# Anti-Caching: A New Approach to Database Management System Architecture

Justin DeBrabant
Brown University
debrabant@cs.brown.edu

Andrew Pavlo
Brown University
pavlo@cs.brown.edu

Stephen Tu
MIT CSAIL
stephentu@csail.mit.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

## ABSTRACT

The traditional wisdom for building disk-based relational database management systems (DBMS) is to organize data in heavily-encoded blocks stored on disk, with a main memory block cache. In order to improve performance given high disk latency, these systems use a multi-threaded architecture with dynamic record-level locking that allows multiple transactions to access the database at the same time. Previous research has shown that this results in substantial overhead for on-line transaction processing (OLTP) applications [15].

The next generation DBMSs seek to overcome these limitations with architecture based on main memory resident data. To overcome the restriction that all data fit in main memory, we propose a new technique, called *anti-caching*, where cold data is moved to disk in a transactionally-safe manner as the database grows in size. Because data initially resides in memory, an anti-caching architecture reverses the traditional storage hierarchy of disk-based systems. Main memory is now the primary storage device.

We implemented a prototype of our anti-caching proposal in a high-performance, main memory OLTP DBMS and performed a series of experiments across a range of database sizes, workload skews, and read/write mixes. We compared its performance with an open-source, disk-based DBMS optionally fronted by a distributed main memory cache. Our results show that for higher skewed workloads the anti-caching architecture has a performance advantage over either of the other architectures tested of up to 9× for a data size 8× larger than memory.

## 1. INTRODUCTION

Historically, the internal architecture of DBMSs has been predicated on the storage and management of data in heavily-encoded disk blocks. In most systems, there is a header at the beginning of each disk block to facilitate certain operations in the system. For example, this header usually contains a "line table" at the front of the block to support indirection to tuples. This allows the DBMS to reorganize blocks without needing to change index pointers. When a disk block is read into main memory, it must then be translated into main memory format.

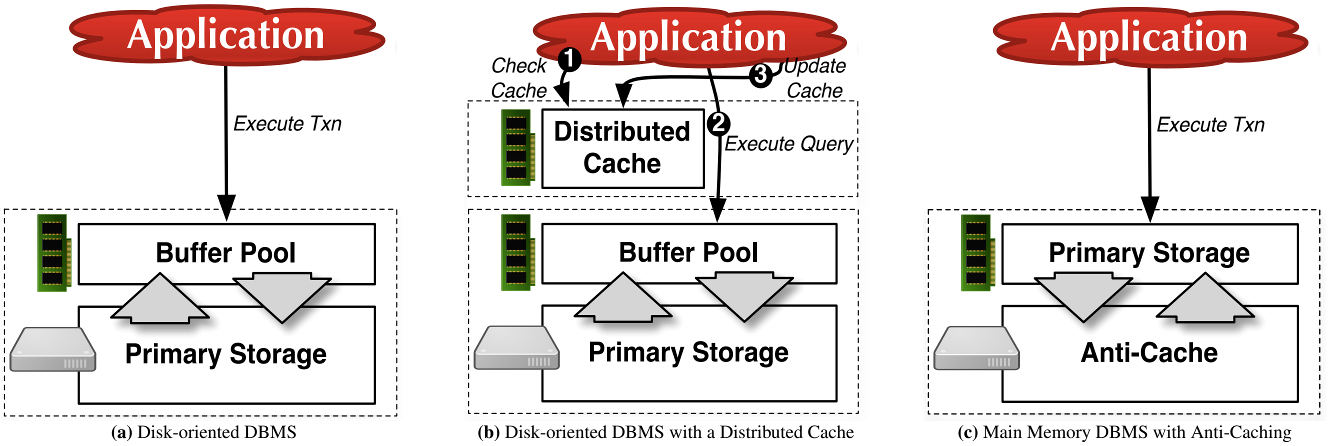
DBMSs invariably maintain a buffer pool of blocks in main memory for faster access. When an executing query attempts to read a disk block, the DBMS first checks to see whether the block already exists in this buffer pool. If not, a block is evicted to make room for the needed one. There is substantial overhead to managing the buffer pool, since blocks have to be pinned in main memory and the system must maintain an eviction order policy (e.g., least recently used). As noted in [15], when all data fits in main memory, the cost of maintaining a buffer pool is nearly one-third of all the CPU cycles used by the DBMS.

The expense of managing disk-resident data has fostered a class of new DBMSs that put the entire database in main memory and thus have no buffer pool [11]. TimesTen was an early proponent of this approach [31], and more recent examples include H-Store [2, 18], MemSQL [3], and RAMCloud [25]. H-Store (and its commercial version VoltDB [4]) performs significantly better than disk-based DBMSs on standard OLTP benchmarks [29] because of this main memory orientation, as well as from avoiding the overhead of concurrency control and heavy-weight data logging [22].

The fundamental problem with main memory DBMSs, however, is that this improved performance is only achievable when the database is smaller than the amount of physical memory available in the system. If the database does not fit in memory, then the operating system will start to page virtual memory, and main memory accesses will cause page faults. Because page faults are transparent to the user, in this case the main memory DBMS, the execution of transactions is stalled while the page is fetched from disk. This is a significant problem in a DBMS, like H-Store, that executes transactions serially without the use of heavyweight locking and latching. Because of this, all main memory DBMSs warn users not to exceed the amount of real memory [5]. If memory is exceeded (or if it might be at some point in the future), then a user must either (1) provision new hardware and migrate their database to a larger cluster, or (2) fall back to a traditional disk-based system, with its inherent performance problems.

One widely adopted performance enhancer is to use a main memory distributed cache, such as Memcached [14], in front of a disk-based DBMS. Under this two-tier architecture, the application first looks in the cache for the tuple of interest. If this tuple is not in the cache, then the application executes a query in the DBMS to fetch the desired data. Once the application receives this data from the DBMS, it updates the cache for fast access in the future. Whenever a tuple is modified in the database, the application must invalidate its cache entry so that the next time it is accessed the application will retrieve the current version from the DBMS. Many notable web

*Proceedings of the VLDB Endowment,* Vol. 6, No. 14



**(a)** Disk-oriented DBMS **(b)** Disk-oriented DBMS with a Distributed Cache **(c)** Main Memory DBMS with Anti-Caching

**Figure 1: DBMS Architectures** – In (a) and (b), the disk is the primary storage for the database and data is brought into main memory as it is needed. With the anti-caching model shown in (c), memory is the primary storage and cold data is evicted to disk.

sites, such as Facebook, use a large cluster of Memcached nodes in front of their sharded MySQL installation.

There are two problems with this two-tier model. First, data objects may reside both in the cache (in main memory format) and in the DBMS buffer pool (in disk format). This double buffering of data is a waste of resources. The second issue is that it requires developers to embed logic in their application to keep the two systems independently synchronized. For example, when an object is modified, the update is sent to the back-end DBMS. But now the states of the object in the DBMS and in the cache are different. If the application requires up-to-date values, the application must also update the object in the cache.

To overcome these problems, we present a new architecture for main memory DBMSs that we call *anti-caching*. In a DBMS with anti-caching, when memory is exhausted, the DBMS gathers the "coldest" tuples and writes them to disk with minimal translation from their main memory format, thereby freeing up space for more recently accessed tuples. As such, the "hotter" data resides in main memory, while the colder data resides on disk in the anti-cache portion of the system. Unlike a traditional DBMS architecture, tuples do not reside in both places; each tuple is either in memory or in a disk block, but never in both places at the same time. In this new architecture, main memory, rather than disk, becomes the primary storage location. Rather than starting with data on disk and reading hot data into the cache, data starts in memory and cold data is evicted to the anti-cache on disk.

This approach is similar to virtual memory swapping in operating systems (OS). With virtual memory, when the amount of data exceeds the amount of available memory, cold data is written out to disk in pages, typically in least recently used (LRU) order. When the evicted page is accessed, it is read back in, possibly causing other pages to be evicted. This allows the amount of virtual memory to exceed the amount of physical memory allocated to a process. Similarly, anti-caching allows the amount of data to exceed the available memory by evicting cold data to disk in blocks. If data access is skewed, the working set will remain in main memory.

With anti-caching, it is the responsibility of the DBMS to read and write data as needed. An alternative is to let the virtual memory system do the paging of the data to and from disk. Indeed, this is the approach taken in [28]. However, anti-caching has several advantages over virtual memory in the context of a main memory DBMS. In particular, it provides fine-grained control of the data evicted to disk and non-blocking reads of evicted data from disk. These two main differences are described in detail below:

**Fine-Grained Eviction:** A key advantage of anti-caching over virtual memory in the context of a main memory DBMS is the granularity at which data can be evicted. In anti-caching, eviction decisions are performed at the tuple-level. This means that the coldest tuples will be written to disk. In virtual memory, OS makes eviction decisions at the page-level. A virtual memory page is likely to be significantly larger than a typical OLTP tuple. Thus, each page selected for eviction will contain multiple tuples, each with potentially varying levels of coldness. A single hot tuple on a page will cause the entire page to be hot and kept in memory, even if the other tuples are cold. It is best to make evictions at the same level of granularity that the data is accessed, which in a DBMS is at the tuple level. Anti-caching provides a method for this finer-grained control of evicted data by building pages of cold tuples only.

**Non-Blocking Fetches:** Another difference is how evicted data is retrieved when it is needed. In a virtual memory system, the OS blocks a process when it incurs a page fault from reading a memory address that is on disk. For certain DBMSs [29, 34], this means that no transactions are executed while the virtual memory page is being fetched from disk. In an anti-caching DBMS, a transaction that accesses evicted data is simply aborted and then restarted at a later point once the data that it needs is retrieved from disk. In the meantime, the DBMS continues to execute other transactions without blocking. Lastly, since every page fault triggers a disk read, queries that access multiple evicted pages will page fault several times in a sequential fashion. We instead use a *pre-pass* execution phase that attempts to identify all evicted blocks needed by a transaction, which will allow all blocks to be read together [23].

In this paper, we explore the details of our anti-caching proposal. We have implemented a prototype in the H-Store DBMS [2] and performed a thorough experimental evaluation of the three different DBMS architectures depicted in Fig. 1:

1. Traditional, disk-based DBMS (MySQL).
2. Traditional, disk-based DBMS with a distributed cache front-end (MySQL + Memcached).
3. Anti-caching in a main memory DBMS (H-Store).

The results of these experiments show that the anti-caching architecture outperforms both the traditional disk-based and hybrid architecture on popular OLTP workloads. The difference is even more pronounced at higher skew levels, and demonstrates that main memory databases designed around the anti-caching architecture

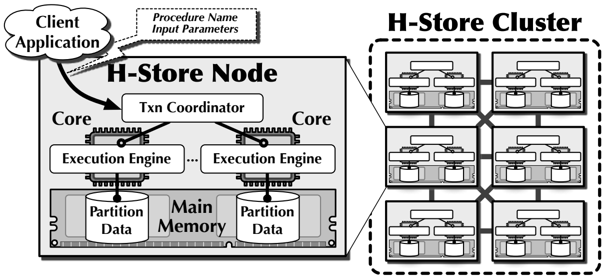


**Figure 2:** The H-Store Main Memory OLTP system.

can scale to significantly larger than the available main memory while experiencing minor throughput degradation.

Our anti-cache design is based on two key assumptions. Foremost is that our current prototype restricts the scope of queries to fit in main memory. We do not consider this a significant hindrance, since such large queries are uncommon in OLTP workloads. The other design assumption is that all indexes fit in memory. The trade-offs of using large secondary indexes is a well-studied topic in database optimization and we do not believe that this requirement is overly restrictive. We propose alternative designs to obviate the need to keep secondary indexes in memory.

## 2. H-STORE SYSTEM OVERVIEW

Before discussing the details of our anti-caching model, we first review H-Store's architecture and the motivations behind its design. In a disk-oriented DBMS, the system retrieves tuples from blocks on disk as they are requested by transactions. These blocks are stored in an in-memory buffer pool. If a transaction invokes a query that accesses data that is not in memory, the DBMS stalls that transaction until the block with that data is retrieved from disk and added to the buffer pool. If the buffer pool is full, then the DBMS chooses another block to evict to make room for the incoming one. Since the transaction waits until this disk operation completes, such systems employ a concurrency control scheme to allow other transactions to execute while the stalled one is waiting for the disk. The overhead of this movement of data and coordination between concurrent transactions has been shown to be significant [15].

This DBMS architecture made sense when compute nodes with enough RAM to store an entire database in memory were either non-existent or prohibitively expensive. But modern distributed DBMSs are able to store all but the largest OLTP databases entirely in the collective memory [29].

Given these observations, H-Store is designed to efficiently execute OLTP workloads on main memory-only nodes [18, 29]. As shown in Fig. 2, an H-Store *node* is a single physical computer system that manages one or more partitions. A partition is a disjoint subset of the data [26]. Each partition is assigned a single-threaded execution engine at its node that is responsible for executing transactions and queries for that partition.

Although H-Store supports ad hoc queries, it is primarily optimized to execute transactions as stored procedures. In this paper, we use the term *transaction* to refer to an invocation of a stored procedure. Stored procedures are an effective way to optimize OLTP applications because they execute entirely at the data node, thereby reducing the number of round-trips between the client and the database. A stored procedure contains *control code* (i.e., application logic) that invokes pre-defined parameterized SQL commands. A client application initiates a transaction by sending a request to any node in the cluster. Each transaction request contains the name of a stored procedure and the input parameters for that procedure's control code. H-Store assumes a workload of transactions with the following composition:

**Single-Partition Transactions:** In this case, there is a database design that allocates the various partitions of each table to nodes in such a way that most transactions are local to a single node [26]. Looking up a banking account balance or a purchase order is an example of a single-partition transaction.

A single-partition transaction is examined in the user-space H-Store client library, where parameters are substituted to form a runnable transaction. The user-level library is aware of H-Store's partitioning scheme [26], so the transaction can be sent to the correct node where it is executed from beginning to end without any blocking. Hence, single-partition transactions are serialized at each node, and any application that consists entirely of single-partition transactions will obtain maximum parallelism.

**Multi-Partition Transactions:** These transactions consist of multiple phases, each of which must be completed before the next phase begins. Moreover, one or more of the phases touches multiple partitions.

Each H-Store transaction is given a unique transaction ID, based on the time it arrived in the system. Standard clock-skew algorithms are used to keep the various CPU clocks synchronized. If a transaction with a higher transaction ID has already arrived at a node, then the incoming transaction is refused. In this way transactions are synchronized in timestamp order at the various nodes, without the need for any deadlock detection. Multi-Partition transactions use an extension of this protocol, where each local executor cannot run other transactions until the multi-partition transaction finishes execution. This scheme gives good throughput for workloads with a preponderance of single-partition transactions.

To ensure that all modifications to the database are durable and persistent, each DBMS node continuously writes asynchronous snapshots of the entire database to disk at fixed intervals [21, 29]. In between these snapshots, the DBMS writes out a record to a command log for each transaction that completes successfully [22]. The DBMS combines multiple records together and writes them in a group to amortize the cost of writing to disk [16, 34]. Any modifications that are made by a transaction are not visible to the application until this record has been written. This record only contains the original request information sent from the client, which is more lightweight than record-level logging [22].

## 3. ANTI-CACHING SYSTEM MODEL

We call our architecture anti-caching since it is the opposite architecture to the traditional DBMS buffer pool approach. The disk is used as a place to spill cold tuples when the size of the database exceeds the size of main memory. As stated earlier, unlike normal caching, a tuple is never copied. It lives in either main memory or the disk based anti-cache.

At runtime, the DBMS monitors the amount of main memory used by the database. When the size of the database relative to the amount of available memory on the node exceeds some administrator-defined threshold, the DBMS "evicts" cold data to the anti-cache in order to make space for new data. To do this, the DBMS constructs a fixed-size *block* that contains the least recently used (LRU) tuples from the database and writes that block to the anti-cache. It then updates a memory-resident catalog that keeps track of every tuple that was evicted. When a transaction accesses one of these evicted tuples, the DBMS switches that transaction into a "pre-pass" mode to learn about all of the tuples that the transaction needs. After this pre-pass is complete, the DBMS then aborts that transaction

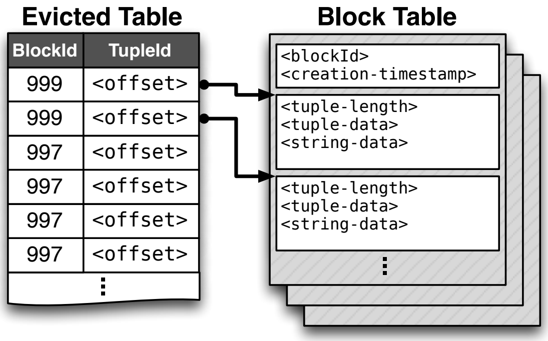


**Figure 3:** A logical representation of the layout of the in-memory Evicted Table and the disk-resident Block Table. The arrows represent integer offsets of tuples in a block.

(rolling back any changes that it may have made) and holds it while the system retrieves the tuples in the background. Once the data has been merged back into the in-memory tables, the transaction is released and restarted.

We now describe the underlying storage architecture of our anti-cache implementation. We then discuss the process of evicting cold data from memory and storing it in the non-volatile anti-cache. Then, we describe how the DBMS retrieves data from the anti-cache. All of the DBMS's operations on the anti-cache are transactional and any changes are both persistent and durable.

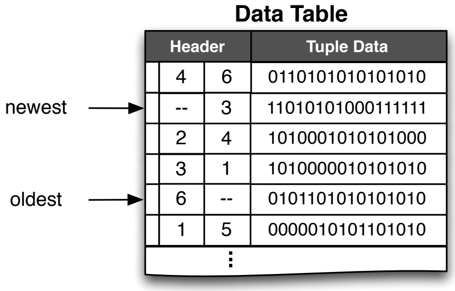
## 3.1 Storage Architecture

The anti-cache storage manager within each partition contains three components: (1) a disk-resident hash table that stores evicted blocks of tuples called the *Block Table,* (2) an in-memory *Evicted Table* that maps evicted tuples to block ids, and (3) an in-memory *LRU Chain* of tuples for each table. As with all tables and indexes in H-Store, these data structures do not require any latches since only one transaction is allowed to access them at a time.

One of the trade-offs that we need to consider is the storage overhead of this bookkeeping, given that the main goal of evicting tuples is to free up memory. Obviously the amount of memory used to keep track of evicted tuples should only be a small fraction of the memory gained from evicting tuples. Our current implementation also requires that all of the database's primary key and secondary indexes fit in memory. We explore this issue further in Section 5.6.

**Block Table:** This is a hash table that maintains the blocks of tuples that have been evicted from the DBMS's main memory storage. Each block is the same fixed-size and is assigned a unique 4-byte key. A block's header contains the identifier for the single table that its tuples were evicted from and the timestamp when the block was created. The body of the block contains the serialized evicted tuples from a single table. Every tuple stored in a block is prefixed with its size and is serialized in a format that closely resembles its in-memory format (as opposed to a format that is specifically designed for disk-based storage). The key portion of the Block Table stays in memory while its values (i.e., the block data) are stored on disk without OS or file-system caching.

**Evicted Table:** The Evicted Table keeps track of the tuples that have been written out to blocks on disk. When a tuple is evicted, the DBMS removes it from the regular storage space for tables and adds it to a dynamically-constructed block that is then stored in the Block Table. Each evicted tuple in a block is assigned a 4-byte identifier that corresponds to its offset in the block it resides in. The DBMS updates any indexes containing evicted tuples to reference the Evicted Table. As discussed in Section 3.4, the Evicted Table ensures that the DBMS is able to identify all of the evicted tuples that are needed by a transaction.



| Header | | | Tuple Data |
|---|---|---|---|
| | 4 | 6 | 0110101010101010 |
| | -- | 3 | 1101010100011111 |
| | 2 | 4 | 1010001010101000 |
| | 3 | 1 | 1010000010101010 |
| | 6 | -- | 0101101010101010 |
| | 1 | 5 | 0000010101101010 |
| | ⋮ | | |

**Figure 4:** Physical representation of the LRU Chain embedded in the tuple headers. Each tuple header contains 1 byte for bit flags (left-most box) followed by two 4-byte tuple IDs of the tuples adjacent in the linked list.

**LRU Chain:** Lastly, H-Store also maintains an in-memory list of all the tuples for each table in LRU order. This allows the DBMS to quickly ascertain at runtime the least-recently used tuples to combine into a new block to evict. The LRU Chain is a doubly-linked list where each tuple points to the next and previous most-recently used tuple for its table. Tuples are added to the tail of the chain whenever they are accessed, modified, or inserted by a transaction. When a tuple is read or updated, it is first removed from its original location in the chain and inserted at the back. The tuples that were previously adjacent to it in the chain are then linked to each other.

Rather than maintain a separate data structure for the LRU Chain, the DBMS embeds the pointers directly in the tuples' headers. To reduce the memory overhead of this, the pointer for each tuple is a 4-byte offset of that record in its table's memory at that partition (instead of an 8-byte address location).

To reduce the CPU overhead of tracking the total ordering of each table's LRU Chain, the DBMS selects a fraction of the transactions to monitor at runtime. The selected transactions are used to update data in the LRU Chain. Because hot tuples are, by definition, accessed more frequently, they are more likely to be accessed in the transactions sampled and thus are more likely to be updated in the LRU Chain. The rate at which transactions are sampled is controlled by parameter $\alpha$, where $0 < \alpha \leq 1$. We explore the affect of sampling and other trade-offs in Section 5.4.

In addition, there are often tables that are accessed frequently and should not be allowed to be evicted to disk (e.g., small lookup tables). Because these tables would be considered hot, it is unlikely that any portion of such a table would be evicted to disk. Still, there is added overhead of maintaining the LRU chain for such tables. To remove this, tables can be specifically flagged as *evictable* during schema creation. Any table not labeled as *evictable* will not maintain an LRU chain and will remain entirely in main memory.

## 3.2 Block Eviction

Ideally, our architecture would be able to maintain a single global ordering of tuples in the system, thus globally tracking hot and cold data. However, the costs of maintaining a single chain across partitions would be prohibitively expensive due to the added costs of inter-partition communication. Instead, our system maintains a separate LRU Chain per table that is local to a partition. Thus, in order to evict data the DBMS must determine (1) what tables to evict data from and (2) the amount of data that should be evicted from a given table. For our initial implementation, the DBMS answers these questions by the relative skew of accesses to tables. The amount of data accessed at each table is monitored, and the

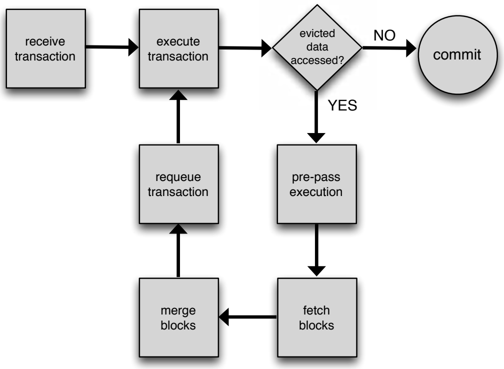


**Figure 5: Transaction Execution State Diagram** – If the transaction accesses evicted data, then the transaction enters pre-pass execution, fetches and merges the data before the transaction is requeued.

amount of data evicted from each table is inversely proportional to the amount of data accessed in the table since the last eviction. Thus, the hotter a table is, the less data will be evicted. For the benchmarks tested, this approach is sufficient, but we expect to consider more sophisticated schemes in the future.

After determining how much data to evict from each table, H-Store executes special single-partition transactions that select tuples for eviction and writes blocks to disk. Since transactions are executed one-at-a-time at each partition, these eviction transactions automatically block all other transactions at their target partition without needing any additional locking mechanisms.

When the eviction transaction executes, it creates a new block by popping tuples off the head of the target table's LRU Chain. For each tuple being evicted, H-Store copies its data into the eviction block buffer. It then adds an entry into the Evicted Table and updates all indexes to point to this entry instead of the original tuple location. Each tuple in the Evicted Table includes a special *evicted* flag in its header that enables the DBMS to recognize when a transaction accesses evicted data. This eviction process continues until the block is full, at which point the transaction will create the next block. The process stops once the transaction has evicted the requisite amount of data from each table. Groups of blocks are written out in a single sequential write. For example, if the table is asked to evict a set of $n$ blocks, it will create each of the $n$ blocks independently, and only when all $n$ blocks have been created will it write the result to disk in one sequential write.

It is also important to note that the state of the database is consistent during the eviction process. Although indexes are updated and the tuple is removed from the original table before the block is written to disk, the single-threaded nature of the execution engine means that no other transactions access these changes until the special transaction finishes. Other transactions will not execute until the entire set of blocks requested for eviction are written to disk. Also, at no point during this process is data un-recoverable if the DBMS crashes (see Section 3.6).

## 3.3 Transaction Execution

Main memory DBMSs, like H-Store, owe their performance advantage to processing algorithms that assume that data is in main memory. But any system will slow down if a disk read must be processed in the middle of a transaction. This means that we need to avoid stalling transaction execution at a partition whenever a transaction accesses an evicted tuple. We now describe how this is accomplished with anti-caching.

A query can access evicted data through either an index or a sequential look-up (i.e., a full table scan). For the latter, the DBMS will need to store the entire table in memory, which may exceed the physical memory available. We discuss this problem in Section 6.1.

For index look-up queries, the system searches the target index to find the keys that match the query's predicate. Each key in the index points to a tuple that is either in the normal table storage or in the Evicted Table. If none of the accessed tuples are evicted, then the DBMS allows the transaction to continue. If evicted data is needed, the transaction will then enter a special phase to determine exactly which data is needed and where that data exists on disk.

**Pre-pass Phase:** A transaction enters the *pre-pass phase* if evicted data is needed to continue execution. The goal of the pre-pass phase is to determine all of the evicted data that the transaction needs to access so that it can be retrieved together. To do this, the transaction executes as normal, except that the DBMS checks the *evicted* flag for each tuple that it accesses to determine whether the tuple has been evicted. If it has, then the DBMS records the evicted tuple's block ID and offset from the Block Table (see Fig. 3). When pre-pass has finished execution, the DBMS rolls back any changes that the transaction made at any partition and then re-queues the transaction along with the list of evicted tuple identifiers that it attempted to access during the pre-pass phase. Also, during the pre-pass phase, any in-memory tuples are updated in the LRU Chain to reduce the likelihood that these tuples are evicted before the transaction is re-queued. This minimizes the possibility of a transaction being restarted multiple times due to evicted data.

Although it is small, the overhead of aborting and restarting transactions is not zero. Thus, in the pre-pass phase, the DBMS attempts to identify all of the data that a transaction needs by allowing that transaction to continue executing after it encounters an evicted tuple [23]. This allows the DBMS to batch fetch requests and minimize the possibility of restarting a transaction multiple times. In contrast, in the event of a page fault in virtual memory, execution halts for each individual evicted page access [28].

For some transactions, it is not possible for the DBMS to discover all of the data that it needs in a single pre-pass. This can occur if the non-indexed values of an evicted tuple are needed to retrieve additional tuples in the same transaction. In this case, the initial pre-pass phase will determine all evicted data that is not dependent on currently evicted data. Once this data is successfully merged and the transaction is restarted, this unevicted data will be used to resolve any data dependencies and determine if any additional data needs to be unevicted. From our experience, however, we believe that such scenarios are rare. The more typical access pattern is that a transaction retrieves the key of a record from a secondary index, in which case the DBMS will still be able to run the transaction in the pre-pass phase because the indexes always remain in memory.

We next describe how the DBMS retrieves the evicted tuples identified during the pre-pass and merges them back into the system's in-memory storage.

## 3.4 Block Retrieval

After aborting a transaction that attempts to access evicted tuples, the DBMS schedules the retrieval of the blocks that the transaction needs from the Block Table in two steps. The system first issues a non-blocking read to retrieve the blocks from disk. This operation is performed by a separate thread while regular transactions continue to execute at that partition. The DBMS stages these retrieved blocks in a separate buffer that is not accessible to queries. Any transaction that attempts to access an evicted tuple in one of these blocks is aborted as if the data was still on disk.

Once the requested blocks are retrieved, the aborted transaction is then rescheduled. Before it starts, the DBMS performs a "stop-and-copy" operation whereby all transactions are blocked at that partition while the unevicted tuples are merged from the staging buffer back into the regular table storage. It then removes all of the entries for these retrieved tuples in the Evicted Table and then updates the table's indexes to point to the real tuples.

The key issue that we must consider during this step is on how much data to merge from a retrieved block back into the in-memory storage. For example, the DBMS can choose to merge all of the tuples from the recently retrieved block or just the tuple(s) that the previous transaction attempted to access that caused the block to be retrieved in the first place. We now discuss two different solutions for this problem. We compare the efficacy and performance of these approaches in Section 5.1.

**Block-Merging:** The simplest method is for the DBMS to merge the entire retrieved block back into the regular table storage. All of the tuples in the block are inserted back into the in-memory table. The requested tuple(s) are placed at the back of the table's LRU Chain. Conversely, any tuples not needed by pending transactions are added to the front (i.e., cold end) of the LRU Chain, which means that they are more likely to be chosen for eviction in the next round. This ensures that only the tuples that were needed by the transaction that caused the block to be un-evicted become hot, whereas the rest of the block is still considered cold. After the DBMS merges the tuples from the block, it can delete that block from the Evicted Table.

The overhead of merging all the tuples from the un-evicted block can be significant, especially if only a single tuple is needed from the block and all of the other tuples are re-evicted shortly thereafter. In the worst case, there is a continuous un-eviction/re-eviction cycle, where unwanted tuples are brought into the system and then immediately re-evicted.

**Tuple-Merging:** To avoid this oscillation, an alternative strategy is to only merge the tuples that caused the block to be read from disk. When a block is retrieved from disk, the DBMS extracts only the tuples that are needed from that block (based on their offsets stored in the Evicted Table) and then only merges those tuples back into the in-memory table. Once the desired tuples are merged, the fetched block is then discarded without updating the block on disk. This reduces the time of merging tuples back into their tables and updating their indexes. It now means that there are now two versions of the tuple, the one in memory and the stale one in the anti-cache on disk. But since the DBMS removes the merged tuples' from the Evicted Table, all subsequent look-ups of these tuples will use the in-memory version. If this block is ever fetched again, the stale entries of the already unevicted tuples are ignored.

Over time, these "holes" in the blocks accumulate. This means the amount of valid data that is retrieved in each block is reduced. We employ a *lazy block compaction* algorithm during the merge process. This compaction works by tracking the number of holes in each of the blocks in the Block Table. When the DBMS retrieves a block from disk, it checks whether the number of holes in a block is above a threshold. If it is, then the DBMS will merge the entire block back into the memory, just as with the block-merge strategy. We discuss more sophisticated approaches in Section 6.2.

## 3.5 Distributed Transactions

Our anti-caching model also supports distributed transactions. H-Store will switch a distributed transaction into the "pre-pass" mode just as a single-partition transaction when it attempts to access evicted tuples at any one of its partitions. The transaction is aborted and not requeued until it receives a notification that all of the blocks that it needs have been retrieved from the nodes in the cluster. The system ensures that any in-memory tuples that the transaction also accessed at any partition are not evicted during the time that it takes for each node to retrieve the blocks from disk.

## 3.6 Snapshots & Recovery

Persistence and durability in disk-based systems is typically achieved using a combination of on-disk data and logging. In a main memory DBMS, however, other techniques such as snapshots and command logging [22, 29] are used. This does not change for a DBMS with anti-caching, except that now the system must also snapshot the additional data structures discussed Section 3.1.

To do this, the DBMS serializes all the contents of the regular tables and index data, as well as the contents of the Evicted Table, and writes it to disk. At the same time, the DBMS also makes a copy of the Block Table on disk as it existed when the snapshot began. No evictions are allowed to occur in the middle of a snapshot. To recover after a crash, the DBMS loads in the last snapshot from disk. This will set up the tables, indexes, Block Table, and Evicted Table as it existed before the crash. The DBMS then replays the transactions in the command log that were created after this snapshot was taken. With this process, all anti-caching data is persistent and the exact state of a system is recoverable in the event of a crash.

Making a snapshot of the Block Table could be prohibitively expensive for large data sizes. Instead of making copies for each checkpoint, the DBMS takes *delta snapshots*. Because the data within a block in the Block Table is not updated, the DBMS just checks to see which blocks were added or removed from the Block Table since the last snapshot. This technique greatly reduces the amount of data copied with each snapshot invocation.

# 4. ARCHITECTURE COMPARISON

To evaluate our anti-caching model, we implemented a prototype in H-Store and compared its performance against MySQL, an open-source, disk-oriented DBMS. We tested MySQL with and without Memcached [14] as a front-end distributed cache.

We first describe the two benchmarks and the three DBMS configurations that we used in this analysis.

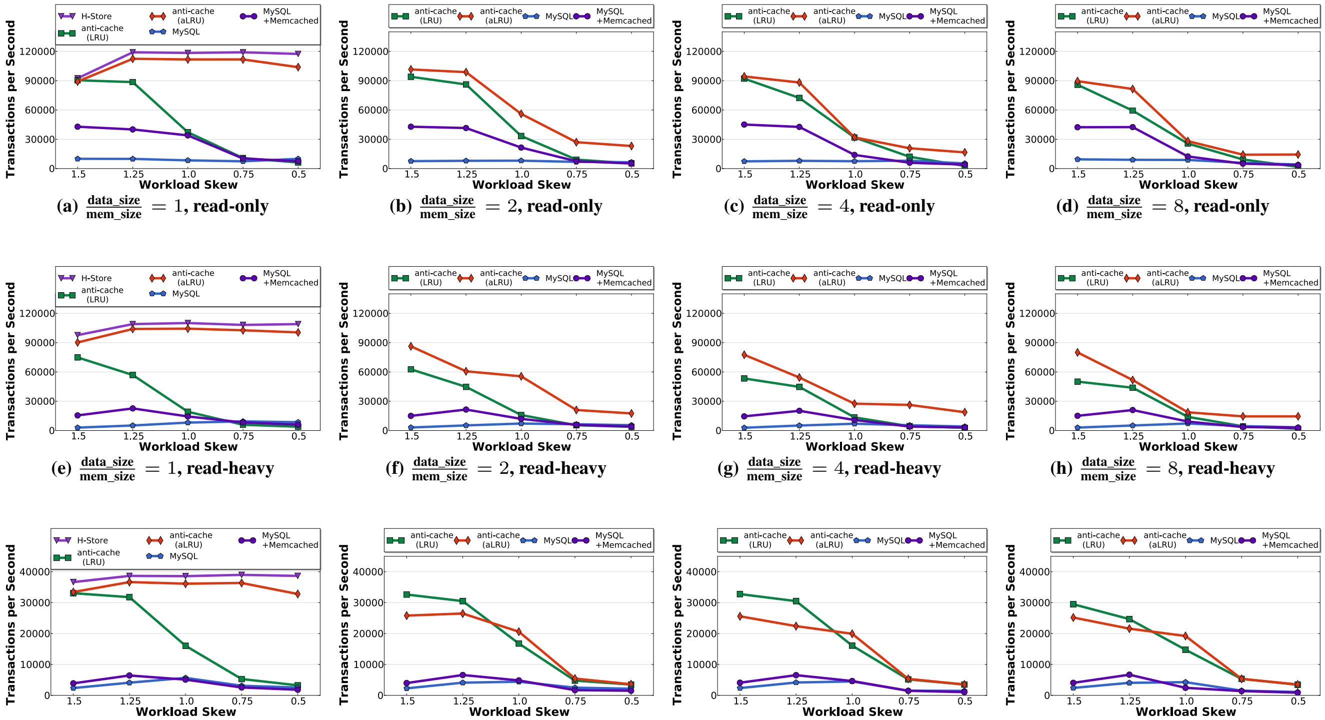
## 4.1 Benchmarks

We used the OLTP-Bench [10] framework for the MySQL experiments and H-Store's built-in benchmarking framework for the anti-caching experiments.

**YCSB:** The Yahoo! Cloud Serving Benchmark is a collection of workloads that are representative of large-scale services created by Internet-based companies [9]. For all of the YCSB experiments in this paper, we used a ~20GB YCSB database containing a single table with 20 million records. Each YCSB tuple has 10 columns each with 100 bytes of randomly generated string data. The workload consists of two types of transactions; one that reads a single record and one that updates a single record. We use three different transaction workload mixtures:

- **Read-Heavy:** 90% reads / 10% updates
- **Write-Heavy:** 50% reads / 50% updates
- **Read-Only:** 100% reads

We also vary the amount of skew in workloads to control how often a tuple is accessed by transactions. For these experiments, we use YCSB's Zipfian distribution as it is emblematic of skewed workloads where older items are accessed much less frequently than newer items. The amount of skew in the Zipfian distribution is controlled by the constant $s$, where $s > 0$. Higher values of $s$



(a) $\frac{\text{data\_size}}{\text{mem\_size}} = 1$**, read-only** (b) $\frac{\text{data\_size}}{\text{mem\_size}} = 2$**, read-only** (c) $\frac{\text{data\_size}}{\text{mem\_size}} = 4$**, read-only** (d) $\frac{\text{data\_size}}{\text{mem\_size}} = 8$**, read-only**

(e) $\frac{\text{data\_size}}{\text{mem\_size}} = 1$**, read-heavy** (f) $\frac{\text{data\_size}}{\text{mem\_size}} = 2$**, read-heavy** (g) $\frac{\text{data\_size}}{\text{mem\_size}} = 4$**, read-heavy** (h) $\frac{\text{data\_size}}{\text{mem\_size}} = 8$**, read-heavy**

(i) $\frac{\text{data\_size}}{\text{mem\_size}} = 1$**, write-heavy** (j) $\frac{\text{data\_size}}{\text{mem\_size}} = 2$**, write-heavy** (k) $\frac{\text{data\_size}}{\text{mem\_size}} = 4$**, write-heavy** (l) $\frac{\text{data\_size}}{\text{mem\_size}} = 8$**, write-heavy**

**Figure 6:** YCSB experiments. In aLRU, $\alpha = 0.01$.

signify higher skews. In our experiments, we use a Zipfian skew with values of $s$ between 0.5 and 1.5.

**TPC-C:** This benchmark is the current industry standard for evaluating the performance of OLTP systems [32]. It consists of nine tables and five procedures that simulate a warehouse-centric order processing application. Only two of these procedures modify or insert tuples in the database, but they make up 88% of the benchmark's workload. For our experiments, we used a ∼10GB TPC-C database containing 100 warehouses and 100,000 items. We configured both the H-Store and OLTP-Bench benchmark frameworks such that each transaction only accesses data from a single warehouse (i.e., there are no distributed transactions).

We must further decide which tables will be designated as *evictable*. Some of the tables in the TPC-C benchmark are called *lookup tables* and contain only static data. For example the `CUSTOMERS`, `DISTRICT`, and `WAREHOUSE` tables fall into this category. Once initially loaded, no new data is added to these tables. Also, these tables are used by a majority of the transactions in the workload, and are unlikely to be evicted. Thus, we did not mark them as *evictable*. This allows the system to not maintain a LRU Chain for these tables. In most real-world deployments, static lookup tables on the order of a few gigabytes will easily fit in memory. Thus, these tables will not be evicted and will reside in memory throughout the duration of the benchmark.

On the other hand, some tables are used to record orders, but this data is not read by transactions in the future. These include the `HISTORY`, `ORDERS` and `ORDER_LINE` tables. It is these tables that cause a TPC-C database to grow over time. In our benchmark, these tables are labeled as *evictable*. For the benchmark, we set the available memory to the system to 12GB. This allows all static tables to fit in memory. As the benchmark progresses and more orders accumulate, the data size will continue to grow, eventually exhausting available memory, at which point the anti-caching architecture will begin evicting cold data from the *evictable* tables to disk.

## 4.2 System Configurations

All three systems were deployed on a single node with a dual-socket Intel Xeon E5-2620 CPU (12 cores per socket, 15M Cache, 2.00 GHz) processor running 64-bit Ubuntu Linux 12.04. The data for each respective DBMS was stored on a single 7200 RPM disk drive. According to *hdparm*, this disk delivers 7.2 GB/sec for cached reads and about 297 MB/sec for buffered reads. All transactions were executed with a serializable isolation level.

**MySQL:** We used MySQL (v5.6) with the InnoDB storage engine. We tuned MySQL's configuration to optimize its execution for the type of short-lived transactions in our target benchmarks. In particular, we also used 512 MB log file cache and 10 MB query cache. We configured InnoDB's buffer pool according to the workload size requirement for the different experiments. We did not limit the number of CPU cores that the DBMS is allowed to use.

**MySQL + Memcached:** In our second configuration, we used MySQL with Memcached (v1.4) deployed on the same node. We modified the transaction code for the different benchmarks to store cached query results in Memcached as serialized JSON objects. As described below, the amount of memory allocated to Memcached is based on the working set size of the benchmark.

The primary benefit of using Memcached as a front-end to MySQL is to improve the performance of read queries. Because Memcached does not use heavyweight locking, simple key-based lookups of cached tuples can be faster than in MySQL. However, writes must be propagated to both the Memcached frontend and MySQL backend, thus incurring additional overhead.

**H-Store with Anti-Caching:** Lastly, we deployed the latest version of H-Store [2] that uses our anti-caching prototype. In our current implementation, we use BerkeleyDB's hash table to store the Block Table [24]. BerkeleyDB is configured to use direct I/O without any caching or locks. We split each benchmark's database into

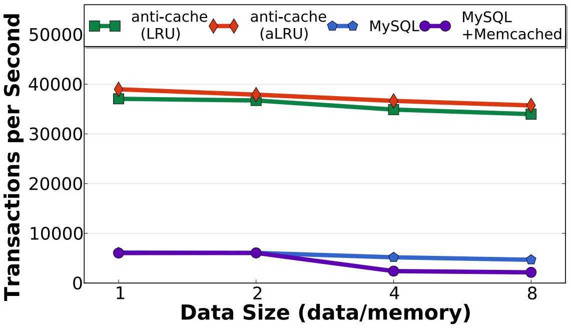


**Figure 7:** TPC-C experiments.

six partitions using a partitioning scheme that makes all transactions single-partitioned [26]. We configured the memory threshold for the system according to the workload size requirement for the different experiments. We set the system to check its database size every 5 seconds and evict data in 1MB blocks.

The benchmark clients in each experiment are deployed on a separate node in the same cluster. For each experiment, we execute the benchmarks three times and report the average throughput of these trials. In each trial, the DBMSs are allowed to "warm-up" for two minutes. Empirically, we found that sustained transaction throughput had stabilized within this period. During the warm-up phase, transactions are executed as normal but throughput is not recorded in the final benchmark results. For H-Store, cold data is evicted to the anti-cache and hot data is brought into memory. For MySQL, hot data is brought into the buffer pool. For the Memcached deployment, the client pre-loads relevant objects into Memcached's memory. After the warm-up, each benchmark is run for a duration of five minutes, during which average throughput is recorded. The final throughput is the number of transactions completed in a trial run divided by the total time (excluding the warm-up period). Each benchmark is run three times and the throughputs from these runs are averaged for a final result.

For the anti-caching architecture, we evaluate H-Store's performance using a LRU Chain sampling rate of $\alpha = 0.01$ (aLRU) and $\alpha = 1.00$ (LRU). Thus, for aLRU, only one out of every one hundred transactions updates the LRU chain while for LRU every transaction will update the LRU chain.

## 4.3 Results & Discussion

We now discuss the results of executing the two benchmarks in Section 4.1 on the three DBMS architectures across a range of workload skew and data size configuration.

**YCSB:** The results in Fig. 6 are for running the YCSB benchmark with all three workload types (read-only, read-heavy, write-heavy) across a range of data sizes and workload skews. These results show that as database size increases relative to the amount of memory in the system, the throughput of all three systems degrades, since they perform more disk reads and writes. Similarly, as the skew decreases, their performance also degrades since transactions are more likely to access tuples that are evicted and need to be retrieved from disk.

We observe, however, that for highly-skewed workloads (i.e., workloads with skews of 1.5 and 1.25) the anti-caching architecture outperforms MySQL by a factor of 9× for read-only, 18× for read-heavy, and 10× on write-heavy workloads for datasets 8× memory. For the same high skews, out anti-caching architecture outperforms the hybrid MySQL + Memcached architecture by a factor of 2× for read-only, 4× for read-heavy, and 9× on write-heavy workloads for datasets 8× memory. There are several reasons for this. One is that H-Store's lightweight concurrency control scheme is more efficient than MySQL's model [15]. Another advantage is that tuples are not converted back-and-forth between disk and main memory format. Evicted tuples are copied in and out of eviction blocks as contiguous chunks of memory. Also, in an anti-caching architecture, eviction blocks are composed dynamically of cold tuples, rather than evicting fixed blocks which could contain some relatively hot data. Hence, anti-caching provides finer-grained control of the bytes evicted to disk.

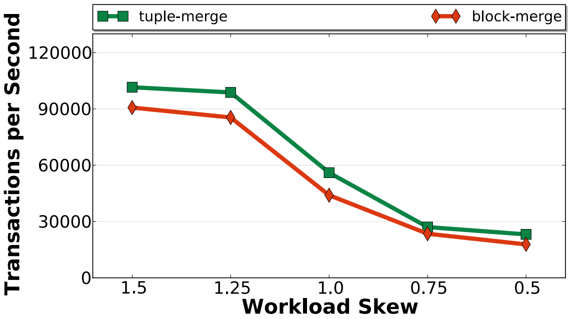
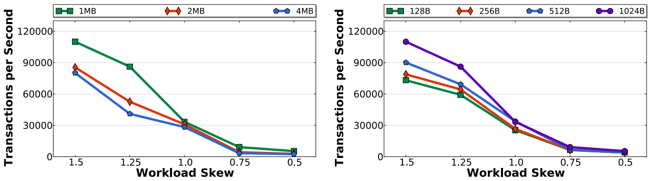


**Figure 8: Merge Strategy Analysis** – YCSB read-only, 2× memory, 1MB evict blocks.

There are several interesting results regarding the MySQL benchmarks. One is that that Memcached improves the throughput of MySQL most on the read-only workloads and only for high skew. The lower performance in the other workloads is due to the overhead of synchronizing values in Memcached and in MySQL in the event of a write. For low skew workloads, there is a high cost of cache misses in this hybrid architecture. If Memcached is queried and does not contain the requested data, the application must then query MySQL, resulting in a cache miss. If Memcached is queried and contains the requested data (i.e. a cache hit), the MySQL backend is not queried at all. Because of the lower overhead of Memcached over MySQL, the benefits of a cache hit can be significant. However, for the OLTP benchmarks tested, tuples are relatively small and queries are relatively simple, so the cost of a cache miss outweighs the cost of a cache hit. It is only in read intensive, higher skewed workloads (where the likelihood of a cache hit is higher) that hybrid architecture outperforms standalone MySQL. Also noteworthy is that for workloads with writes, MySQL actually performs worse for skews of 1.5 and 1.25. This results in higher lock contention for hot tuples in a disk-based DBMS that uses heavyweight locking and latching.

Another result is that for almost all data sizes and skews tested, aLRU performs as well or better than the standard LRU. As discussed previously, sampling of transactions is an effective way to capture workload skew and is able to significantly lessen the overhead of maintaining the LRU chain. Default H-Store provides an effective baseline by which to compare the overheads of the anti-caching components. In Figs. 6a, 6e and 6i, we see where the throughput of the aLRU implementation is close to H-Store baseline, ranging from a 2–7% throughput overhead. Conversely, the anti-caching with traditional LRU suffers significantly as skew is decreased, meaning the maintenance of the LRU chain is a major bottleneck at lower skews.

**TPC-C:** The results for TPC-C are shown in Fig. 7. Because the anti-cache architecture is able to efficiently evict cold data from the tables that are growing (i.e., `HISTORY`, `ORDERS` and `ORDER_LINE`) the throughput declines little. In TPC-C, the only transaction that potentially accesses evicted data is the `Order-Status` transaction.



**(a)** Block Size **(b)** Tuple Size

**Figure 9: Block and Tuple Size Analysis** – YCSB read-only workload with 2× data size.

However, this transaction is only 4% of the workload and reads the *most* recent order for a given customer. Thus, the data that these transactions need is unlikely to be evicted, meaning the slight decrease in throughput for anti-caching is not due to unevicting data. Instead, it is a result of the increasing memory overhead as the amount of evicted data grows, since there is an entry in the Evicted Table for each evicted tuple. Due to the amount of writes, the hybrid architecture performs worse than stand-alone MySQL when data size is larger than memory. Overall, anti-caching provides a 7× improvement in throughput over the other architectures.

# 5. EXPERIMENTAL ANALYSIS

We also conducted additional experiments to evaluate our design and test its sensitivity to changes in key parameters. These experiments were conducted on the same hardware configuration used for the system comparison study in Section 4.

## 5.1 Merge Strategies

We first compare the two block retrieval strategies from Section 3.4: (1) block-merge and (2) tuple-merge. For this experiment, we use the YCSB read-only workload at 2× memory with an eviction block size of 1MB. The tuple-merge *fill-factor* (i.e., when the lazy compaction merges the entire block into memory) is set to 0.50, meaning that each block can contain no more than 50% holes.

The results in Fig. 8 show that across the skews tested the tuple-merge policy outperforms the block-merge policy. There are two reasons for this. First is that the larger merge costs of the block-merge policy, shown in Fig. 11a. Because merging tuples blocks transactions from executing on the target partition, this can negatively affect throughput. Second is that in the block-merge policy, unrequested tuples (i.e., tuples that were part of the fetched block but were not requested by a transaction) are merged and placed at the cold end of the LRU Chain. Thus, these tuples were unevicted only to shortly thereafter be evicted once again. This uneviction/re-eviction cycle creates unnecessary overhead and is another reason for the lower throughput of the block-merge policy.

## 5.2 Evicted Table Block Size

We next investigate the impact on performance of different Evicted Table block sizes. This parameter controls how many tuples are in each evicted block. Because we have already shown the advantage of the tuple-merge policy over block-merge in Section 5.1, we only evaluate the tuple-merge policy. In this experiment, we use the read-only YCSB workload and with a database size of 2× memory.

The results in Fig. 9a show that larger block sizes reduce overall throughput, especially for highly skewed workloads. The throughput degradation is not due to the eviction process, which evicts batches of blocks in a single sequential disk write. Thus, writing five 1MB blocks or twenty 256KB blocks is nearly equivalent in terms of I/O cost. The main difference is due to the added costs of fetching larger blocks.

Another result from this experiment is that the difference in throughput for larger block sizes is most pronounced at higher skewed workloads. The main reason for this is that with a highly skewed workload, the DBMS needs to retrieve fewer blocks from disk. Because each block is unlikely to be retrieved from disk, it is also relatively less common that multiple tuples from a single block will be requested together. Thus, the system is less likely to benefit from the locality of tuples on the same block.

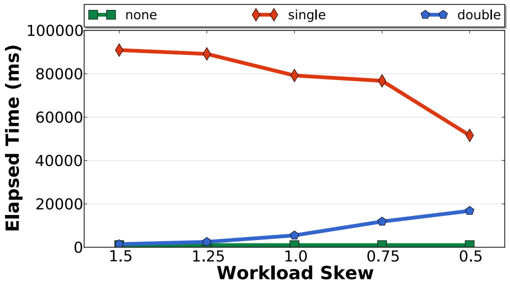


**Figure 10:** Eviction Chain Analysis.

## 5.3 Tuple Size

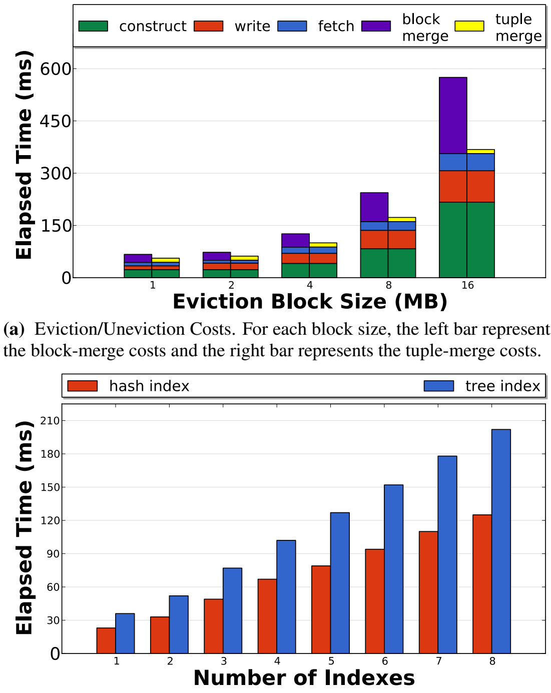
Another important factor in the performance of a DBMS with anti-caching is tuple size. The memory overhead of anti-caching's internal data structures is much greater for smaller tuples than for large tuples. Also, when evicting large blocks of smaller tuples, the CPU overhead of eviction could be significant, because the DBMS must update indexes and the Evicted Table for each evicted tuple. The cost of eviction from the LRU Chain is constant regardless of tuple size. Thus, to measure the affect of tuple size we will use the read-heavy YCSB workload with 2× data size and 1MB block sizes. We vary size of tuples in each trial from 128B to 1024B.

The results in Fig. 9b show that the DBMS achieves higher throughputs for the larger tuple sizes. This may seem counterintuitive, but the reason is because there is a small but unavoidable memory overhead for eviction per tuple. Thus, with smaller tuples anti-caching is able to reclaim less memory with each tuple eviction. This means that to reclaim a fixed amount of memory, more tuples need to be evicted. However, evicting more tuples increases the CPU resources consumed. This additional cost degrades the DBMS's throughput since transaction processing at a partition is blocked while the eviction process executes.

## 5.4 LRU Chain Overhead

We next analyze the internal bookkeeping in anti-caching used to keep track of tuple accesses. We compare the DBMS's performance using either a *doubly-linked* or a *singly-linked* list for the LRU Chain. As discussed in Section 3.1, there is an inherent trade-off between the amount of memory used to track the LRU ordering of tuples in a table and the cost of maintaining that ordering. To show this, we implemented a micro-benchmark in H-Store's execution engine that updates 100,000 tuples and reports the total elapsed time needed to update the LRU Chain. As a baseline, we compare against the cost when anti-caching is disabled, and thus no eviction chain is maintained.

The results are shown in Fig. 10. The baseline is constant across all skew levels, as expected. For higher skewed workloads, the doubly-linked list performs within 5% of the baseline and 20× faster than the singly-linked list. The two strategies slowly converge as skew is decreased. The difference in performance between the singly-linked list and doubly-linked list is due to the high cost of updating a tuple in the chain. Choosing a tuple for eviction involves removing the front tuple of the chain, which can be done in $\mathcal{O}(1)$. Similarly, adding a tuple to the back of the chain can also be done in constant time. However, updating a tuple could,



**(a)** Eviction/Uneviction Costs. For each block size, the left bar represents the block-merge costs and the right bar represents the tuple-merge costs.

**(b)** Index Update Costs. Elapsed time represents cost of updating the specified number of indexes for a block of 1,000 tuples.

**Figure 11:** Eviction and Uneviction analysis.

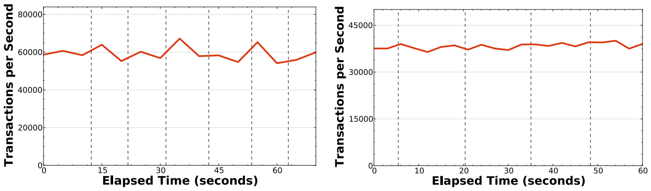
in the worst case, involve scanning the entire chain to find the tuple. This is an $\mathcal{O}(n)$ operation, where $n$ is the number of tuples in the chain. For high skew workloads, it is likely that the hot tuples that are being updated frequently will be found at the back of the chain, since the chain is ordered from coldest to hottest. Thus, it is better to scan the chain from back-to-front rather than from front-to-back, necessitating a doubly-linked list. We contend that the added memory overhead of a doubly-linked list is a necessary trade-off to optimize for skewed workloads.

## 5.5 Eviction Overhead Micro-benchmark

We created another micro-benchmark to compare the relative costs of evicting and unevicting an anti-cache block for a fixed-size 1KB tuple. The engine first reads the oldest tuples from one table, copies them into an eviction block, updates an index, and then writes the block to disk. Then, the engine reads that block back from disk, copies the data back into the table, and updates the index. This constitutes a single run of the benchmark. This is repeated three times and the run times averaged for the final result.

The results in Fig. 11a show that the cost of updating indexes and copying data to and from disk scales linearly relative to the block size. Although in this experiment the *construct* and *merge* phases take longer than the disk I/O operations, this experiment was conducted with no other disk traffic.

Additionally, we created a micro-benchmark to analyze the cost of updating indexes as the number of indexes is increased, since it is not uncommon for OLTP tables to have more than one secondary index. In our analysis, for completeness we test with up to eight indexes, but acknowledge this is more indexes than would be likely in practice due to the high costs of maintaining secondary indexes in any DBMS, independent from anti-caching. Each benchmark updates 1,000 tuples in a number of indexes varied from one to eight. The choice of 1,000 tuples represents the number of tuples in a single eviction block assuming 1KB tuples and a 1MB block. Both a hash index and a balanced tree index were used. Reported times represent the total time required to update all 1,000 tuples in each of the $n$ indexes, averaged over three runs of the benchmark. The results are seen in Fig. 11b. The performance is shown to scale linearly with the number of indexes and even for a larger number of indexes elapsed time is reasonable. We conclude that the update of secondary indexes is unlikely to be a bottleneck for most OLTP workloads.



**(a)** YCSB

**(b)** TPC-C

**Figure 12:** Eviction overhead measurements for a 60 second interval of the TPC-C and YCSB (read-heavy, 2× memory, 1.0 skew) benchmarks. Each vertical line represent a point in time when block(s) were evicted.

## 5.6 Overhead Measurements

Lastly, we measured the affect of evictions on the sustained throughput in the system. For this, we record the throughput of the YCSB and TPC-C benchmarks over time, also recording when evictions occur. The graphs in Fig. 12 show a timeline view of the throughput of our anti-caching implementation while it is evicting and unevicting data. The vertical lines represent when an eviction occurs in the system. Eviction happens less frequently in TPC-C because data takes longer to accumulate in memory due to few unevictions. Also due to the few unevictions in TPC-C, throughput is less volatile over time compared to YCSB. There are several reasons for throughput volatility. One is group commit of writes, which commits transactions in large batches, thereby making the throughput more volatile over time. Eviction is another factor, and the throughput can be seen to decrease during an eviction. This decrease is caused by the creation of the eviction block, which must block other transactions. However, once created, writing of the eviction block to disk is done asynchronously.

# 6. FUTURE WORK

We now discuss several extensions to our anti-caching model that we plan to investigate in the future.

## 6.1 Larger-than-Memory Queries

Anti-caching allows main memory DBMSs to manage databases that are larger than the amount of collective memory at all nodes. Our current implementation works as long as the scope of each transaction (i.e., the amount of data that it reads from or writes to the database) fits in memory. But some applications contain queries that need to access more data than can fit in memory, thus we need to extend our anti-caching model to support them. We first note that we have never seen an OLTP application in which a transaction needs to write a large number of tuples all at once. Thus, there is no need to support writes that exceed the size of main memory. Reads are a different matter. While very uncommon in OLTP, it is possible for an application to need to perform simple analytical queries (e.g., aggregates) on entire tables. This is a problem with our current anti-cache design, since all of data needed to complete such a query must be in memory first before the query can be processed. We now discuss three possible solutions.

Obviously, large read queries generate a concurrency control problem when mixed with transactions that execute write queries. In a traditional DBMS, the query will commence after acquiring a table-level lock, at which point no writes will be processed in parallel.

Once the query finishes, the lock is released and all the queued up writes can move forward. In this scenario, the total time in the DBMS is divided into two modes of operations: (1) small write queries are running or (2) large read queries are running. This could be implemented in H-Store, but we suspect that it will have the same performance as traditional DBMS architectures [29].

The second solution is to process large read queries in historical mode [30]. For this approach, each transaction is assigned a timestamp of when it enters the system and then is only allowed to read tuples with a timestamp that is less than or equal to it. The DBMS will not overwrite tuples when one of these queries is running, but instead must keep both the before and after images of the database to ensure that the large read queries are provided with the correct version. H-Store already does this type of no-overwrite processing through its asynchronous checkpoints [22]. Hence, extending it to include timestamps is straightforward. Furthermore, time-travel reads, originally proposed in Postgres, are already supported in several DBMSs, including Vertica and Oracle. Again, this solution is readily implementable and should perform in a comparable fashion to the same solution in a traditional architecture.

Finally, the third solution that is often proposed for this problem is to allow dirty reads (but not dirty writes) [19]. In this case, all read-write conflicts between queries are ignored. The result of a large read query will include the affects of some updates from parallel transactions, but not necessarily all of them. For this solution, no guarantees can be made about the semantics of the read result. In a partitioned system like H-Store, the query is decomposed into individual, single-partitioned operations and then aggregated together after processing the data at each partition separately. Again, this solution should execute with comparable performance to the same solution in existing systems.

Although we plan to explore these options in detail in the future, we do not expect the results to change the benefits of anti-caching.

## 6.2 Block Reorganization

As described in Section 3.4, when the DBMS uses the tuple-merge retrieval policy, the anti-cache blocks could contain "holes" of tuples that were selectively unevicted. Depending on the workload, over time these holes reduce the number of tuples that are retrieved when a block is retrieved from disk. Thus, we are investigating how to reorganize blocks to reduce the number disk operations without affecting the system's runtime performance.

There are several drawbacks to our lazy compaction scheme described in Section 3.4. First, while the holes accumulate within a block but remain below the threshold that triggers the compaction, every time the block is read the garbage data is retrieved. Ideally, each block fetched from disk would be full. Another problem is that under the lazy block compaction scheme, when the number of holes rises above the acceptable threshold and the entire block is merged into memory, the non-hole tuples being merged in are cold and likely unwanted. Thus, this has the same drawback as the block-merge strategy, though to a lesser degree. It is likely that these tuples will be immediately evicted during the next eviction cycle. But if we know these tuples are cold, a better design would be to never move them back into memory. As future work, we plan to explore a background block compaction process that compacts blocks without un-evicting the tuples. This could be done by merging half-full blocks and updating the appropriate Evicted Table entries for the evicted tuples compacted. Of course, this would have to be done in a transactionally consistent way, ideally without affecting the overall performance of the system.

Also possible in a block reorganization scheme would be semantic reorganization of blocks consisting of tuples from a set of tables. For example, if one of the queries in the workload often reads data from several tables, then the data is considered semantically related. It would be desirable to have all of these tuples reside on the same block, so that if that data is requested from disk only a single block will need to be read.

## 6.3 Query Optimizations

There are several potential optimizations that would allow H-Store to process queries on evicted tuples without needing to retrieve it from disk. For example, the DBMS does not need to retrieve an evicted tuple if an index "covers" a query (i.e., all of the columns that the query needs in its predicate and output are in the index). Another idea to further reduce the size of the database that needs to be kept in memory is to evict only a portion of a tuple to disk. That is, rather than evicting the entire tuple, the system could only evict those columns that are unlikely to be needed. The system could analyze transactions and identify which columns in each table are not accessed often by queries and then choose the optimal design that minimizes the number of block retrievals but also maximizes the memory saved.

# 7. RELATED WORK

There is an extensive history of research on main memory DBMSs. Notable systems include PRISMA/DB [6], Dalí [17] (later renamed to DataBlitz [7]), and TimesTen [31]. Commercial implementations include VoltDB [4], SAP's HANA [13], MemSQL [3], and EXtremeDB [1]. RAMCloud [25] provides a scalable main memory resident key-value store for use in various cloud computing environments, though it does not provide some other common DBMS features, such as secondary indexes and multi-object transactions. All of these systems are limited to databases that are smaller than the amount of available memory.

The HyPer DBMS [19] is a main memory system that is designed to execute both OLTP and OLAP queries simultaneously. Similar to H-Store, OLTP queries are executed serially at partitions without the need for a heavyweight concurrency control scheme. HyPer creates periodic memory snapshots to execute long running OLAP queries. The DBMS relies on virtual memory paging to support databases that are larger than the amount of available memory.

In [28], similar to this work, the authors address the problem of evicting cold data to disk in a main memory database. However, their approach is very different, and relies on virtual memory to swap data from memory to disk. Tuple-level access patterns are analyzed off line, and in-memory data is reorganized according to these access patterns. Cold data is moved to a memory location that is more likely to be paged to disk by the OS. This approach is similar to anti-caching in that it attempts to evict the cold data to disk and maintain the hot working set in memory. The major difference between the two approaches is that the anti-caching architecture does not block while an evicted block is being read from disk. This allows other transactions to execute during the disk read. In contrast, during a virtual memory page fault, no further transactions can be executed.

The goals of Project Siberia, part of Microsoft's Hekaton [12] main memory table extension for SQL Server, are also similar to our anti-caching proposal, but the implementations are different. In Hekaton, a table either exists entirely in main memory or is considered disk-based, meaning it is controlled by the standard disk-based execution engine with buffer pool, locks and latches. Anti-caching, which evicts data at the granularity of individual tuples, offers finer-grained control over which data is evicted.

In [20] the authors propose a method for identifying hot and cold tuples from a sample of transactions in Hekaton. For our implementation in H-Store, we use a LRU-based identification method that does not require an off-line mechanism. We consider this work complementary and plan to investigate more complicated schemes for identifying cold data.

Calvin [33] is a main memory OLTP system that is designed to efficiently handle distributed transactions. It is also able to read disk-resident data in a transactionally consistent way. To do this, Calvin serializes transactions similar to a disk-based system. If a small percentage of transactions need disk-resident data (the paper suggests less than 1%), it is possible to hide disk latency and avoid performance degradation.

The problem of maintaining coherence between an in-memory buffer pool and data store on disk is explored in several previous works by retrofitting DBMSs to work with distributed caches. MemcacheSQL [8] does this by modifying Postgres's buffer pool to use Memcached [14] as an extended distributed memory. All transactions interact only with the Postgres front-end. In [27], the authors propose TxCache, a transactionally consistent DBMS that automatically manages data in a standalone instance of Memcached. The user must still specify how long data will remain in the cache and the application still must perform a separate query each time to determine whether an object is in the cache.

Our pre-pass execution phase is similar to run-ahead execution models for processors explored in [23], where that goal is to pre-execute instructions to identify page faults and pre-fetch data pages.

## 8. CONCLUSION

In this paper, we presented a new architecture for managing datasets that are larger than the available memory while executing OLTP workloads. With anti-caching, memory is the primary storage and cold data is evicted to disk. Cold data is fetched from disk as needed and merged with in-memory data while maintaining transactional consistency. We also presented an analysis of our anti-caching model on two popular OLTP benchmarks, namely YCSB and TPC-C, across a wide range of data sizes and workload parameters. On the workloads and data sizes tested our results are convincing. For skewed workloads with data 8× the size of memory, anti-caching has an 8×-17× performance advantage over a disk-based DBMS and a 2×-9× performance advantage over the same disk-based system fronted with a distributed main memory cache. We conclude that for OLTP workloads, in particular those with skewed data access, the results of this study demonstrate that anti-caching can outperform traditional architectures popular today.

## 9. ACKNOWLEDGEMENTS

This work was funded in part by the Intel Big Data ISTC. In addition, the authors would like to thank Donghui Zhang and VoltDB for their input in the early stages of this project.

## 10. REFERENCES

[1] eXtremeDB. http://www.mcobject.com.

[2] H-Store. http://hstore.cs.brown.edu.

[3] MemSQL. http://www.memsql.com.

[4] VoltDB. http://www.voltdb.com.

[5] Oracle TimesTen Products and Technologies. Technical report, February 2007.

[6] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, and A. N. Wilschut. PRISMA/DB: A parallel, main memory relational DBMS. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):541–554, 1992.

[7] J. Baulier, P. Bohannon, S. Gogate, S. Joshi, C. Gupta, A. Khivesera, H. F. Korth, P. McIlroy, J. Miller, P. P. S. Narayan, M. Nemeth, R. Rastogi, A. Silberschatz, and S. Sudarshan. Datablitz: A high performance main-memory storage manager. VLDB, pages 701–, 1998.

[8] Q. Chen, M. Hsu, and R. Wu. MemcacheSQL a scale-out sql cache engine. In *Enabling Real-Time Business Intelligence*, volume 126 of *Lecture Notes in Business Information Processing*, pages 23–37. 2012.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[10] C. A. Curino, D. E. Difallah, A. Pavlo, and P. Cudre-Mauroux. Benchmarking OLTP/Web Databases in the Cloud: The OLTP-Bench Framework. CloudDB, pages 17–20, October 2012.

[11] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[12] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, pages 1–12, 2013.

[13] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.

[14] B. Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004(124):5–, Aug. 2004.

[15] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[16] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter. Group commit timers and high volume transaction systems. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 301–329, 1989.

[17] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dalí: A high performance main memory storage manager. In *VLDB*, pages 48–59, 1994.

[18] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[19] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. ICDE, pages 195–206, 2011.

[20] J. J. Levandoski, P.-A. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, 2013.

[21] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. *DPDS*, pages 177–187, 1988.

[22] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Recovery algorithms for in-memory OLTP databases. *In Submission*, 2013.

[23] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. ISCA, pages 370–381, 2005.

[24] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. USENIX ATEC, 1999.

[25] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, Jan. 2010.

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

[27] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. OSDI'10, pages 1–15, 2010.

[28] R. Stoica and A. Ailamaki. Enabling efficient os paging for main-memory oltp databases. In *DaMon*, 2013.

[29] 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*, pages 1150–1160, 2007.

[30] M. Stonebraker and L. A. Rowe. The design of POSTGRES. SIGMOD, pages 340–355, 1986.

[31] T. Team. In-memory data management for consumer transactions the timesten approach. SIGMOD '99, pages 528–529, 1999.

[32] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, June 2007.

[33] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[34] A. Whitney, D. Shasha, and S. Apter. High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++. In *HPTS*, 1997.