-optimal read-only transactions," in *OSDI*, 2020, pp. 333–349.

[20] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," *ACM Trans. Comput. Syst.*, vol. 35, no. 4, pp. 12:1–12:37, 2018.

[21] H. Fan, W. Golab, and C. B. M. III, "ALOHA-KV: high performance read-only and write-only distributed transactions," in *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, 2017, pp. 561–572.

[22] M. J. Kishi, S. Peluso, H. F. Korth, and R. Palmieri, "SSS: scalable key-value store with external consistent and abort-free read-only transactions," in *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*, 2019, pp. 589–600.

[23] X. Yu, Y. Xia, A. Pavlo, D. Sánchez, L. Rudolph, and S. Devadas, "Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system," *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1289–1302, 2018.

[24] S. Ghandeharizadeh, J. Yap, and H. Nguyen, "Strong consistency in cache augmented SQL systems," in *Middleware*, L. Réveillère, L. Cherkasova, and F. Taïani, Eds. ACM, 2014, pp. 181–192.

[25] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Characterizing facebook's memcached workload," *IEEE Internet Comput.*, vol. 18, no. 2, pp. 41–49, 2014.

[26] "Foshttpcache: An introduction to cache invalidation," https://foshttpcache.readthedocs.io/en/stable/invalidation-introduction.html, 2021.

[27] A. Fiat and G. J. Woeginger, *Online algorithms: The state of the art*. Springer, 1998, vol. 1442.

[28] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A workload-driven approach to database replication and partitioning," *PVLDB*, vol. 3, no. 1–2, p. 48–57, 2010.

[29] G. Prasaad, A. Cheung, and D. Suciu, "Handling highly contended OLTP workloads using fast dynamic partitioning," in *SIGMOD*, 2020, pp. 527–542.

[30] A. Pavlo, C. Curino, and S. B. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," in *SIGMOD*, 2012, pp. 61–72.

[31] D. J. Abadi and J. M. Faleiro, "An overview of deterministic database systems," *Commun. ACM*, vol. 61, no. 9, pp. 78–88, 2018.

[32] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi, "Fast distributed transactions and strongly consistent replication for OLTP database systems," *ACM Trans. Database Syst.*, vol. 39, no. 2, pp. 11:1–11:39, 2014.

[33] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era (it's time for a complete rewrite)," in *VLDB*. ACM, 2007, pp. 1150–1160.

[34] K. Ren, A. Thomson, and D. J. Abadi, "VLL: a lock manager redesign for main memory database systems," *VLDB J.*, vol. 24, no. 5, pp. 681–705, 2015.

[35] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *SIGMOD*, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds. ACM, 2012, pp. 1–12.

[36] K. Ren, J. M. Faleiro, and D. J. Abadi, "Design principles for scaling multi-core OLTP under high contention," in *SIGMOD*, 2016, pp. 1583–1598.

[37] E. Zamanian, J. Shun, C. Binnig, and T. Kraska, "Chiller: Contention-centric transaction execution and data partitioning for modern networks," in *SIGMOD*, 2020, p. 511–526.

[38] A. Quamar, K. A. Kumar, and A. Deshpande, "Sword: Scalable workload-aware data placement for transactional workloads," in *EDBT*, 2013, p. 430–441.

[39] Y. Lu, X. Yu, L. Cao, and S. Madden, "Aria: A fast and practical deterministic OLTP database," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2047–2060, 2020.

[40] D. V. Pattishall, "Friendster: Scaling for 1 billion queries per day," 2005.

[41] ——, "Federation at flickr (doing billions of queries per day)," 2007.

[42] R. Shoup and D. Pritchett, "The ebay architecture," 2006.

[43] "Voltdb," https://www.voltdb.com/, 2021.

[44] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *SIGMOD*, P. Buneman and S. Jajodia, Eds. ACM Press, 1993, pp. 297–306.

[45] "Full version," http://homepages.inf.ed.ac.uk/ycao/CCfull.pdf, 2021.

[46] E. P. C. Jones, D. J. Abadi, and S. Madden, "Low overhead concurrency control for partitioned main memory databases," in *SIGMOD*, A. K. Elmagarmid and D. Agrawal, Eds. ACM, 2010, pp. 603–614.

[47] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, "Competitive paging algorithms," *J. Algorithms*, vol. 12, no. 4, pp. 685–699, 1991.

[48] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 1, pp. 1–294, 2020.

[49] M. A. Maddah-Ali and U. Niesen, "Fundamental limits of caching," *IEEE Trans. Inf. Theory*, vol. 60, no. 5, pp. 2856–2867, 2014.

[50] M. Chrobak and J. Noga, "Lru is better than fifo," *Algorithmica*, vol. 23, no. 2, pp. 180–185, 1999.

[51] V. Touzeau, C. Maïza, D. Monniaux, and J. Reineke, "Fast and exact analysis for LRU caches," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 54:1–54:29, 2019.

[52] S. Irani, "Page replacement with multi-size pages and applications to web caching," *Algorithmica*, vol. 33, no. 3, pp. 384–409, 2002.

[53] M. Chrobak, G. J. Woeginger, K. Makino, and H. Xu, "Caching is hard - even in the fault model," *Algorithmica*, vol. 63, no. 4, pp. 781–794, 2012.

[54] L. Folwarczný and J. Sgall, "General caching is hard: Even with small pages," *Algorithmica*, vol. 79, no. 2, pp. 319–339, 2017.

[55] L. A. Belady, "A study of replacement algorithms for virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[56] M. Chrobak, H. J. Karloff, T. H. Payne, and S. Vishwanathan, "New results on server problems," in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, D. S. Johnson, Ed., 1990, pp. 291–300.

[57] S. Albers, S. Arora, and S. Khanna, "Page replacement for general caching problems," in *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland, USA*, R. E. Tarjan and T. J. Warnow, Eds., 1999, pp. 31–40.

[58] S. Albers, "New results on web caching with request reordering," *Algorithmica*, vol. 58, no. 2, pp. 461–477, 2010.

[59] S. Ding, J. Attenberg, R. Baeza-Yates, and T. Suel, "Batch query processing for web search engines," in *WSDM*, I. King, W. Nejdl, and H. Li, Eds. ACM, 2011, pp. 137–146.

[60] T. Feder, R. Motwani, R. Panigrahy, and A. Zhu, "Web caching with request reordering," in *SODA*, D. Eppstein, Ed. ACM/SIAM, 2002, pp. 104–105.

[61] T. Feder, R. Motwani, R. Panigrahy, S. S. Seiden, R. van Stee, and A. Zhu, "Combining request scheduling with web caching," *Theor. Comput. Sci.*, vol. 324, no. 2-3, pp. 201–218, 2004.

[62] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020.* USENIX Association, 2020, pp. 191–208.

[63] "Varnish," https://varnish-cache.org/, 2021.

[64] S. Albers, "Online algorithms: a survey," *Mathematical Programming*, vol. 97, no. 1, pp. 3–26, 2003.

[65] A. Moga, I. Botan, and N. Tatbul, "Upstream: storage-centric load management for streaming applications with update semantics," *VLDB J.*, vol. 20, no. 6, pp. 867–892, 2011.

[66] R. Alonso, D. Barbará, and H. Garcia-Molina, "Data caching issues in an information retrieval system," *ACM Trans. Database Syst.*, vol. 15, no. 3, pp. 359–384, 1990.

[67] L. Golab, T. Johnson, and V. Shkapenyuk, "Scheduling updates in a real-time stream warehouse," in *ICDE*. IEEE Computer Society, 2009, pp. 1207–1210.

[68] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, Eds. ACM, 2010, pp. 143–154.

[69] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64.

[70] R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 553–564, 2017.

[71] Z. Song, D. S. Berger, K. Li, and W. Lloyd, "Learning relaxed belady for content distribution network caching," in *NSDI*, R. Bhagwan and G. Porter, Eds., 2020, pp. 529–544.

[72] "Amazon ec2," https://aws.amazon.com/ec2/, 2021.