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

Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, Stan Zdonik

Presented by Christian Valdemar Mathiesen

cmath@cs.brown.edu

February 23, 2015

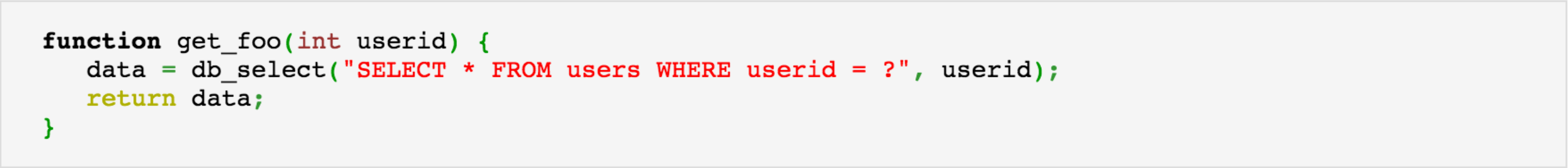
Agenda

- Introduction
- Anti-caching system model
- Benchmarks
- Conclusion

Why is anti-caching necessary?

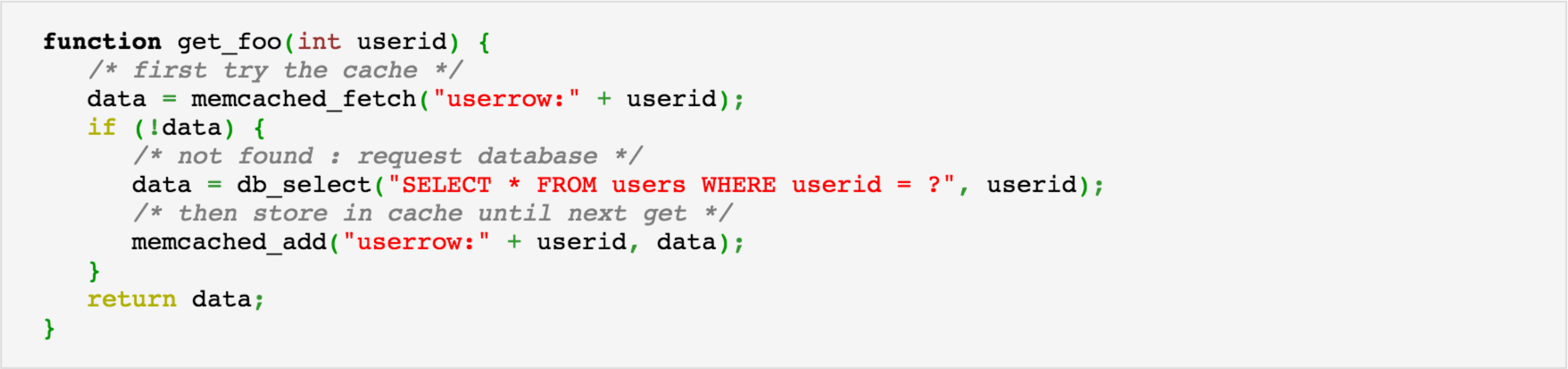
MySQL + memcached

With an IMDB, **fetching** could simply be



```
function get_foo(int userid) {
   data = db_select("SELECT * FROM users WHERE userid = ?", userid);
   return data;
}
```

But now has to be

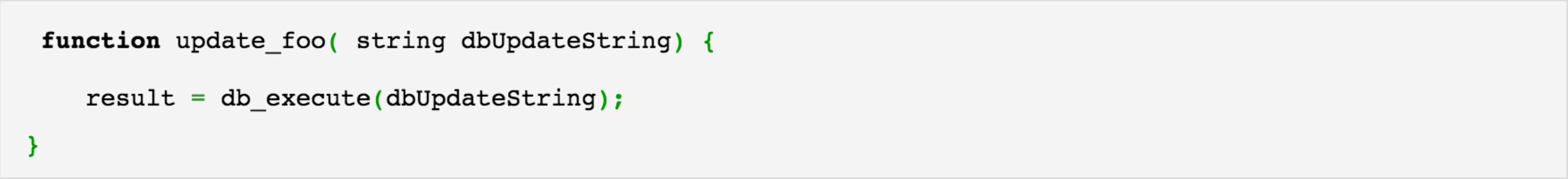


```
function get_foo(int userid) {
   /* first try the cache */
   data = memcached_fetch("userrow:" + userid);
   if (!data) {
      /* not found : request database */
      data = db_select("SELECT * FROM users WHERE userid = ?", userid);
      /* then store in cache until next get */
      memcached_add("userrow:" + userid, data);
   }
   return data;
}
```

Pseudo code courtesy of Wikipedia: http://en.wikipedia.org/wiki/Memcached

MySQL + memcached

With an IMDB, **updating** could simply be

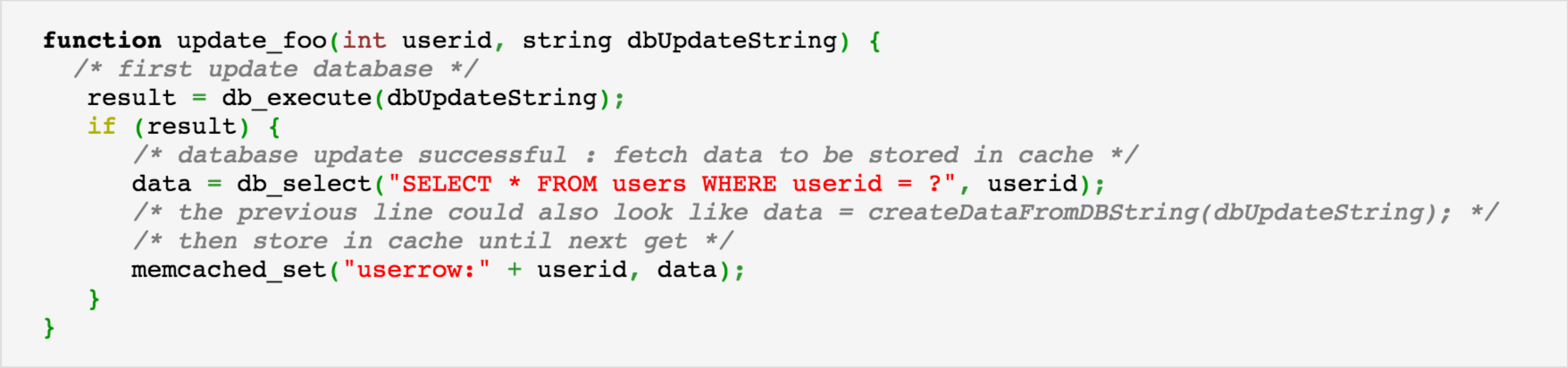


```
function update_foo( string dbUpdateString) {

   result = db_execute(dbUpdateString);

}
```

But now has to be



```
function update_foo(int userid, string dbUpdateString) {
  /* first update database */
   result = db_execute(dbUpdateString);
   if (result) {
      /* database update successful : fetch data to be stored in cache */
      data = db_select("SELECT * FROM users WHERE userid = ?", userid);
      /* the previous line could also look like data = createDataFromDBString(dbUpdateString); */
      /* then store in cache until next get */
      memcached_set("userrow:" + userid, data);
   }
}
```

Pseudo code courtesy of Wikipedia: http://en.wikipedia.org/wiki/Memcached

This shouldn’t be a job for the developer.

Mission:

Keep **hot** data in main memory

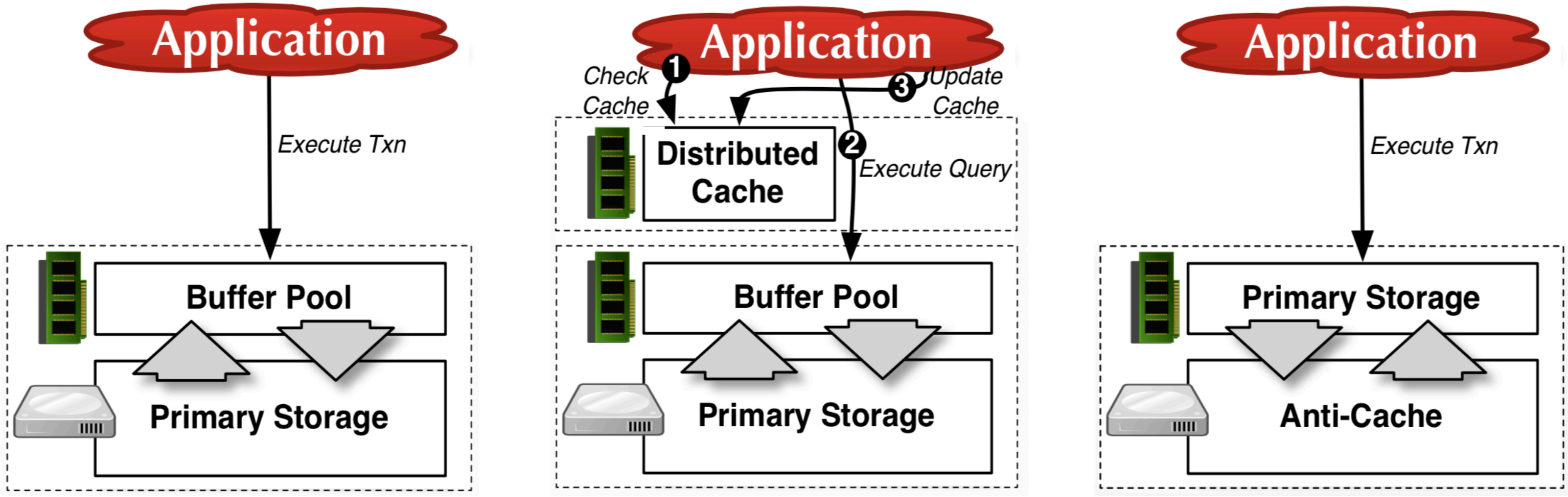
Store **cold** data on disk

Eliminate manual labor

The answer:

An IMDB with **anti-caching**

High-level architectural differences



(a) Disk-oriented DBMS

(b) Disk-oriented DBMS with a Distributed Cache

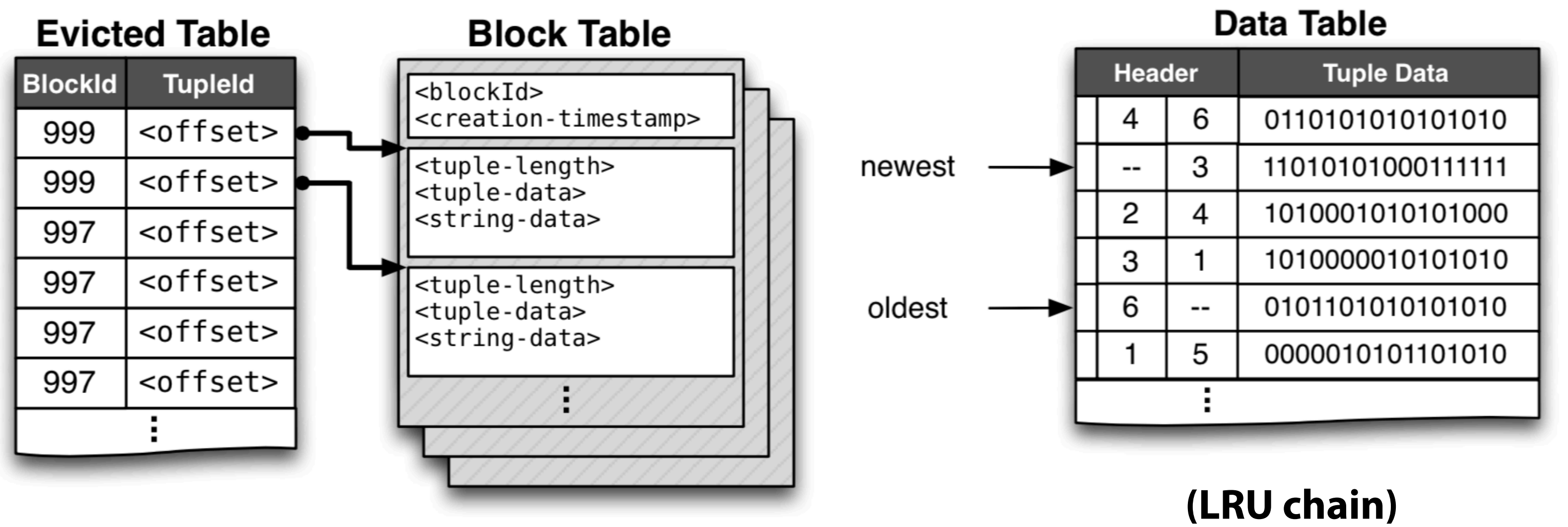
(c) Main Memory DBMS with Anti-Caching

Anti-caching system model

- Extends **H-store**, an IMDB
- Uses an LRU scheme to evict cold data from Main Memory
- Transactions involving tuples stored on disk are stalled and restarted after involved data is copied back to MM

Anti-caching storage model

3 components:



Evicted Table

BlockId	TupleId
999	<offset>
999	<offset>
997	<offset>
997	<offset>
997	<offset>
997	<offset>
⋮	

Block Table

```
<blockId>
<creation-timestamp>
```

```
<tuple-length>
<tuple-data>
<string-data>
```

```
<tuple-length>
<tuple-data>
<string-data>
```

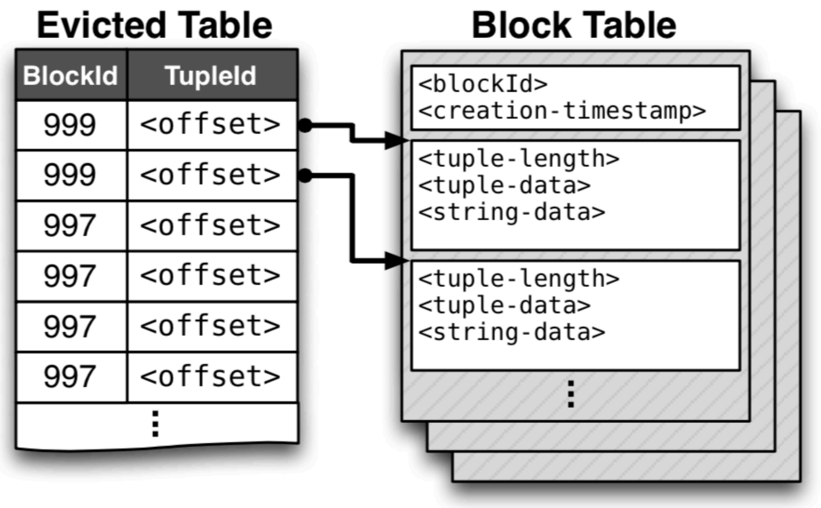
⋮

Data Table

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

(LRU chain)

Anti-caching storage model



Evicted Table

BlockId	TupleId
999	<offset>
999	<offset>
997	<offset>
997	<offset>
997	<offset>
997	<offset>
⋮	

Block Table

```
<blockId>
<creation-timestamp>
```

```
<tuple-length>
<tuple-data>
<string-data>
```

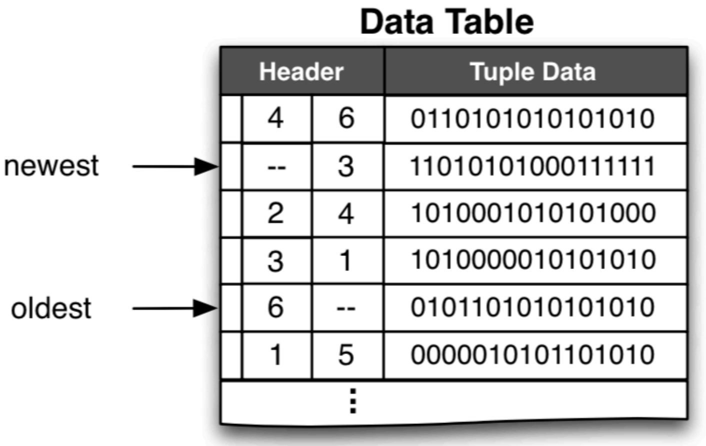
```
<tuple-length>
<tuple-data>
<string-data>
```

⋮

- Block table resides on disk and contains evicted tuple data
- Evicted table resides in memory and maps evicted tuples to Block table

Anti-caching storage model

- Doubly-Linked List data structure in tuple header
- To avoid CPU overhead, only a certain fraction (α) of transactions are used to update the LRU chain
- Data tables are marked as *evictable/non-evictable*

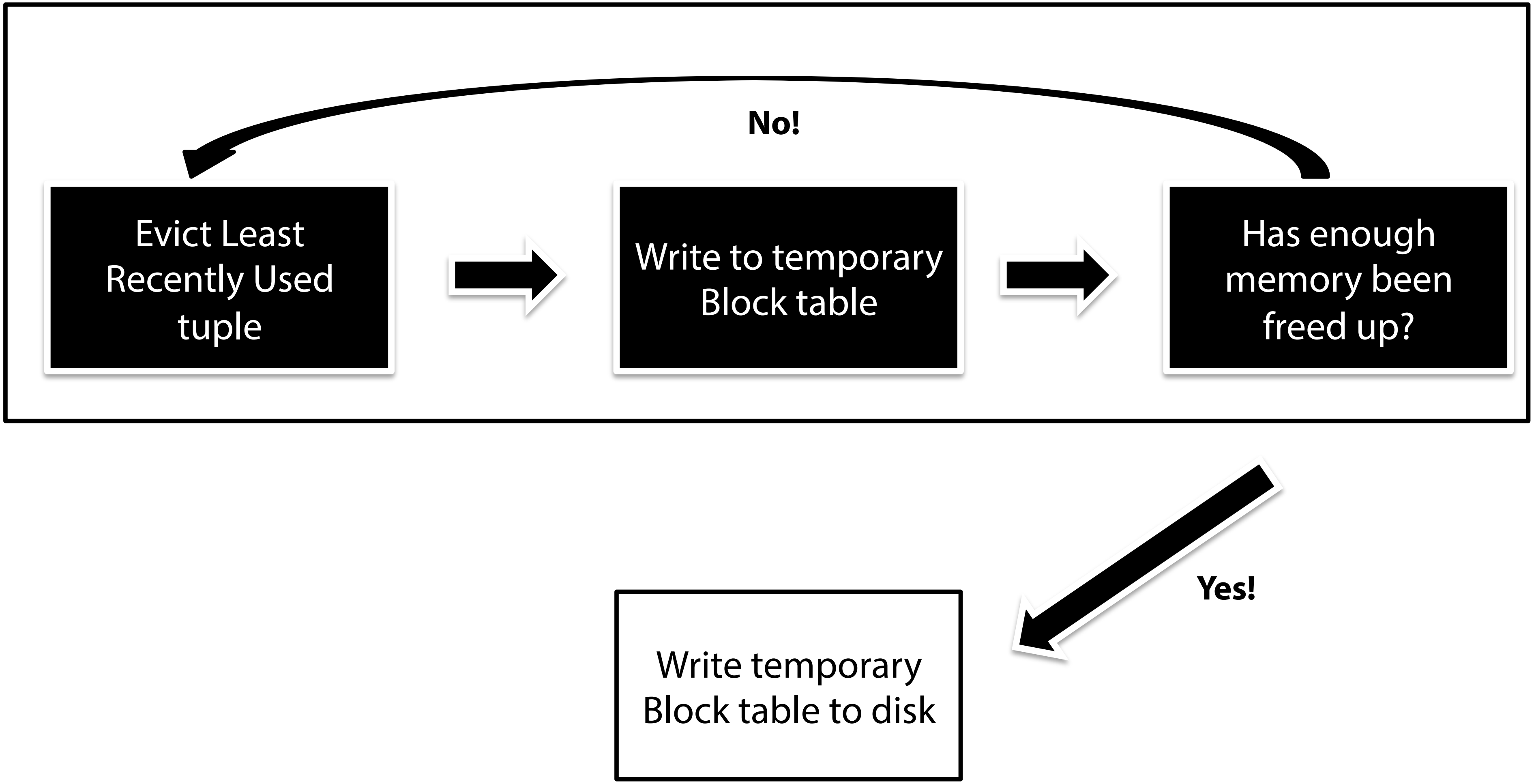


Data Table

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

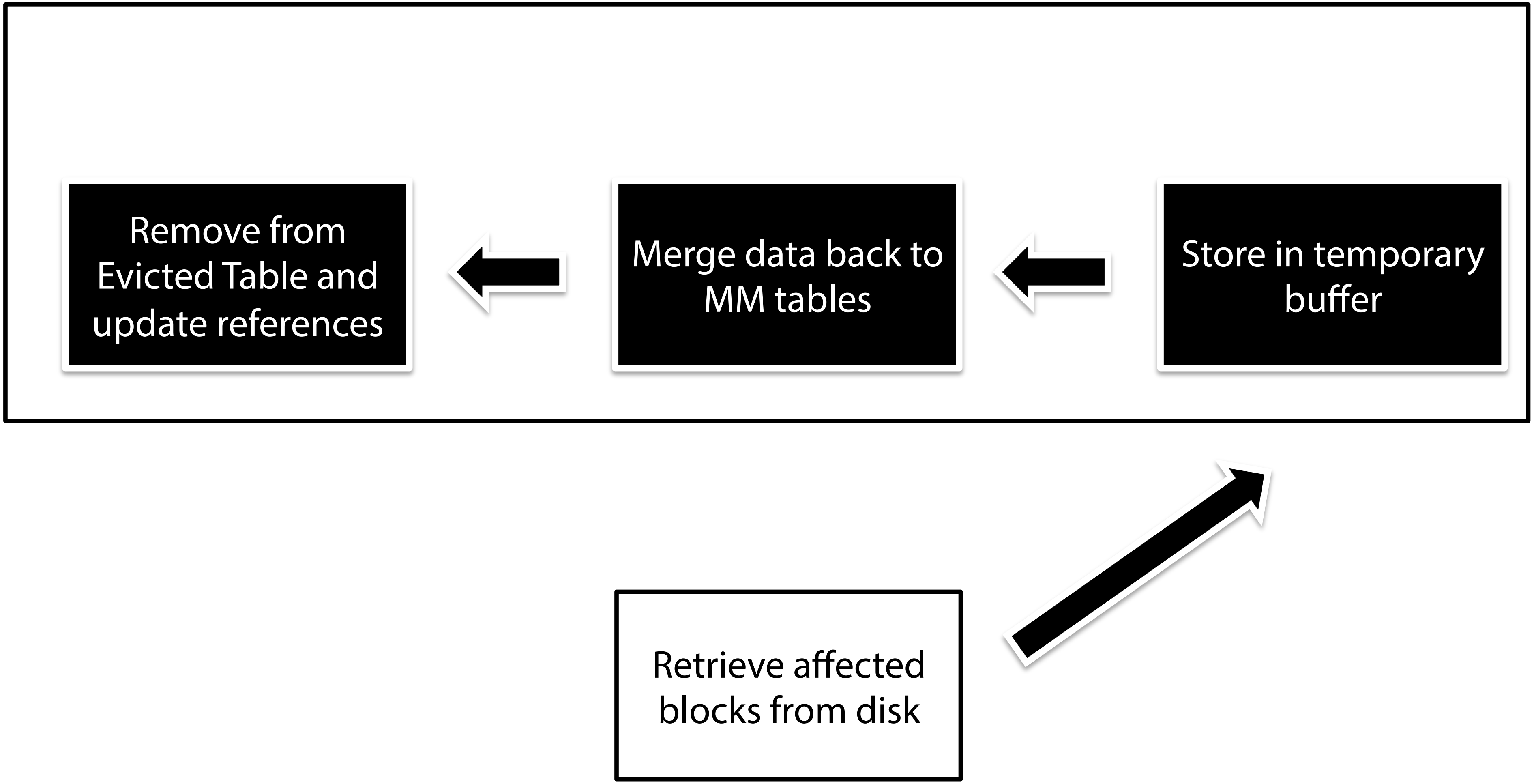
(LRU chain)

How is data evicted?



No!
Evict Least Recently Used tuple
Write to temporary Block table
Has enough memory been freed up?
Yes!
Write temporary Block table to disk

How is data retrieved?



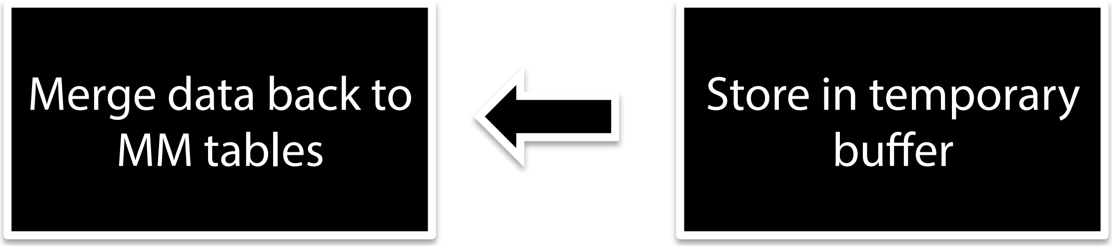
Remove from Evicted Table and update references

Merge data back to MM tables

Store in temporary buffer

Retrieve affected blocks from disk

How is data retrieved?



Merge data back to MM tables

Store in temporary buffer

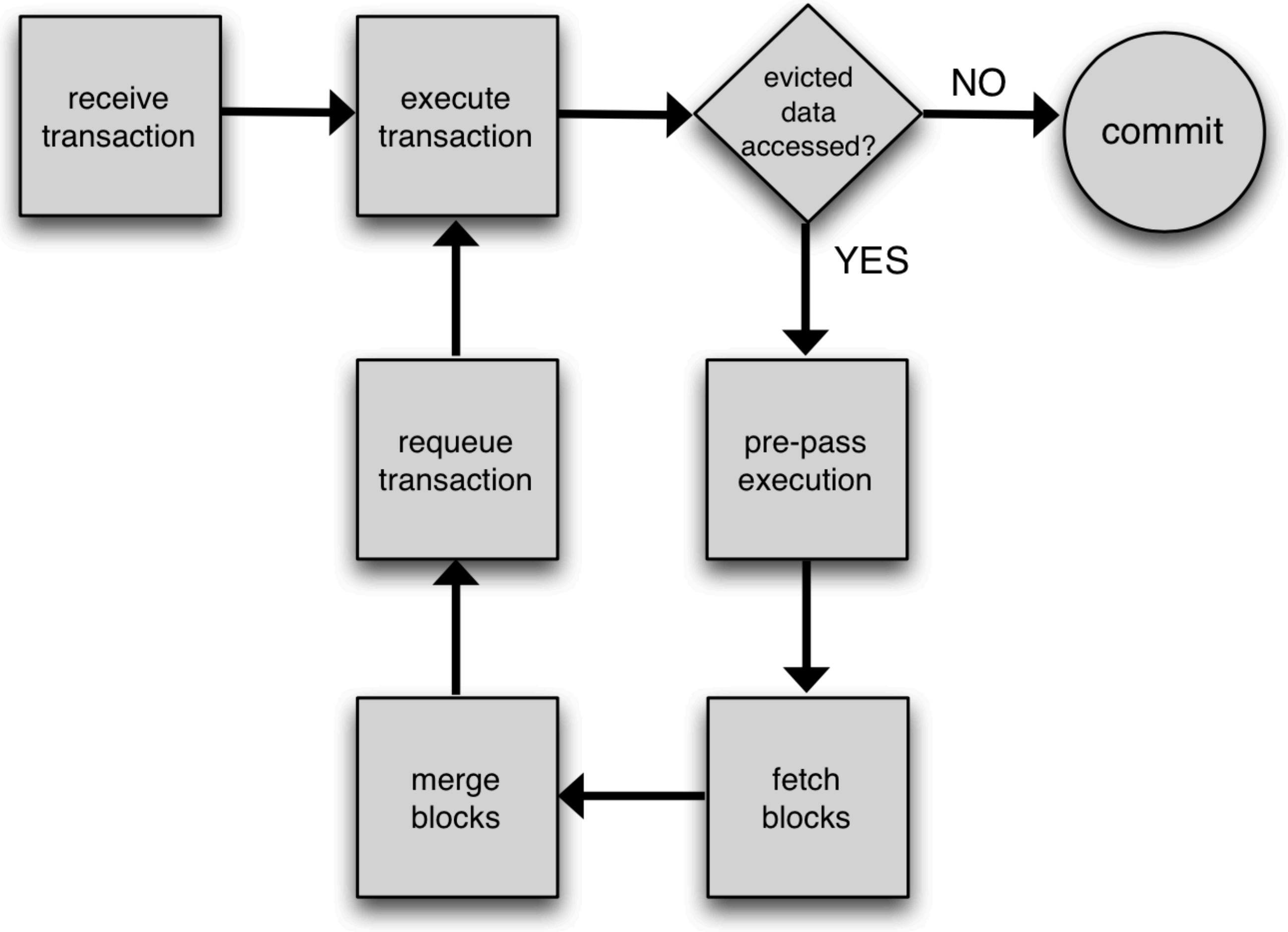
2 ways of doing this:

Load **entire** Block back

- or -

Only load back **affected tuples** from Block

Finally, the over-all **transaction execution diagram**



So how well does it work?

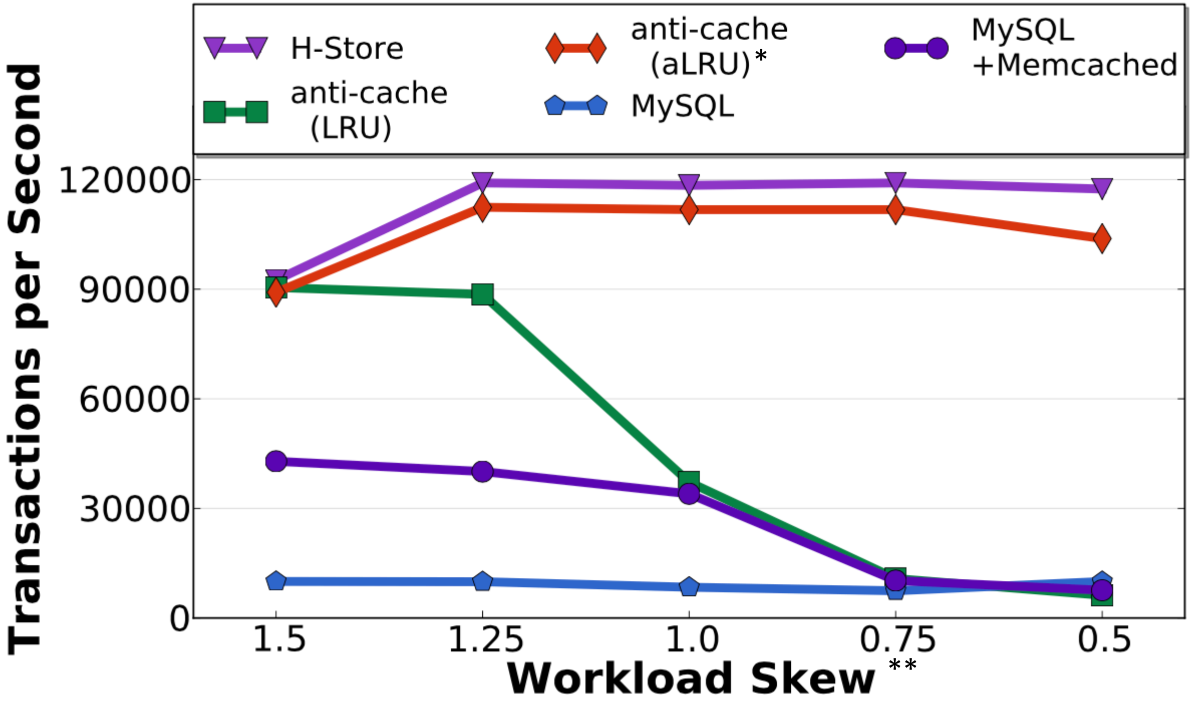
Datasets

- **Yahoo! Cloud Serving Benchmark** (YCSB)
  - Simulates data from large-scale services created by Internet-based companies
  - 20 GB single table, 7 columns of random string data for each tuple
- **TPC-C**
  - Current industry standard for evaluating OLTP systems
  - 10GB containing 100 warehouses and 100,000 items

System Configurations

- **MySQL**
  - The world's most popular disk-based DBMS
- **MySQL + memcached**
  - Popular tool to speed up MySQL's performance by storing hot data in MM
- **H-Store with Anti-caching**
  - No further introduction necessary

YCSB experiments

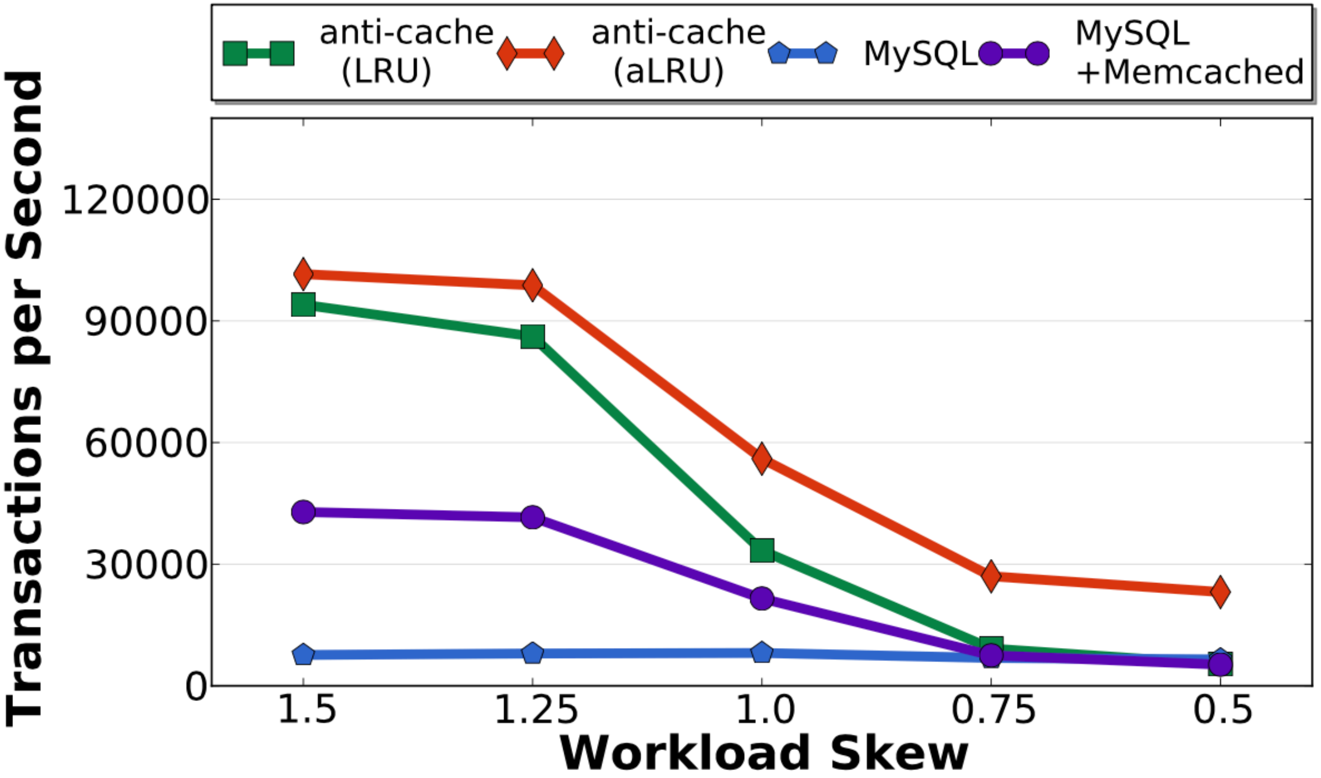


(a) $\frac{\textbf{data\_size}}{\textbf{mem\_size}} = 1$, **read-only**

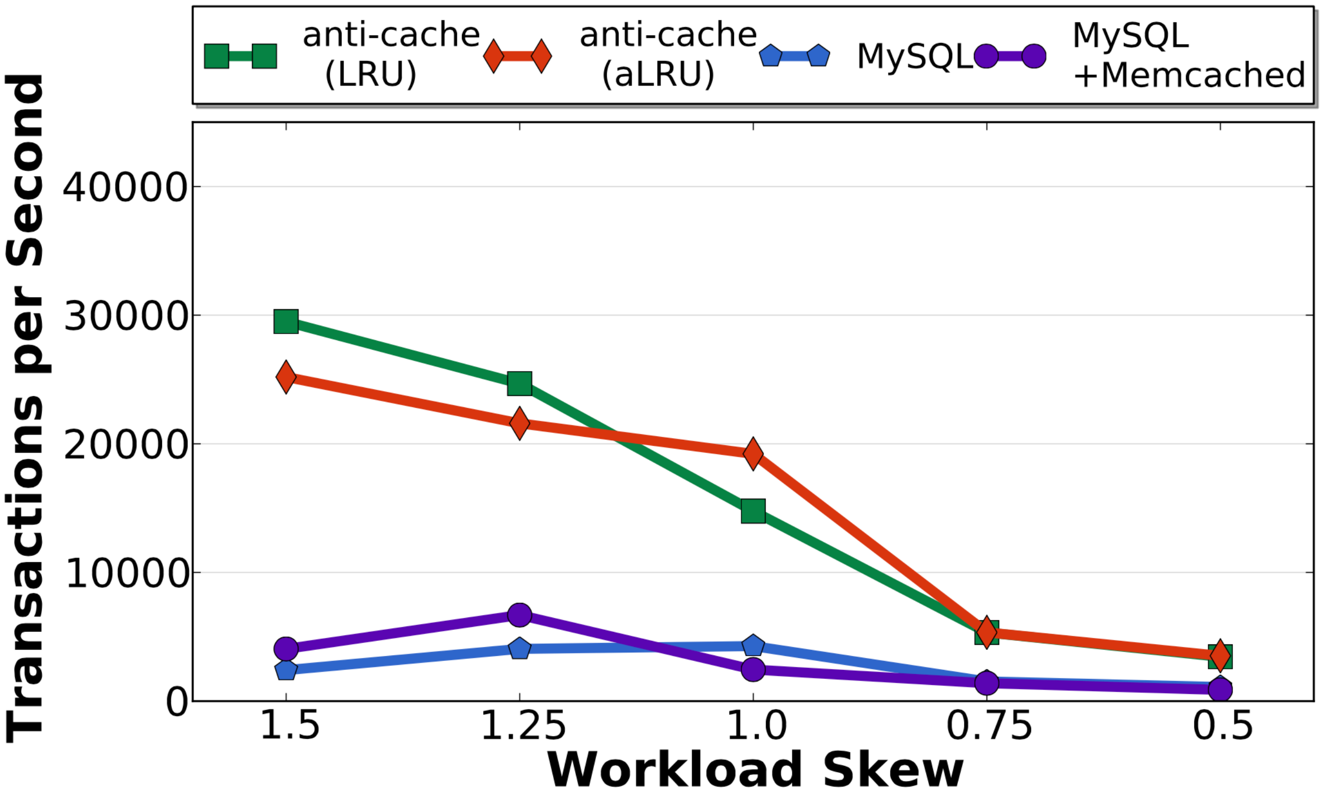
* aLRU represents a low LRU transaction sample rate (alpha = 0.01)

** Higher values of workload skew means that older items are accessed much less frequently than newer items

Look what happens, when we have twice as much data as we have memory..

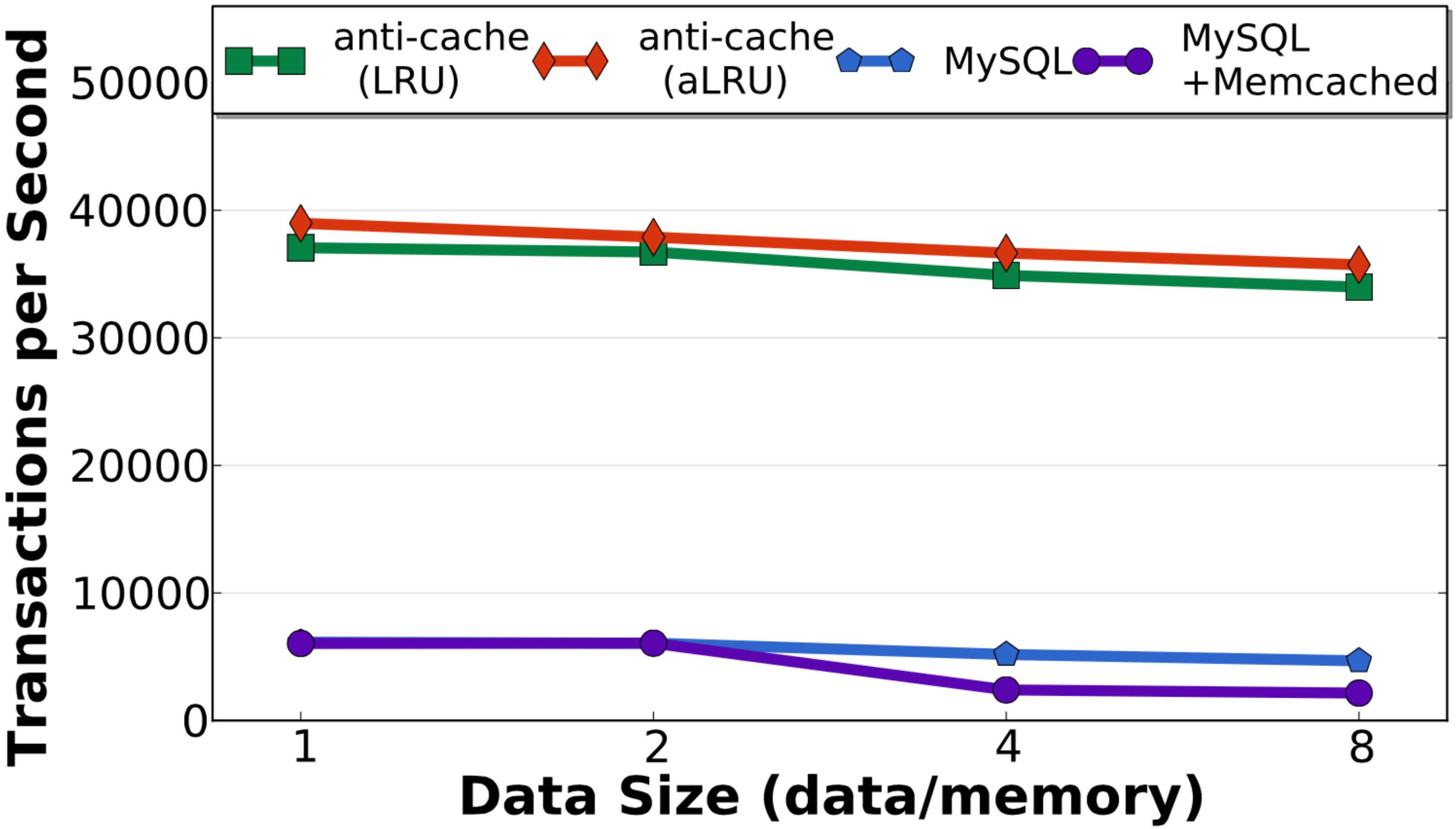


(b) $\frac{\textbf{data\_size}}{\textbf{mem\_size}} = 2$, **read-only**



(l) $\frac{\textbf{data\_size}}{\textbf{mem\_size}} = 8$, **write-heavy**

TPC-C experiments



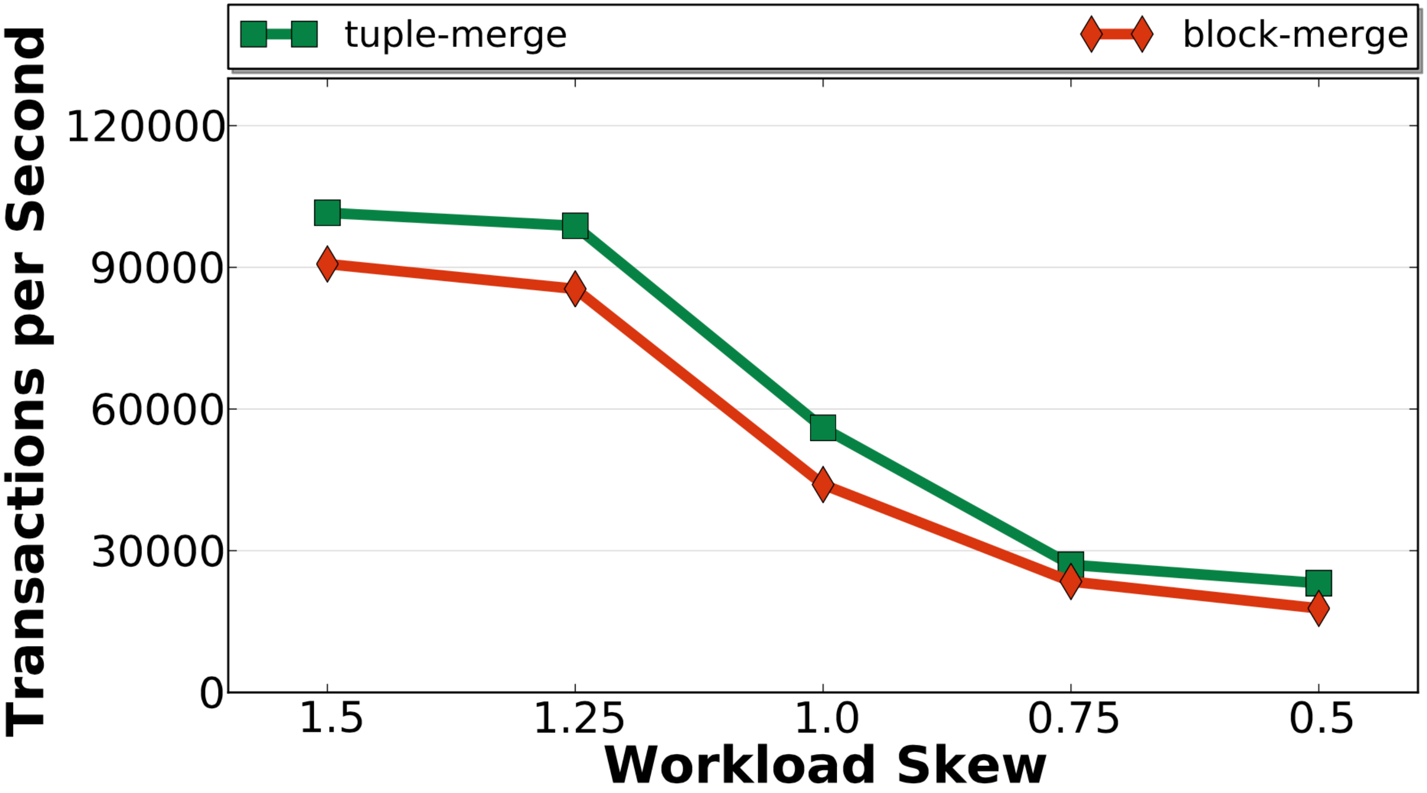


Figure 8: Merge Strategy Analysis – YCSB read-only, 2× memory, 1MB evict blocks.

Conclusion

- Anti-caching offers an excellent and clean solution to one of the shortcomings of IMDBs
- From the benchmark experiments conducted, H-Store with Anti-caching performs significantly better than MySQL (8x-17x performance advantage)
- This is true - even with memcached enabled (2x-9x performance advantage)