KVell+: Snapshot Isolation without Snapshots

Baptiste Lepers and Oana Balmau, *University of Sydney;*
Karan Gupta, *Nutanix Inc.;* Willy Zwaenepoel, *University of Sydney*

https://www.usenix.org/conference/osdi20/presentation/lepers

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

KVell+: Snapshot Isolation without Snapshots

Baptiste Lepers
University of Sydney

Oana Balmau
University of Sydney

Karan Gupta
Nutanix

Willy Zwaenepoel
University of Sydney

Abstract

Snapshot Isolation (SI) enables online analytical processing (OLAP) queries to observe a snapshot of the data at the time the query is issued, despite concurrent updates by online transactional processing (OLTP) transactions. The conventional implementation of SI creates a new version of a data item when it is updated, rather than overwriting the old version. Versions are garbage collected when they can no longer be read by any OLAP query. Frequent updates during long-running OLAP queries therefore create significant space amplification, and garbage collection can give rise to latency spikes for OLTP transactions. These problems are exacerbated on modern low-latency drives that can persist millions of updates per second.

We observe that analytic queries often consist in large part of commutative processing of data items resulting from range scans in which each item in the range is read exactly once. We introduce Online Commutative Processing (OLCP), a new model for processing analytical queries, that takes advantage of this observation. Under OLCP, analytical queries observe the same snapshot of the data as they would under conventional SI, but space amplification and garbage collection costs are largely and oftentimes nearly entirely avoided. When an item in such a range is updated, the old version of the item is propagated to the OLCP queries that might need it instead of being kept in the store.

We demonstrate OLCP's expressiveness by showing how to formulate, among others, the TPC-H benchmark queries in OLCP. We implement OLCP in KVell+, an extension of KVell, a key-value store for NVMe SSDs. Using YCSB-T, TPC-CH and production workloads from Nutanix, we run a wide range of analytics queries concurrently with write-intensive transactions. We show that OLCP incurs little or no space amplification or garbage collection overhead. As a surprising by-product we also show that OLCP speeds up analytical queries compared to SI.

1 Introduction

The desire to run frequent analytics on fresh data has led to the recent development of databases that allow concurrent processing of online transaction processing (OLTP) and online analytical processing (OLAP) [34]. To isolate OLAP queries from OLTP updates, databases typically rely on Snapshot Isolation (SI) [51, 66, 69]. SI provides OLAP queries with a snapshot of the database at the time the query is issued, independent of later updates made by OLTP transactions. Conventionally, SI is implemented by multi-versioning [6]: an update generates a new version of a data item, and previous versions are kept for as long as they belong to an active OLAP query's snapshot. Versions that no longer belong to such a snapshot are garbage collected. Long queries may thus cause the store to grow—a phenomenon known as *space amplification*, and garbage collection may provoke latency spikes.

Minimizing disk usage is important in production systems. Facebook found that "storage space is the bottleneck" [23], and Alibaba Group runs garbage collection with "the highest priority to prevent waste of storage space" [32]. Space amplification is particularly problematic when the dataset is stored on modern storage devices. NVMe SSDs can persist millions of items per second. Furthermore, because random and sequential access bandwidth are nearly identical, scanning data is no longer faster than performing random access updates. An analytical query running concurrently with write-intensive transactions may therefore cause the size of the store to increase manyfold.

Space amplification is a well-known problem for inmemory SI data stores. Various solutions have been developed, but they perform poorly on disk-based systems. Executing transactions sequentially avoids the need for locking and versioning [28], but is impractical when I/O latencies have to be overlapped with CPU use. Creating snapshots using operating system fork and copy-on-write techniques [39] incurs very high file system overheads when applied to disk-based systems. Closest to our work, Steam [8] trims versions that do not belong to any *active* snapshot, providing efficiencies for some workloads. We propose a more radical re-design, suitable also for disk storage, that seeks to altogether avoid keeping old versions in the store.

Our approach is based on the following two observations. First, most OLAP queries scan data, but are oblivious to the order in which they read items, because the operations performed on items are commutative. Second, OLAP queries read scanned items at most once. For instance, queries that compute sales statistics (e.g., the most popular item in a region) can perform their operation by scanning items once in any order. Based on these observations, we define a new class of processing: OnLine Commutative Processing (OLCP). OLCP queries declare so called *scan ranges*. When an item in a scan range is updated by a concurrent transaction, its old value is processed by OLCP queries and then discarded, instead of being kept in the store.

Scan ranges have numerous advantages. First, because no versions are kept for items in scan ranges, space amplification is limited, and GC overhead is reduced. Second, because OLCP queries process items in scan ranges as they are modified, they read more data from memory and less from disk, and thus have higher throughput than their OLAP counterparts. The trade-off is that an OLCP query can read items belonging to its scan ranges only once. In addition, scanned items are not guaranteed to be read in order.

In addition, OLCP queries can declare *point ranges*, ranges of items on which they want to perform point queries. Items in point ranges are versioned, as in the conventional SI implementation. The combination of scan ranges and point ranges allows OLAP queries to be expressed efficiently in OLCP. Moreover, a very large subset can be expressed in a manner so that they derive great benefit from OLCP, including reduced space amplification, no GC-induced latency spikes, and higher throughput.

We implement OLCP queries in Kvell+, an extension of KVell [45]. In KVell+, scan and point ranges are declared through an interface inspired by the MapReduce paradigm [19]. OLCP queries declare a `map` function that is called *exactly once* on all items that belong to scan ranges. The items that `map` reads correspond to the items that the query would have read under conventional SI (i.e., belonging to the snapshot at the start of the query). The `map` function can also perform point queries on items in point ranges. In the absence of updates by OLTP transactions, `map` is called on items in scan ranges in lexicographic order, but when an item is updated, we *propagate* its old value to OLCP queries. The old value is processed by the OLCP queries (potentially breaking the lexicographic order of the scans) and then deleted from the store. Space freed by the deletion can be reused to store new items.

OLCP can easily be integrated in existing applications, either manually, using an SQL-to-MapReduce tool [44,71], or automatically at the SQL query-plan level. OLAP and OLCP queries can run simultaneously on the same data. A developer may therefore choose to port existing OLAP queries that create substantial space amplification to OLCP, while leaving less problematic OLAP queries to run under SI.

We make the following contributions:

- The OLCP query model.
- A detailed explanation and examples showing how to port OLAP queries (e.g., MapReduce analytics, TPC-H queries) to OLCP.
- The implementation of OLCP in KVell+.
- A comparison of OLCP to SI and Steam [8] in terms of space amplification, tail latency and throughput.

Roadmap. Section 2 presents the key OLCP principles. Section 3 explains in detail how data analytics workloads can greatly benefit from OLCP. Section 4 discusses the implementation. Section 5 shows our experimental evaluation results. Section 6 presents the related work, and Section 7 concludes.

2 OLCP Overview

In this section, we explain OLCP's design principles, advantages, and limitations. We explain how to perform scans concurrently with propagation events.

2.1 OLCP in a nutshell

The main goal of OLCP is to reduce the time that old item versions spend in the store. OLCP allows the store to *reduce the lifespan of old versions down to the duration of OLTP commits*. In a conventional SI implementation, OLAP queries force the store to keep old versions for the entire duration of the queries. In contrast, OLCP queries process old versions as they are generated. Once the old version of an item has been processed, it is deleted from the store and its space can be reused to store new items.

OLCP advantages: OLCP provides the same guarantees as SI, with virtually no space amplification. Furthermore, because OLCP queries process items as they are being updated, OLCP queries avoid reads to disk, improving the throughput of analytical queries.

OLCP requirements: To completely avoid space amplification under OLCP, queries need to support scanning out-of-order and to access each item in the scan ranges only once. These requirements can be relaxed at the expense of increased space amplification, but OLCP's space amplification is always lower than that of conventional SI implementations. OLCP queries are widely applicable and constitute an efficient replacement of OLAP queries, as we demonstrate in Section 3.

2.2 OLCP interface

MapReduce interface: The interface of OLCP is inspired by the event-driven MapReduce paradigm [19]. An OLCP query is created and executed by a single function call:

t = olcp_query(map, payload, [$scan\_range_1$,
$scan\_range_2$, ...], [$point\_range_1$, ...])

The olcp_query call takes the following parameters:

- **A map function callback.** OLCP guarantees that the map callback is called exactly once on all items within the scan ranges. The exactly-once guarantee is essential to limit overheads. Without it, OLCP would have to maintain a list of items they have already seen, which could have prohibitive CPU and memory overhead for large scans. The item versions provided to the map callback correspond to those that the query would have scanned under SI (i.e., those belonging to the snapshot at the time the query is launched).
- **Payload for the map callback.** An arbitrary pointer to application specific data. Usually used to retrieve or store intermediary computation results.
- **Scan ranges.** This range can be the entire store. If ranges overlap, the map function is called only once per item belonging to the ranges. Items belonging to the ranges are not versioned and induce no space amplification. In return, items belonging to the ranges are not guaranteed to be scanned in order (old versions might be scanned before their turn to avoid keeping them in snapshots).
- **Point ranges.** OLCP queries may also declare ranges of items that they might access using point queries. Items within those ranges are versioned until the query is committed and may induce space amplification. Items in the point ranges can be accessed multiple times, and scans on these ranges are guaranteed to happen in lexicographic order. Many analytical processing queries can be expressed without using point queries, as we show in Section 3.

Items outside of the scan and point ranges are neither versioned nor propagated. The olcp_query function blocks until the scans are complete. After calling olcp_query, a developer might choose to do further processing on the payload. In the remainder of this paper, we do this processing in a reduce function.

2.3 Scans, propagation and space reclamation

Algorithm 1 presents pseudo code for scanning, updating and propagating updates in a store that supports OLCP queries. For simplicity, we present a sequential implementation that does not support point ranges. A full implementation would have to handle possible races between scans and propagations and delay the deletion of items belonging to point ranges. We also assume the use of timestamps to define snapshots, as is common in SI implementations.

Algorithm 1 Pseudo-code of a sequential implementation of updates, propagations, and scans.

```
1   /*OLCP commit: create a new version and add the
2   old version in GC queue */
3   timestamp t_commit = now();
4   active_commit_timestamps.add(t_commit);
5   foreach(item i in updated_items) {
6       kv.write(i, t_commit);
7       gc.add(get_oldest(i), t_commit);
8   }
9   active_commit_timestamps.delete(t_commit);

11  /* GC */
12  timestamp t_min=min(active_commit_timestamps);
13  foreach(item i in gc) {
14      // Only delete items from
15      // fully committed transactions
16      if(i.t_commit >= t_min)
17         break;
18      foreach(olcp o in running_olcp) {
19        if(o.in_snapshot(i)
20         && i.key > o.last_scanned)
21          o.propagation_queue.add(i);
22      }
23      delete(i); // remove from the store
24  }

26  /* OLCP query thread */
27  item last_scanned = get_first(scan_range);
28  do {
29    if(last_scanned != EOF) {
30      map(last_scanned, payload);
31      get_next(&last_scanned);
32    }
33
34    while(item i = propagation_queue.pop())
35      if(i.key > last_scanned)
36          map(i, payload);
37  } while(last_scanned != EOF);
```

Scans: OLCP queries request items from the store in lexicographic order using the get_next function (line 31 of Algorithm 1). When there are no concurrent OLTP transactions, the scan happens as it would in a conventional SI implementation: items are read in lexicographic order, and the map function is called on each of them. In OLCP, however, this order can be "interrupted" by propagations resulting from updates by OLTP transactions to items in the scan ranges. When receiving a propagated item, the OLCP query checks that it has not yet scanned the item and, if so, calls map on it (lines 35-36). Afterwards, the OLCP query resumes the scan from the last scanned item using the get_next function.

Propagation and space reclamation: The key to avoiding space amplification with OLCP queries is to delete old data as soon as possible. However, an old version of an item cannot be deleted as soon as a new version is created. When an OLTP transaction updates multiple items, old items can be deleted only after *all* new items are persisted (to allow recovery in case of a mid-commit crash). Hence, the deletions must happen *after* a transaction has committed. Consequently, the store must maintain multiple versions of committed items for the duration of an OLTP commit.

Committing an OLTP transaction then consists of updating the modified items in the store and enqueueing the oldest version of those items on the GC queue (lines 6-7 of Algorithm 1). After the commit is completed, the GC propagates and deletes items. An item is propagated to an OLCP query only if it belongs to the query's snapshot and if it has not yet been scanned. We rely on the lexicographic order of the scan to efficiently ensure this latter property (line 20). In our pseudo code, we choose to enqueue propagated elements in a per-OLCP query queue (line 21), but an implementation might choose a different communication mechanism between the store and running OLCP queries.

Space reclamation efficiency: In practice, the number of versioned items in OLCP is small. When an OLCP query does not use point ranges, a rough estimate of the number of versioned items in OLCP is the number of updates per transaction times the number of concurrent commits. In a conventional implementation of SI, this number is much higher, since the system needs to keep old versions of all items updated during the lifetime of OLAP queries.

Old versions are also only kept for a much shorter time in OLCP. Figure 1 summarizes the lifespan of objects, executing an OLTP transaction concurrently with an (a) OLAP or (b) OLCP query. With OLAP queries, the store has to keep all versions of items for the duration of long queries (minutes), while OLCP allows the store to remove old versions after at most a few commits (microseconds).

OLCP queries can run alongside OLAP queries. In that case the deletion and propagation of items is postponed to ensure the correct execution of OLAP queries: items are deleted and propagated when they no longer belong to an OLAP snapshot. OLAP queries may thus reduce the effectiveness of using OLCP.

2.4 Informal correctness argument

Correctness requires that, despite concurrent OLTP transactions, OLCP queries read the same items from their scan ranges as they would have read under a conventional implementation of SI, and that these items are processed *exactly once*. The correctness relies on the following observations.

An item is propagated at most once, and the propagated item belongs to the query's snapshot. If an item is not updated, then no propagation occurs. If an item is updated once, its old version is propagated only if it belongs to the snapshot (line 19 in Algorithm 1). If an item is updated multiple times, all old versions are put in the GC queue, but only one of them belongs to the snapshot and is propagated.

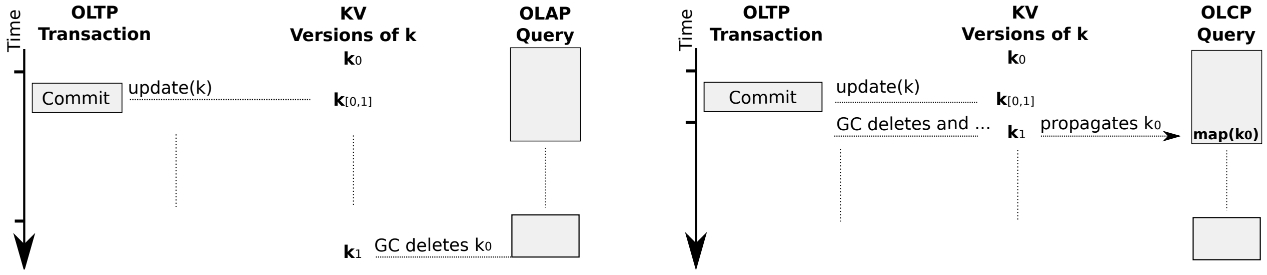
An item is processed exactly once. If an item is not propagated, it is read as part of the scan. The scan does not "skip" items: after scanning an item, a query always requests the next item from the store regardless of concurrent propagations. Thus, a query always scans its entire *scan_ranges*. Only items that have not yet been scanned are propagated (lines 20 and 35 in Algorithm 1).

From the previous observations, we conclude that an OLCP query processes all the items belonging to its scan ranges exactly once, and that the processed items belong to its snapshot. As a result, a developer need not consider the distinction between scanned and propagated items.

2.5 Example

Figure 2 illustrates with an example some of the complex interleavings between OLTP and OLCP. An OLCP query T scans a range of 5 items. T has snapshot timestamp 0. The initial versions of all five items have timestamp 0, and therefore belong to T's snapshot. Despite various updates by OLTP transactions, T correctly calls `map` exactly once on all five initial item versions.

Of particular interest in this execution is item d that is updated twice, at t_2 and t_4, but only d_0 is propagated. Despite being interrupted by the propagation of d_0, T correctly



(a)

(b)

Figure 1: Lifespan of items under OLAP (a) and OLCP (b). OLAP forces an old item version to be kept for the entire duration of the OLAP query, while OLCP needs to keep it only for the duration of the commit of the OLTP transaction that produces the new version.

resumes its scan from b_0 at time $t3$. Finally, at t_5, a is not propagated because the query already scanned b, and at t_6, c is not propagated because it has just been scanned.

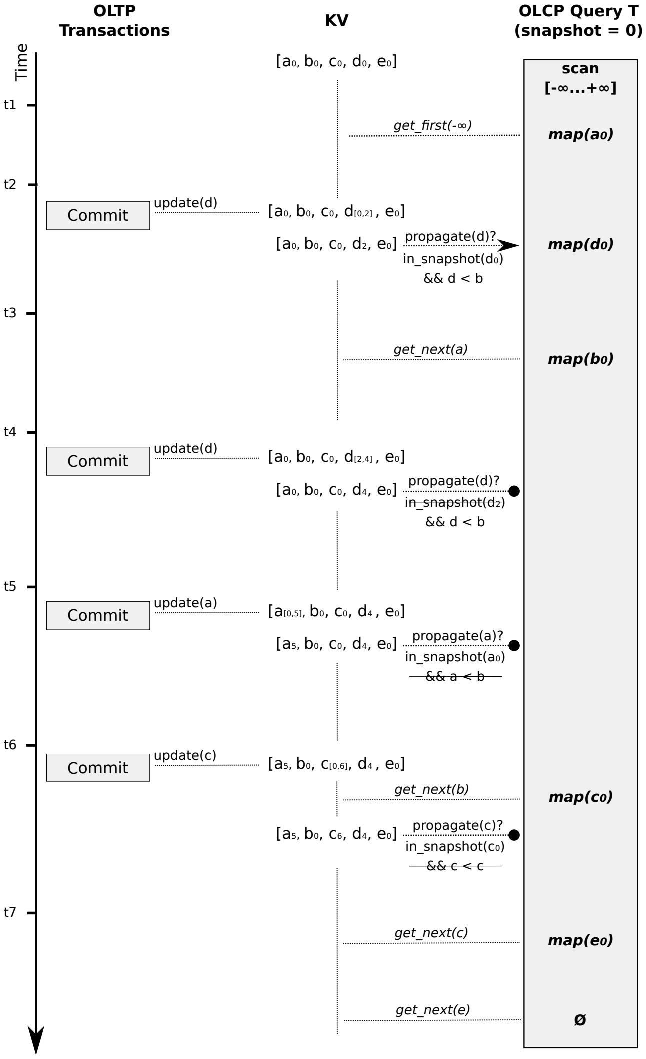


Figure 2: Possible interleavings between a scan and various propagations. Arrows indicate that an item is propagated to the OLCP query, rounded segments indicate that an item is not. At the end, the `map` function has been called exactly once on all items initially contained in the scan range.

3 Using OLCP in practice

Below, we explain how OLCP can be widely used in practice to eliminate space amplification. Analytical processing is typically done in the following three ways:

1. Multidimensional OLAP (MOLAP) analytics,
2. Relational OLAP (ROLAP) analytics, and
3. MapReduce-style analytics.

3.1 MOLAP analytics

MOLAP databases provide a traditional platform for data analytics which is widely used in Business Intelligence applications (e.g., IBM Cognos [14], Oracle Essbase [56], and iccube [35]). The data is first extracted from a relational database, transformed into a specialized multidimensional cube format, and then transferred into the MOLAP database. OLCP can be used during the extraction phase to get a snapshot of the database with little space overhead. Using a conventional implementation of SI, updates performed during the extraction create space amplification, and the relational database may stall once the extraction is complete because of GC. To avoid these issues, database administrators usually run the extraction during the night when the load is low. OLCP allows extractions to occur at any time without space overhead or database stalls.

3.2 ROLAP analytics

ROLAP tools query the main relational database directly through a language like SQL. In this paper we use SQL syntax for simplicity, but other languages with similar constructs can be used as well. Analytical queries consist of a combination of three types of building blocks.

1. Decomposable aggregate functions (e.g., `SUM`, `COUNT`).
2. Aggregate functions (e.g., `GROUP BY`, `CUBE`, `ROLLUP`).
3. Joins.

We show that for these three types of operations OLCP reduces space amplification. *Most ROLAP operations have no space overhead under OLCP.*

Decomposable aggregate functions: Decomposable aggregate functions are the least complex of the three query building blocks. They consist of commutative operations that only require one pass over the data (e.g., `SUM`, `MIN`, `MAX`, `AVG`).

Nutanix uses decomposable aggregate functions to compute simple statistics on a store that keeps track of disk blocks allocated to virtual machines in a datacenter. For instance, Algorithm 2 counts the number of disk blocks that have not been accessed for the last two hours. The query is used to estimate the percentage of allocated storage that is infrequently accessed. Algorithm 3 presents the equivalent using OLCP. For simplicity, we present a sequential version of the algorithm. In practice the `map` function can be called concurrently by multiple threads, and we use per thread payloads that are merged at the end of the scan.

The query is executed with low priority in order to avoid interfering with other workloads. This query used to be executed under read committed to avoid the space amplification overhead of SI (under read-committed, old data is removed from the store and the scan reads the most recent version of committed items). Unfortunately, under read committed, this query overestimates the number of accessed blocks because, when a block is accessed after the start of the query, it is impossible to know if the block was idle in the past two hours prior to the query, since this information is lost after the update. Executing the query with OLCP, and thus with SI guarantees, produces a precise estimate of disk usage.

Algorithm 2 RocksDB pseudocode for counting the number of disk blocks in a datacenter that have not been used in the last two hours.

```
size_t count = 0, target_time = now() - 2;
Iterator *it = ...;
for(it->SeekToFirst();it->Valid();it->Next()){
     block_t *t = it->value;
     if(t->last_access < target_time)
      count++;
}
```

Algorithm 3 OLCP pseudocode for counting the number of disk blocks in a datacenter that have not been used in the last two hours.

```
map(item *i, payload *p) {
    if(i->last_access < p->target_time)
        p->count++;
}
payload p={.target_time = now()-2, .count=0};
t = olcp_query(map, &p, [...], NULL);
commit(t);
```

Aggregate functions: Like decomposable aggregate functions, these queries do not require items to be accessed in order, and the data is accessed only once. The difference is that these queries group items into categories and compute statistics for each group (e.g., using the `GROUP BY` clause and its extensions like `CUBE` and `ROLLUP`).

We illustrate how OLCP reduces space overhead with the first query from the TPC-H suite [74] (Algorithm 4). Algorithm 5 presents its equivalent using OLCP (for simplicity we use the name of the tables to represent the ranges of keys). The query provides a summary pricing report for all items shipped before a given date, aggregated by a flag and a status. This query is more complex than the previous example because it requires grouping analyzed items in buckets and returning them in order. Since the number of flags and statuses is small, the number of buckets is small, and the summaries can be computed in memory. If the number of summaries to be computed was large, the `map` function could use point queries to load and store temporary summary results from disk. After the scan completes, the summaries are sorted in a `reduce` function.

Joins: ROLAP joins are typically *hash joins* or *nested loop joins* [30]. An analysis of the query plans of Microsoft SQL [12] for the TPC-H queries shows that approximately 70% of the joins are hash joins, and the remaining 30% are nested loop joins.

Algorithm 4 First query of TPC-H.

```
select l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty, [...]
from lineitem
where l_shipdate <= '1998-09-04'
group by l_returnflag, l_linestatus
order by l_returnflag, l_linestatus;
```

Algorithm 5 First query of TPC-H using OLCP.

```
map(item *i, payload *p) {
  if(i->l_shipdate < "1998-09-04")
     return;
  string k=i->l_returnflag+"|"+i->l_linestatus;
  p->sum_qty[k] += i->l_quantity;
}

reduce(payload *p) {
   sort(p->sum_qty); // sort p by key
   return p->sum_qty;
}

payload p = { ... };
t = olcp_query(map, &p, [lineitems], NULL);
commit(t);
reduce(&p);
```

Hash joins are usually performed in two steps. First, the join scans the first table, and builds a hash table (build phase). Then, the join scans the second table and probes the hash table for matches (probing phase). Building the hash table only uses one-time commutative reads. By putting the first table in the scan ranges, OLCP avoids any space amplification during the build phase. The probing phase then occurs in the *reduce* function, with the second table in the point ranges. In general, hash joins can easily be ported to OLCP by placing the more frequently updated table in the scan ranges and the less frequently accessed table in the point ranges.

Algorithm 6 presents the fourth query of TPC-H. The query counts the number of orders ordered in a given quarter of a given year in which at least one lineitem (item of an order) is received by the customer later than its committed date. In Microsoft SQL, the build phase is performed on the "orders" table and the probing phase on the "lineitem" table. Algorithm 7 presents a port of this query plan to OLCP. The "orders" table is placed in the scan ranges, and the "lineitem" table in the point ranges. The hash table is built in the `map` function, and the scan of lineitem and the probing is done in the `reduce` function. While the query is running, updates on the "orders" table, or any table that is not accessed by the query, do not induce any space amplification. Items in the "lineitem" table are versioned. If "lineitem" were known to be frequently updated, the query plan could easily be modified to build the hash table using "lineitems" and scanning the "orders" table next.

Algorithm 6 Query 4 of TPC-H in SQL.

```
select  o_orderpriority, count(*) as order_cnt
from orders
where
  o_orderdate >= date '1995-01-01'
  and o_orderdate < date '1995-04-01'
  and exists (
    select *
    from lineitem
    where l_orderkey = o_orderkey
      and l_commitdate < l_receiptdate
  )
group by o_orderpriority
order by o_orderpriority;
```

Algorithm 7 Pseudo code of Query 4 of TPC-H using OLCP.

```
map(item *i, payload *p) {
  if(i->o_orderdate >= '1995-01-01'
    && i->o_orderdate < '1995-04-01')
    p->hash[i->o_orderkey] = ...;
}
reduce(payload *p) {
  // Scan versioned lineitems
  hash_t res = {};
  foreach(lineitem_t l in lineitems) {
    if(p->hash[l->l_orderkey]
        && l->l_commitdate < l->l_receiptdate)
      res[l->o_orderdate].order_cnt++;
  }
  return sort(res);
}
payload p = { ... };
t=olcp_query(map,&p,[order],[lineitem]);
reduce(&p);
commit(t);
```

Nested loop joins iterate over two tables in order. Because of the order constraint, nested loops do not naturally fit the OLCP model. However, it is often possible to adapt nested loops to OLCP with minor changes to the query plan. Query 17 of the TPC-H benchmark (Algorithm 8) is an example of a complex join query that is executed using nested loops in the Microsoft SQL query plans for TPC-H. This query gets items of a given brand that sold five times less than the same item from other brands. It then computes the total revenue loss that would have occurred if these items had not been sold. The query is divided in two sections: *tinner* computes the average number of sales per "partkey" item, regardless of the brand, and *touter* gets the sales information for a given brand.

The number of "partkey" items is small (10K) compared to the number of order items (90M), and orders are aggregated by "partkey". Algorithm 9 presents pseudo code of a possible implementation. Lineitem (list of ordered items) is scanned, and the `map` function simultaneously computes information for the *tinner* and *touter* queries. The `map` function performs one point query to the "partkey" table to get the brand of the scanned item. A `reduce` function then aggregates per-partkey information and outputs the total price of the items that match the criteria. The memory required to execute this query is low (hashtable with 10K entries). The "partkey" table is the only table that is accessed using a point query. Since "partkey" is read-mostly, this query has negligible space amplification when executed with OLCP.

Algorithm 8 Query 17 of TPC-H.

```
select sum(l_extprice) / 7.0 as avg_yearly
from
  (
    select l_partkey, l_quantity, l_extprice
    from lineitem, part
    where p_partkey = l_partkey
      and p_brand='Brand#34'
      and p_container='MED␣PACK'
  ) touter,
  (
    select l_partkey as lp,
       0.2*avg(l_quantity) as lq
    from lineitem
    group by l_partkey
  ) tinner
where touter.l_partkey = tinner.lp
 and touter.l_quantity < tinner.lq;
```

Algorithm 9 Pseudo code of Query 17 using OLCP.

```
map(item *i, payload *p) {
  string k = i->l_partkey;

  // Tinner
  p->tinner[k].l_quantity_sum += i->l_quantity;
  p->tinner[k].l_quantity_count++;

  // Touter
  part_t *part = kv_get("part"+k); // Seek
  if(part->p_brand == "Brand#34"
    && part->p_container = "MED␣PACK") {
    p->touter[k] += {
        l_quantity = i->l_quantity,
        l_extprice = i->l_extprice
    };
  }
}
reduce(payload *p) {
 double l_extprice_sum = 0;
 foreach(string k in p->touter) {
   double lq = 0.2*p->tinner[k].l_quantity_sum/
     p->tinner[k].l_quantity_count;
   foreach(int i in p->touter[k]) {
     if(p->touter[k][i].l_quantity < lq)
       l_extprice_sum+=p->touter[k].l_extprice;
   }
 }
 return l_extprice_sum / 7.0;
}
payload p = { ... };
t=olcp_query(map,&p,[lineitems],[parts]);
commit(t);
reduce(&p);
```

3.3 MapReduce analytics

MapReduce provides a highly parallelizable and scalable framework. This approach is popular for computing simple analytics on vast amounts of data, employed for instance to obtain cluster management statistics [9], to compute popular search-word and query trends [20], and to analyze time-series workloads in IoT, recommender systems and finance [1]. Essentially, the mappers are doing a background scan on the store (e.g., Cassandra, RocksDB), and push the items of interest into a MapReduce system (e.g., Hadoop, CouchDB, Phoenix [52]). These scans take on the order of a few hours and happen concurrently with the foreground workloads, which can be write-heavy [3]. Using the conventional implementation of SI causes prohibitive space amplification because incoming updates need to be tracked over a long time span. To avoid the space explosion, these statistics are usually collected in read-committed mode and thus have lower accuracy. In contrast, OLCP supports consistent one-pass scans, with no space overhead.

4 Implementation

In this section, we describe our implementation of SI and OLCP. The source code of our implementation is available at *https://github.com/BLepers/KVell*. Our implementation adds approximately 4,000 lines of code on top of KVell.

4.1 KVell

As noted in previous work [45], when running on modern fast drives, existing KVs that support SI, such as WiredTiger and RocksDB, run into a CPU bottleneck and are unable to write data at disk speed. As a result they are not suitable for studying space amplification on such drives. We therefore extend KVell [45], a recent KV designed for NVMe SSDs.

KVell has two main components: an ordered index residing in RAM, and an unsorted data structure on disk similar to a slab memory allocator, which groups items with similar sizes in the same file. Reads are either served from a cache (0 I/O), or from disk (1 I/O). Updates fetch a 4KB block from disk, modify it in memory, and then write the dirty block back to disk (1 or 2 I/Os, depending on whether the block was cached or not). The index and the disk data structure are partitioned among multiple worker threads, with each worker handling a range of the key space.

Ideally, analytical queries should not slow down OLTP transactions. Even on modern drives, a fine balance has to be maintained between sending too few simultaneous requests (resulting in sub-optimal bandwidth) and sending too many (resulting in high latency). In its original implementation, KVell scans ranges by reading all items of the range in parallel. We change the implementation of scans to ensure that scans do not overwhelm the disk with with requests. Scans request batches of items from the store, with the size of a batch adjusted depending on the current disk utilization. In practice, we aim at having between 32-64 pending disk I/O requests at all time. When reading the next batch, we adjust the batch size to keep the number of disk I/Os within this bound.

In its original implementation, KVell did not support transactions. We first describe our conventional implementation of SI in KVell+ and the extension to reduce space amplification proposed in Steam [8]. We then describe our implementation of OLCP in KVell+.

4.2 Conventional SI

Our implementation of SI is inspired by those of RocksDB and WiredTiger, two KVs that are widely used in industry.

Timestamps: We add a global logical timestamp in KVell. The global timestamp is incremented every time it is read. When a transaction *commits*, it is given a commit timestamp t_{commit} equal to the current global timestamp. When a transaction *starts*, it is given a snapshot timestamp, $t_{snapshot}$. The snapshot timestamp is chosen so that a transaction can only read data that has already been committed, using the following formula: $t_{snapshot} = min_{active}(t_{commit}) - 1$. If no transaction is committing, the $t_{snapshot}$ is set to the current global timestamp.

In the original version of KVell, persisted items are already timestamped; we use these timestamps in the read and writing path: a transaction can only read or write an item with a timestamp less than or equal to its $t_{snapshot}$.

Writing data: To perform a *write* on a key, a transaction locks the index entry for that key in the main memory index. If the key is not present in the store, a new locked index entry is created. To prevent write-write conflicts, a transaction that fails to lock an item aborts. It also removes all previously acquired locks and any newly created index entries. Before commit, only the in-memory index is updated. The new item versions are kept in a private in-memory buffer (similarly to RocksDB).

Reading data: When *reading* an item, the worker first checks if the item is in its private buffer. If not, the item is read from the main store. If the memory index contains multiple versions of an item, the transaction reads the most recent version that belongs to its snapshot.

Committing updated data: To commit, a transaction persists an tuple "(t_{commit}, N)", where N is the number of updated items. This tuple is used in case of a crash to avoid recovering items from partially committed transactions. The transaction then writes the new items to disk, timestamped with t_{commit}. Once all new items have been persisted, the transaction deletes the "(t_{commit}, N)" tuple. During a commit, the transaction updates the index non-atomically: entries for the new versions are added to the index, and index entries are unlocked as they are updated. This process is safe because no other transaction can read or write any of the updates before the commit ends (by the definition of $t_{snapshot}$), so transactions cannot access partially committed data. Hence, transactions appear "atomically" in the system.

KVell did not use a commit log in its original implementation, and we do not add one to support transactions. This design choice is essential for performance on modern drives. Historically, commit logs were cheap to maintain compared to the cost of updating the store – a fast sequential append vs. a slow random update to a complex data structure. NVMe drives can perform random I/Os as fast as sequential I/Os. In KVell, persisting an item is performed in as low as 1 I/O. Adding a commit log would essentially double the number of I/Os required to perform an update and halve the speed of the store. We acknowledge the usefulness of logs (e.g., for accountability, audit, etc.), and developers might choose to log store accesses via a fast logging system. Our implementation has the advantage of placing logs outside of the critical path.

Garbage collection: After commit, the location of the old versions of updated items are placed in a per-worker cleaning list. Workers periodically check the smallest active $t_{snapshot}$. When this value changes, they scan their cleaning list and delete obsolete items. Workers stop cleaning as soon as they find an item with a timestamp higher than or equal to $min(t_{snapshot})$ (similarly to WiredTiger).

4.3 Steam

Steam [8] uses a more aggressive form of garbage collection that aims to reduce the number of old versions. When an item is updated, Steam scans that item's versions, and deletes the ones that do not belong to any active transaction. Steam was originally implemented in an in-memory database and does not handle recovery in case of a crash. In our implementation, we delay the deletion of old versions to after the commit to avoid deleting versions that might be needed during recovery. Otherwise, our implementation is similar to the original one.

4.4 OLCP in KVell+

OLCP further modifies garbage collection and implements propagation, We also describe a key optimization to avoid extra I/Os as a result of propagation.

Garbage collection: The main difference between OLCP and OLAP is the time during which old versions need to be kept in the store. Workers have two cleaning lists: one for items belonging to the scan ranges, and one for items belonging to the point ranges. GC for point ranges happens as it would under SI. GC for scan ranges happens as described in Algorithm 1.

Propagations: Key to the proper functioning of OLCP is implementing an efficient propagation mechanism. KVell uses an asynchronous interface: threads send requests to the datastore, and the datastore enqueues answers in a per-transaction queue. We build on this mechanism for propagations. Propagating an item I to a OLCP query T consists of enqueuing I in T's queue (as if T had requested to read the item). At the data store level, data is sharded between single threaded workers, so propagations do not introduce any data races. For instance, if an item is propagated "while" being requested by a scan, the scan request and the propagation request are serialized at the worker level and only one of the requests causes the item to be enqueued in the queue. If multiple OLCP queries are running, an item may be enqueued in multiple queues.

Concurrency: In KVell, items are sharded between multiple workers. To speed up queries, we start the scan on all workers. All workers progress in their scan concurrently and may propagate updates concurrently. No synchronization is required between workers because workers work on distinct items. In practice, the scan happens as if multiple single threaded scans were launched on disjoint sets of items.

Avoid reading old versions from disk: In Section 2, old item versions are propagated immediately after committing the new versions. This approach is sub-optimal: updates are not performed in place, and therefore propagating old versions at this time requires reading them from disk, adding an extra read to an update. To eliminate this extra read, we delay propagations and deletions. Instead of propagating and deleting the old versions in the GC (lines 18- 23 in Algorithm 1), we keep the entries for them in the index, and we put their location in a list of reusable spots. When such a spot is later reused, the disk block containing that spot is read, and we take advantage of that to propagate the old version without an extra disk read.

This optimization raises the possibility that versions of the same item might not be overwritten in the order they are created. For instance, if versions of the same data item are of different size, they are allocated in different slabs, and a more recent version may be overwritten before an older version.

Figure 3 presents a case where an item has three versions (k_0, k_1 and k_2). k_2 is the current version. k_0 and k_1 are old versions ($t0 < t1 < t2$). k_0 and k_1 are in the free list of reusable spots and have not yet been overwritten. At the beginning of the execution, the in-memory index still contains all three versions. Indeed, k_0 and k_1 have not been overwritten, and so have not yet been propagated. In Figure 3 an OLCP query T executes with a snapshot timestamp equal to $t1$. Assume that the slot containing k_1 is reused before the one containing k_0, and assume furthermore that k has not been scanned. k_1 is propagated and deleted, since it has not been scanned and it is part of T's snapshot.

However, k_1's index entry must not be immediately removed. In the absence of any record of k_1 in the index, if k_0's slot is overwritten before the scan reaches k, as depicted in Figure 3, it would be propagated to T. Similarly, if T's scan reaches k before k_0 is overwritten, it would be read by the scan. In both cases, T would erroneously read k_0, a version that does not belong to its snapshot. To avoid these situations, we keep k_1 in the index, but flag it as deleted. It then becomes clear that k_0 does not belong to T's snapshot, and it is neither propagated nor read by the scan. Once k_0 has been overwritten, it is removed from the index because it is the oldest version. k_1 is then removed from the index as well because it is now the oldest version and flagged as deleted.

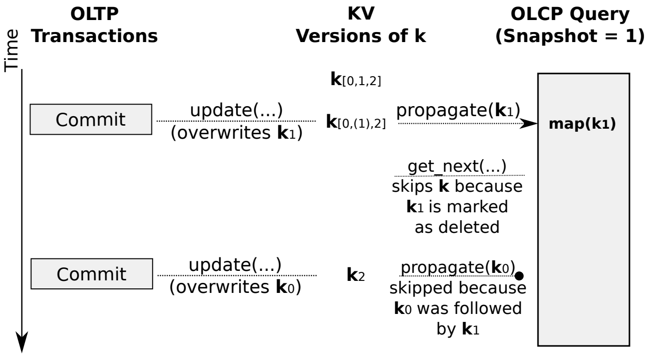

OLTP Transactions
KV Versions of k
OLCP Query (Snapshot = 1)
Time
k[0,1,2]
Commit
update(...) (overwrites k1)
k[0,(1),2]
propagate(k1)
map(k1)
get_next(...) skips k because k1 is marked as deleted
Commit
update(...) (overwrites k0)
k2
propagate(k0) skipped because k0 was followed by k1

Figure 3: Under optimized OLCP old versions might not be overwritten in version order. In that case, the query must skip k_0, even though it would appear to belong to its snapshot ($t0 < t1$).

4.5 OLCP in other stores

In the previous section, we focused on the implementation of OLCP in KVell, but the OLCP paradigm is general and applicable to other datastores and other storage devices. For instance, OLCP can be implemented in RocksDB and WiredTiger and on slower SSDs. In RocksDB, old items can be propagated during compactions: when merging two SSTables, old items can be propagated and discarded with no extra I/Os. Similarly, WiredTiger can propagate old items during checkpointing. OLCP can also be implemented in in-memory databases like Hyper [39]: Hyper maintains free lists of reusable spots, so it can propagate old items when reusing their spot (just as in our KVell+ implementation).

5 Evaluation

5.1 Goals

We evaluate OLCP queries on a variety of synthetic and production workloads. We seek to answer the following questions:

- **Resource utilization:** What is the space amplification of OLCP compared with existing SI implementations? What is the impact of using OLCP queries on throughput and tail latency?
- **Scalability:** How does OLCP scale with the number of concurrent scans and with the size of the store?
- **Performance:** How does OLCP perform on TPC and production workloads?

5.2 Experimental settings

Hardware: We use the following hardware configurations:

Config-AWS. An AWS i3.metal instance, with 36 CPUs (72 cores) running at 2.3GHz, 488GB of RAM, and 8 NVMe SSD drives of 1.9TB each (brand unknown, 2016 technology). The server can sustain a total of 3M read IOPS and 1.4M write IOPS (on read/write workloads, the maximum number of IOPS varies between 1.4 and 3M). The store is configured to cache 30GB of data.

Config-NVMe. A 4-core 4.2GHz Intel i7, 48GB of RAM, and a 480GB Intel Optane 905P (2018). The server can sustain 500K read or write IOPS. The store is configured to cache 20GB of data.

Workloads: We use the following workloads:

YCSB-T: YCSB-Transactional [21] is inspired by the Yahoo! Cloud Serving Benchmark [16] but groups updates in transactions. The average KV item size is 1024B, and the total data set size is approximately 100GB (100M keys) for the small test and 5TB (5B keys) for the large test. Similarly to previous work [77], we perform 16 updates per transaction and items are accessed uniformly. We use this workload to test the limit of SI and OLCP under a write-heavy workload (100% updates).

TPC-CH: The TPC-CH workload [15] mixes the widely popular TPC-C and TPC-H workloads. Currently, TPC-C is the industry standard to simulate OLTP systems [72] and TPC-H is the industry standard to simulate OLAP systems [74]. The TPC-CH workload harmonizes the representation of the data used by TPC-C and TPC-H so that TPC-C and TPC-H queries can run on the same dataset. The TPC-CH benchmarks [15] remove 3 updates that cause most TPC-C queries to fail due to write-write conflicts. Without this modification, TPC-C transactions abort 85% of the time. The abort rate goes down to less than 1% of the time with the modification. Our implementation is similar to the one for Redis [73].

In order to reach a significant database size, we configure TPC-CH to run with 300 warehouses. In that configuration, the store contains 140M items in total, 90M of which represent orders. The rest of the items represent customer data, stock, etc.

Production workloads from Nutanix: The production workloads are two write-intensive workloads, with a profile of 57:41:2 write:read:scan ratio. The KV item sizes range between 250B and 1KB, with a median of 400B. The total dataset size for the production workload is 256GB. The difference between the two workloads is the data skew: The key distribution in Production Workload 1 is close to uniform, while Production Workload 2 is more skewed. OLTP transactions perform on average 10 requests per transaction.

Existing SI implementations: We compare OLCP to the conventional SI implementations and the Steam SI implementation presented in Section 4. In the remainder of this section, we refer to these implementations as "SI" and "Steam", respectively.

5.3 YCSB-T

In this experiment, we run scans of the store concurrently with YCSB-T transactions. The system is disk-bound. We run the experiment with the 100GB dataset on Config-NVMe.

5.3.1 Space amplification

Figure 4 presents the evolution of the number of old versions in time for a Zipfian and a uniform distribution of updates, varying the number of concurrent scans.

Figure 4(a) - 1 scan - Zipfian distribution: Unsurprisingly, the number of old versions increases linearly with time with the standard implementation of SI and, after 24 minutes of execution, the store has accumulated 350 million old versions and runs out of space. Steams keeps at most one version per active snapshot; since we only execute one scan, Steam only keeps at most one old version per item. Running a Zipfian workload concurrently with a single scan is the best case scenario for Steam because most updates are concentrated on a few items. At the end of the scan, Steam has accumulated 50M old versions. OLCP propagates old versions to the scan, and the number of old versions using OLCP is low and stable throughout the run (a maximum of 1000 old versions).

Figure 4(b) - 1 scan - Uniform distribution: Similarly to the Zipfian distribution, the number of old versions grows linearly with the standard implementation of SI. Because updates are distributed over more items, Steam keeps more versions and eventually the store doubles in size. The number of old versions using OLCP is again negligible throughout the run (a maximum of 1000 old versions).

Figure 4(c) - 3 scans - Uniform distribution: In this experiment, we launch a second scan after 500s of execution, and a third scan after 1,000s of execution. The three scans run concurrently. In this configuration, Steam has to keep up to three versions per item. At the end of the execution of the first scan (not shown in the picture), the store has accumulated 250M old versions (store tripled in size). OLCP propagates old versions to the scans and has close to zero space amplification (a maximum of 1000 old versions).

5.3.2 Throuphput

In this section, we study the performance of the scans and the updates when executed with the various SI implementations. We run the uniform workload with a single scan presented in the previous section. Results are similar with the Zipfian distribution and with more scans. Figure 5(a) shows the scan throughput, and Figure 5(b) shows the update throughput.

Figure 5(a): The scan throughput is the same for the standard SI implementation and Steam. Surprisingly, scanning data is much faster using OLCP. The OLCP scan finishes after 691s. With standard SI, the scan aborts after 1460s because the store runs out of disk space (350GB space amplification). With Steam, the scan takes 1870s to complete, 2.7x as long as OLCP. OLCP queries process items just before they are overwritten, and thus when they are in memory. In contrast, with SI and Steam, queries have to fetch most of their data from disk. This advantage is especially visible at the beginning of the scan. As the scan progresses, the advantage of OLCP over SI decreases, because, statistically, as the scan progresses, most of the overwritten items have already been scanned.

Figure 5(b): OLCP scans also interfere slightly less with updates because updates make better use of the caches with OLCP. Updates happen as follows: read a 4KB block (1 I/O if the block is not cached), modify and persist the block (1 I/O). In a uniform workload, the probability of hitting the cache depends on the store size ($P(hit) = cache\ size/store\ size$). Because the database grows less with OLCP, the read has a higher probability of hitting the cache and updates are faster.

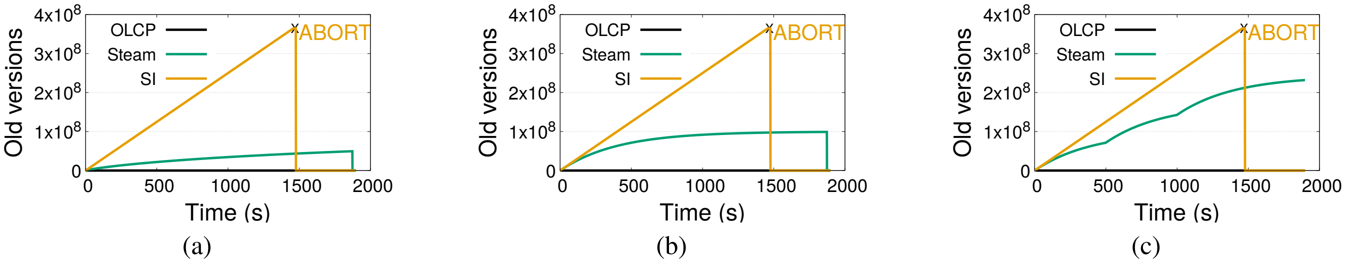
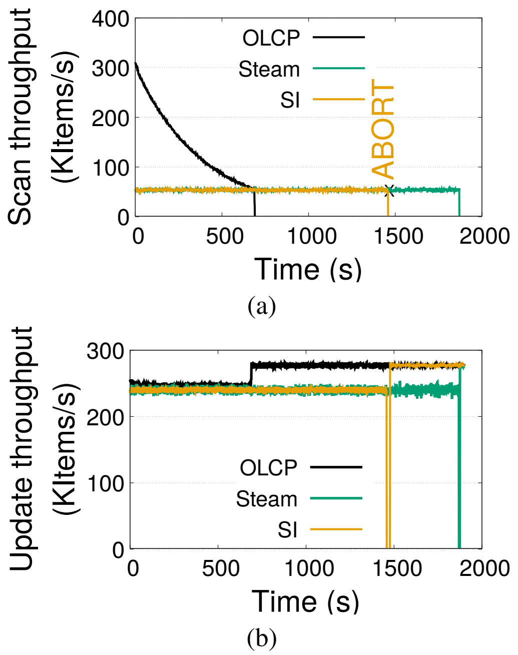


Figure 4: **Config-NVMe**. Evolution of the number of old versions for (a) a Zipfian workload with 1 scan, (b) a uniform workload with 1 scan, and (c) a uniform workload with 3 scans.



(a)

(b)

Figure 5: **Config-NVMe**. Evolution of (a) scanned items/s, and (b) updated items/s. Y-axes differ.

Table 1 shows the tail latency of the updates. At the 99th percentile, it takes 3-3.4ms to commit the 16 updates performed by an OLTP transaction (190-250us per update). Switching to OLCP for scans has no significant impact on the 99th percentile latency of OLTP transactions. In SI, the GC of the 350M old items stalls the store after the scan aborts, so the tail latency is high (18s). Cleaning the 100M old items takes 5s in Steam. A non stop-the-world GC could be used, at the expense of higher average space utilization. In OLCP, regardless of the GC implementation, cleaning overhead is negligible, and tail latency is orders of magnitude lower (9ms).

Latency	**SI**	**Steam**	**OLCP**
99p	3.4ms	3.1ms	3ms
Max	**18s**	**5s**	9ms

Table 1: **Config-NVMe**. Tail latency of OLTP transactions (16 updates).

Config-AWS: Trends are similar on Config-AWS. The full scan of the store finishes in 134s with OLCP and in 256s with SI and Steam. OLTP transactions have similar throughput.

5.3.3 Overhead of propagations

The overhead of propagations depends on the number of running OLCP queries: the more OLCP queries, the more enqueues in propagation queues might be done. The overhead also depends on whether concurrent updates are done on scan ranges or not: an item is only propagated if it belongs to a scan range. We launch up to 32 concurrent scans on two stores containing 100M items (100GB) and 5B items (5TB), respectively. Each scan reads a random range of 1M items. As in the previous section, the scans run concurrently with an update intensive YCSB-T workload. We run all tests on Config-AWS, since it is the only machine able to store 5TB.

Figure 6 presents the average number of scanned items per second, varying the number of concurrent scans. On all tested configurations, OLCP is equivalent or faster than conventional SI and Steam. The difference between OLCP and conventional SI is lower than in the experiments of Section 5.3.2 because (i) the queries only scan a small percentage of the store, so updates are less likely happen in a scanned range and result in a propagation, (ii) the read bandwidth of the disks on Config-AWS is higher than the write bandwidth, so the scan progresses faster than the updates. On the 100M store, the gap between OLCP and SI increases with the number of concurrent scans. Indeed, as the number of scans increases, so does the probability that a propagation happens within a scan range and that OLCP can process items in memory. On the 5TB store this effect is less visible (statistically an update has a lower probability of being in a scan range).

The throughput on the 5TB store is lower than on the 100M store because less data is cached (30% vs. 0.6%). The total number of "scans + updates" requests per second is not constant in the experiments (i.e., adding 100K scans/s does not reduce the update rate by 100K updates/s) because (i) reads are done using at most 1 I/O (vs. 2 for updates), and (ii) Config-AWS disks can sustain a higher number of read IOPS than write IOPS.

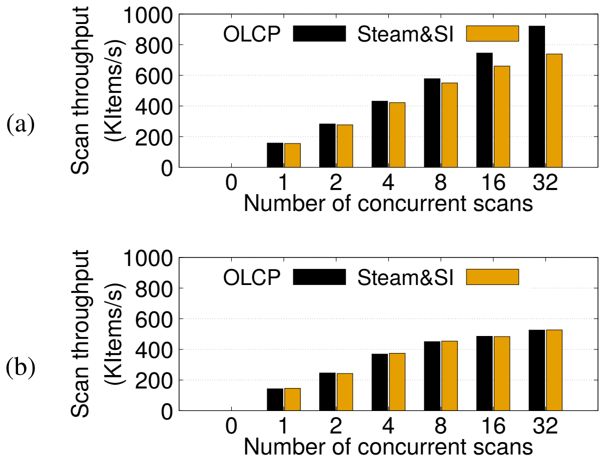


Figure 6: **Config-AWS**. Number of scanned items per second on a (a) 100M and (b) 5TB store, varying the number of scans.

Figure 7 presents the number of updates performed per second. OLCP is slightly faster for the same reasons as those presented in the previous experiment.

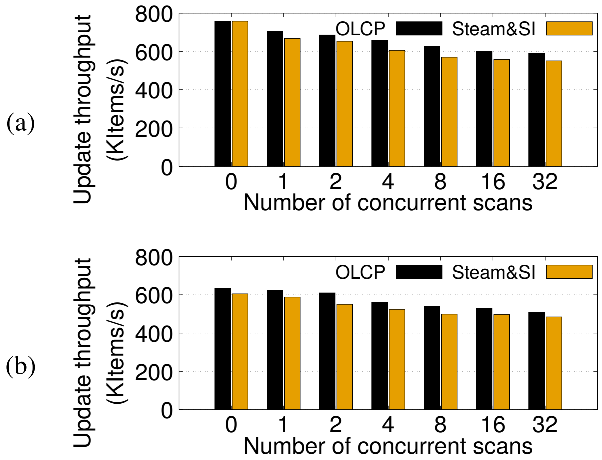


Figure 7: **Config-AWS**. Number of updates per second on a (a) 100M and (b) 5TB store, varying the number of scans.

In conclusion, it is possible to propagate items to OLCP queries with negligible overhead. In all experiments, less than 2% of the time is spent propagating values.

5.4 TPC-CH performance

In this section, we measure space amplification and performance on a TPC-C workload running concurrently with a TPC-H analytical workload. We ran on average 10 TPC-C queries concurrently. Each TPC-C query does an average of 22 requests (17 reads, 5.5 writes), and 30% of the reads hit the cache. Figure 8 presents the throughput of TPC-C running concurrently with TPC-H Query 17, presented in Algorithm 9.

With OLCP, Query 17, which scans 64% of the store, completes without space overhead and creates little interference with TPC-C queries (3% slowdown compared to an execution without a scan). Under Steam, the store doubles in size. Under SI, the store runs out of space, and the query aborts.

5.5 Production workloads performance

We study the performance of conventional SI, Steam and OLCP with the production workloads running on Config-AWS. Figure 9 presents the resulting space amplification, scan throughput and update throughput. At the end of the analytical processing, in Production Workload 2, the store has accumulated 934M old versions with the conventional SI implementation, and GC takes 49s. Steam stores 310M old versions at the end of the analytical processing. OLCP queries cause no space amplification. All implementations have the same throughput.

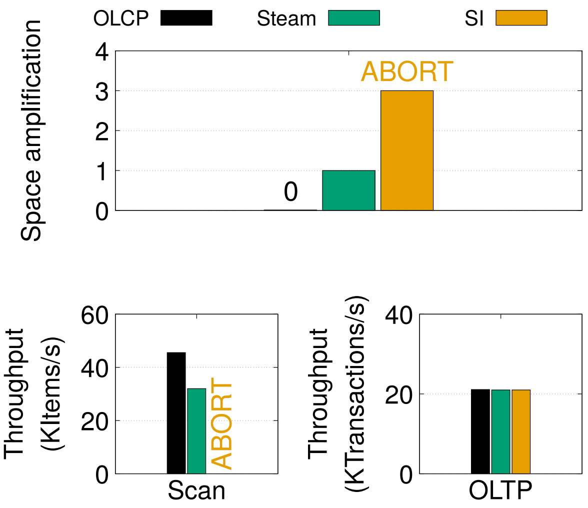


Figure 8: **Config-NVMe**. Space amplification and throughput of TPC-CH.

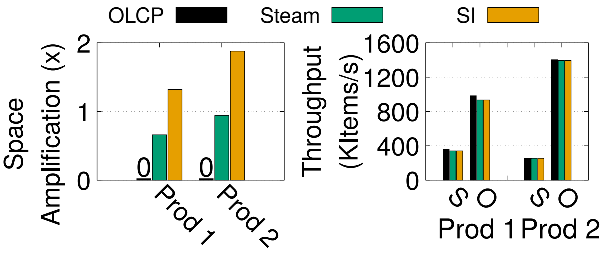


Figure 9: **Config-AWS**, production workloads. Space amplification, S (scan), O (OLTP) throughput.

6 Related Work

Running mixed OLTP/OLAP workloads: OLTP/OLAP workloads are commonly handled by systems that maintain a column-oriented datastore for OLAP (e.g., Vertica [42], C-Store [67], and Hive [71]), isolated from the row-oriented OLTP system (e.g. Cassandra [26], RocksDB [24]). This approach allows to optimize each sub-system independently. The main disadvantages are running analytics on old data and space amplification caused by data replication.

A popular approach to decrease data replication overhead is to design store for OLTP/OLAP workloads from the ground-up [10, 13, 25, 38, 39, 43, 60, 80]. Typically, these systems employ hybrid vertical/horizontal data partitioning schemes, coupled with carefully chosen secondary indexing. A significant drawback of these systems is the performance impact that OLAP and OLTP workloads have on each other (e.g., up to 5x throughput decrease in SAP HANA [63]). In OLCP, the analytics workloads do not impact transactions thanks to OLCP's minimal GC overhead.

Reducing space amplification in SI: The role of SI is to provide a coherent view of the data to OLAP queries [51,66,69]. SI-related space amplification is one of the most challenging issues for stores that run fully in main memory and it has been addressed by many designs. Harizopoulos et al. [28] execute transactions sequentially to avoid MVCC maintenance work. IoSnap provides flash-optimized snapshots that reduce space overhead by reconstructing snapshot metadata in-memory [68]. Hyper [38,39] runs OLTP transactions on a `fork` of the store. BatchDB [51] runs OLAP queries on a replica of the store. These solutions still create problematic space amplification (up to $2x$), and garbage collection times at the end of the execution of OLAP queries. Furthermore, the execution of OLAP queries might be delayed to the next batch, adding possibly minutes/hours of latency to analytical queries. OLCP mitigates the space amplification and garbage collection issues. OLCP model could also be beneficial for in-memory stores.

Space amplification in KVs for fast drives: Much of the prior KVs work relies on SI to provide a consistent view of the data during range scans [2,4,36,37,47,50,57–59,64,65]. Existing systems such as PebblesDB [64], TRIAD [3], WiscKey [50], and HashKV [11] propose optimizations to decrease space amplification caused by compactions in log-structured merge KVs. SlimDB [65] decreases space for caching indexes and filters. Other KVs designed for fast drives do not support transactions [3,5,40,41,45,48]. To the best of our knowledge, OLCP is the first work that focuses on reducing disk space amplification due to SI on fast drives.

Improving the performance of MVCC: In practice, SI is implemented through MVCC [6]. Many recent optimizations and protocols provide support for high transaction rates [8, 33, 46, 49, 55, 75]. Steam [8] trims versions that do not belong to any *active* transaction's snapshot. Steam is efficient for skewed workloads. However, under a uniform load the space amplification is proportional to the number of active transactions. We go one step further by propagating old versions to avoid keeping unnecessary versions in snapshots. Silo [75] chooses provides scalable timestamps and uses RCU to garbage collect old versions. Cicada [49] batches operations to reduce protocol costs. TicToc [77] only keeps the latest version of an item in the store. All these techniques focus on improving the speed of MVCC, but do not address space amplification. They can be used to complement OLCP.

Improving the performance of transactions: Various approaches have been proposed to increase transaction performance such as transactional memory techniques [7, 18, 22, 29, 31, 54, 62], transaction support for byte-addressable persistent memory [27,53], work stealing [79], relaxing ACID properties when possible [17,61,76], decreasing replication overhead [78], and reducing coordination [70]. These techniques are orthogonal to OLCP and can be used together with our model to boost the OLTP workload.

7 Conclusion

Long OLAP queries cause problematic space amplification and long transaction tail latencies when run under SI. To remedy this problem, we propose OLCP, a new query model. OLCP provides the same isolation guarantees as conventional SI implementations, but with much reduced space amplification and interference with concurrent OLTP transactions. We show how OLCP can be used to express a wide range of OLAP queries. We implement OLCP in KVell+, an extension of KVell, a state-of-the-art open-source KV for NVMe SSDs. OLCP achieves low or no space amplification, up to 2x higher throughput for OLAP queries, and order-of-magnitude improvements in tail latency for concurrent OLTP transactions.

References

[1] Nitin Agrawal and Ashish Vulimiri. Low-latency analytics on colossal data streams with SummaryStore. In *Proceedings of SOSP*, 2017.

[2] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. BzTree: A high-performance latch-free range index for non-volatile memory. In *Proceedings of the VLDB Endowment*, 2018.

[3] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of USENIX ATC*, 2017.

[4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *Proceedings of USENIX ATC*, 2019.

[5] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to bε-trees and write-optimization. *;login:*, 40(5), 2015.

[6] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2), 1981.

[7] Jayaram Bobba, Neelam Goyal, Mark D Hill, Michael M Swift, and David A Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of ISCA*, 2008.

[8] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. Scalable garbage collection for in-memory MVCC systems. *Proceedings of the VLDB Endowment*, 13(2), 2019.

[9] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. Curator: Self-managing storage for enterprise clusters. In *Proceedings of NSDI*, 2017.

[10] Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. Es 2: A cloud data storage system for supporting both OLTP and OLAP. In *Proceedings of ICDE*, 2011.

[11] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling efficient updates in KV storage via hashing. In *Proceedings of USENIX ATC*, 2018.

[12] Joe Chang. TPC-H SF100 Non-parallel Plans, SQL Server 2008. http://www.qdpma.com/tpch/TPCH100_Query_plans.html, 2020.

[13] James Cipar, Greg Ganger, Kimberly Keeton, Charles B Morrey III, Craig AN Soules, and Alistair Veitch. LazyBase: trading freshness for performance in a scalable database. In *Proceedings of EuroSys*, 2012.

[14] Cognos. IBM Cognos Analytics. https://www.ibm.com/products/cognos-analytics, 2020.

[15] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, 2011.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of SoCC*, 2010.

[17] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *Proceedings of OSDI*, 2018.

[18] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of ASPLOS*, 2006.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.

[21] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. YCSB+ T: Benchmarking web-scale transactional databases. In *Proceedings of ICDE Workshops*, 2014.

[22] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. Proteustm: Abstraction meets performance in transactional memory. In *Proceedings of ASPLOS*, 2016.

[23] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[24] Facebook. RocksDB: a persistent key-value store for fast storage environments. https://rocksdb.org, 2018.

[25] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database–an architecture overview. *IEEE Data Engineering Bulletin*, 35(1), 2012.

[26] Apache Software Foundation. Cassandra NoSQL key-value store. http://cassandra.apache.org/, 2018.

[27] Kaan Genç, Michael D Bond, and Guoqing Harry Xu. Crafty: Efficient, htm-compatible persistent transactions. In *Proceedings of PLDI*, 2020.

[28] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 2018.

[29] Tim Harris, James Larus, and Ravi Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1), 2010.

[30] Sven Helmer, Guido Moerkotte, et al. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings of the VLDB Endowment*, volume 97, 1997.

[31] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of ISCA*, 1993.

[32] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*, pages 651–665, 2019.

[33] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *Proceedings of the VLDB Endowment*, 13(5).

[34] Hyperdex. Hyperleveldb. https://github.com/rescrv/HyperLevelDB, 2018.

[35] iccube. iccube embedded analytics. https://www.iccube.com/, 2020.

[36] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: Write-optimization in a kernel file system. *ACM Transactions on Storage (TOS)*, 11(4), 2015.

[37] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of USENIX ATC*, 2018.

[38] Alfons Kemper and Thomas Neumann. One size fits all, again! the architecture of the hybrid OLTP&OLAP database management system Hyper. In *International Workshop on Business Intelligence for the Real-Time Enterprise*, 2010.

[39] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of ICDE*, 2011.

[40] Jungwon Kim, Seyong Lee, and Jeffrey S Vetter. PapyrusKV: a high-performance parallel key-value store for distributed NVM architectures. In *Proceedings of SC*, 2017.

[41] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *Proceedings of FAST*, 2019.

[42] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The Vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12), 2012.

[43] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. Real-time analytical processing with SQL server. *Proceedings of the VLDB Endowment*, 8(12), 2015.

[44] Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. Ysmart: Yet another SQL-to-MapReduce translator. In *Proceedings of ICDCS*, 2011.

[45] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of SOSP*, 2019.

[46] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High performance transactions in Deuteronomy. In *Proceedings of CIDR*, 2015.

[47] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of ICDE 2013*, 2013.

[48] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of OSDI*, 2011.

[49] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of SIGMOD*, 2017.

[50] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of FAST*, 2016.

[51] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of SIGMOD*, 2017.

[52] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing mapreduce for multicore architectures. Technical report, Technical Report MIT-CSAIL-TR-2010-020, MIT, 2010.

[53] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, and Dave Dice. Persistent memory transactions (arXiv), 2018.

[54] Shuai Mu, Sebastian Angel, and Dennis Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of EuroSys*, 2019.

[55] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of SIGMOD*, 2015.

[56] Oracle. Oracle essbase. https://www.oracle.com/business-analytics/essbase.html, 2020.

[57] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of USENIX ATC*, 2016.

[58] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An efficient memory-mapped key-value store for flash storage. In *Proceedings of SoCC*, 2018.

[59] Percona. Tokumx. https://www.percona.com/software/mongo-database/percona-tokumx, 2018.

[60] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of SIGMOD*, 2009.

[61] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of OSDI*, 2010.

[62] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of OSDI*, 2008.

[63] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In *Technology Conference on Performance Evaluation and Benchmarking (TPCTC)*, 2014.

[64] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of SOSP*, 2017.

[65] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of VLDB Endowment*, 10(13), 2017.

[66] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of SOSP*, 2011.

[67] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A column-oriented DBMS. In *Proceedings of the VLDB Endowment*, 2005.

[68] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a flash with IoSnap. In *Proceedings of EuroSys*, 2014.

[69] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proceedings of the VLDB Endowment*, 13(2), 2019.

[70] Adriana Szekeres, Michael Whittaker, Naveen Kr. Sharma, Jialin Li, Arvind Krishnamurthy, Dan Ports, and Irene Zhang. Meerkat: Scalable replicated transactions following the zero-coordination principle. In *Proceedings of EuroSys*, 2020.

[71] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2), 2009.

[72] TPC-C. TPC-C, an on-line transaction processing benchmark. http://www.tpc.org/tpcc/, 1992.

[73] Python TPC-C. https://github.com/apavlo/py-tpcc, 2019.

[74] TPC-H. TPC-H a decision support benchmark. http://www.tpc.org/tpch/, 2018.

[75] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[76] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining ACID and BASE in a distributed database. In *Proceedings of OSDI*, 2014.

[77] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of SIGMOD*, 2016.

[78] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4), 2018.

[79] Xiaozhou Zhou, Zhaoguo Wang, Rong Chen, Haibo Chen, and Jinyang Li. Extracting more intra-transaction parallelism with work stealing for oltp workloads. In *Proceedings of the Asia-Pacific Workshop on Systems*, 2017.

[80] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. LiveGraph: A transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment*, 13(7), 2020.