RangeQC: A Framework for Caching Range Predicate Query Results

Shahram Ghandeharizadeh, Yazeed Alabdulkarim and Hieu Nguyen
Database Laboratory Technical Report 2018-03
Computer Science Department, USC
Los Angeles, California 90089-0781
{shahram,yalabdul,hieun}@usc.edu

Abstract

Range predicates are important to many workloads and are supported by both SQL and NoSQL data stores. This study presents RangeQC, a caching framework to expedite processing of range predicates. RangeQC is a framework that scales to a large number of cache servers. It is configurable with alternative data stores, provides strong consistency and supports write-around, write-through, and write-back policies. In addition to presenting its design principles and an implementation for use in a data center, we evaluate RangeQC with both MySQL and MongoDB. Obtained results demonstrate superiority of using RangeQC as long as the size of result set produced by a range predicate is *small*. A surprising result is the superiority of RangeQC with write-heavy workloads.

1 Introduction

Range predicates are essential to many workloads. A variety of techniques have been developed to process these predicates efficiently. Examples include B+-tree [12] and skip-lists [20], LSM-Tree [23] used in Google's LevelDB [17] and WiscKey [21], skip-lists in MemSQL, Bw-tree [20] in Microsoft's Hekaton, and BzTree [8] for the emerging NVM.

In this study, we explore the use of caches to look up the results of range predicates instead of computing them using an index structure. The resulting framework, RangeQC, complements either a SQL or a NoSQL data store to enhance its performance for range predicates with result sets that are *small* in size. Figure 1 shows the throughput improvement of two popular data stores (MySQL and MongoDB) extended with RangeQC using its write-back policy. These results are obtained using a 10 million record YCSB [13] database with a workload consisting of a mix of updates and scans that issue range predicates. Each range predicate fetches 5 YCSB records and its cache entry is 5000 bytes in size. The x-axis of this figure denotes the percentage of scans in the workload, varying the workload from write-heavy to read-heavy. The y-axis is the percentage improvement in throughput observed relative to each data store by itself. With a write-heavy (10% scan) workload, RangeQC

achieves a 400% throughput increase with MySQL. With a read-heavy (99% scan) workload, RangeQC provides a 150% increase in overall system throughput. RangeQC buffers writes in its caching layer and applies them to the data store in the background while MySQL ensures durability by flushing dirty blocks to SSD prior to acknowledging writes. RangeQC continues to increase the throughput by 100% with MongoDB even though MongoDB is configured with the write acknowledged mode (writeConcern = ACKNOWLEDGED) where it buffers writes in memory and flushes them to its SSD every 60 seconds [1]. With hard disk drive as the secondary storage device, the reported percentage improvements with RangeQC is significantly higher.

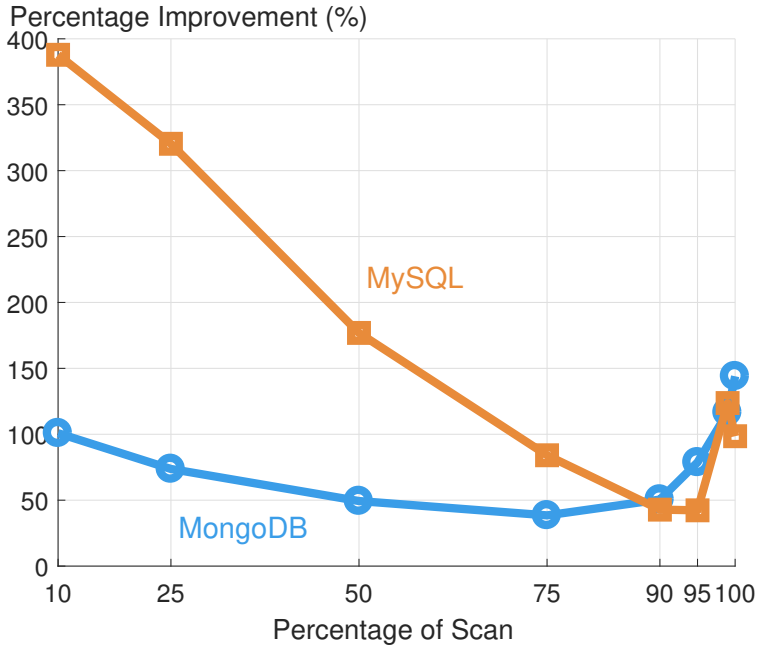


Figure 1: Percentage improvement in throughput with RangeQC and two different data stores.

A range predicate references an attribute of a data item and specifies either (a) a lower bound and an upper bound for its values using mathematical comparison operators, e.g., 10 $<$ temperature $<$ 12, or (b) a lower or an upper bound with a limit on the number of rows to retrieve, e.g., temperature $>$ 10 limit 3, temperature $<$ 11 limit 5, etc. With the latter, the temperature values retrieved from the database define the lower and upper bounds fetched by the predicate.

A caching layer for range predicates must satisfy the following requirements:

- **Strong consistency:** The caching layer must guarantee that data produced by a confirmed write is observed by all subsequent reads assuming the data store provides strong consistency [25].

- **Scale horizontally:** The throughput of the caching layer should increase linearly as a function of its servers.

- **Pauseless:** Failure of one or all cache servers should not disrupt service or compromise strong consistency guarantees. As long as the data store remains available, the system should process application read and write requests.

- **General purpose for use with alternative data stores and caches.** The caching layer should use off-the-shelf hardware and software components. It must function with alternative data stores based on different data models, e.g., MySQL using the relational data model and MongoDB using the JSON data model.

- **Support alternative policies for processing writes.** These include write-around, write-through and write-back. While write-around deletes those cached entries impacted by an update to the data store, write-through updates them. Write-back updates the cached entries and buffers the write in the caching layer. It may store multiple replicas of a buffered write to enhance its availability. The degree of replication should be a configurable parameter. Different applications may require different write policies and the caching layer must provide all three policies.

RangeQC realizes the above requirements using off-the-shelf hardware and software components. It uses leases [16] to provide strong consistency and prevent undesirable race conditions that insert stale data in the cache. It implements all three write policies. It uses configurable log records to tolerate failures of its in-memory data structures.

We evaluate RangeQC using the YCSB benchmark [13]. Obtained results show performance of RangeQC is maximized with the write-back policy and range predicates with small cached entries. Moreover, it scales horizontally with write-back.

We compare RangeQC with Redis' implementation of range predicate. Obtained results show RangeQC's superiority for read heavy workloads. While Redis is superior for write-heavy workloads, it produces stale data (termed *anomalies* [7]). This highlights the requirement for RangeQC's synchronization primitives (leases) to provide strong consistency.

The rest of this paper is organized as follows. Section 2 provides an overview of RangeQC and its interfaces. Section 3 details a specific implementation of RangeQC. We describes how consistency and durability are provided in Section 5 and 6. We evaluate this implementation in Section 7 and discuss the findings in Section 8. Section 9 surveys related work. Our future research directions are outlined in Section 10.

2 General Framework of RangeQC

RangeQC is a scalable plug-N-play framework consisting of one or more Dendrites. A *Dendrite* is an extensible component that facilitates communication with (1) a Cache Manager Instance (CMI) such as memcached and (2) a data store such as MySQL or MongoDB. It implements the following interfaces:

RangeQC Dendrite instance DI = Initialize (Collection-name, Domain-name, Domain-type, Coordinator address, Write-Policy, β)

An application must invoke Initialize to obtain an instance of Dendrite for get, insert, delete, and update commands. This method obtains the necessary configuration parameters from the Coordinator, including the identity of other Dendrites and their assigned ranges. With

implementation of Section 3, it also obtains IP:Port address of the individual CMIs from the Coordinator. The arguments of Initialize include:

- Collection-name identifies the table (SQL) or collection (MongoDB) that is referenced by the range predicate.
- Domain-name specifies an attribute that is queried using range predicate. It pertains to a column of a table (SQL) or a property of a collection (MongoDB).
- Domain-type may be an integer, text, real, float, date.
- Coordinator address specifies the network address (e.g., a URL, IP:Port) of the coordinator for obtaining leases on ranges.
- Write-Policy is either write-back, write-through, or write-around.
- β is the maximum tolerated percentage of duplicated data between two cached result sets.

Value = DI.Get (Lower-bound, Upper-bound, Query execute function)
Get is invoked on an instance of Dendrite obtained using the Initialize method. Lower and upper bound values define the interval referenced by the range predicate of the query. Their values are inclusive and must match the domain-type inputted to Initialize. Query execute function is invoked with a cache miss to compute a cached entry. In our implementation of Section 3, a cached entry is identified as a (key,value) pair where value is the cached result set while the key uniquely identifies this value.

Success = DI.Insert (Lower-bound, Upper-bound, DML command, DML execute function, Cache update function)
Insert is invoked on a Dendrite instance of RangeQC obtained using Initialize method. Lower and upper bound specify the interval impacted by the insert. Their values are inclusive and must match the domain-type inputted to Initialize. DML command is issued at run time and inserts a new row (document) in the table (collection). DML execute function is a function used to apply the DML command to a table (collection). Cache update function is specified with a write-through and write-back policy. It describes how to update the cached entry impacted by this insert.

Success = DI.Delete (Lower-bound, Upper-bound, DML command, DML execute function, Cache update function) with a description similar to Insert.

Success = DI.Update (Lower-bound, Upper-bound, DML command, DML execute function, Cache update function) with a description similar to Insert.

The value of β limits the percentage of duplicated data across the cached result sets of two predicates relative to the predicate with the fewest number of records. Its value ranges from 0% to 100%. While $\beta > 0$ allows two cached result sets with duplicated data, $\beta = 0\%$

de-duplicates them to (a) maximize utilization of cache space by increasing its cache hit rate, and (b) minimize the number of cached entries impacted by insert, delete, and update commands.

A Dendrite is extensible and allows a software developer to implement the interfaces for insert, delete, modify that impact the cached result sets of a range predicate. These interfaces are the building blocks for implementing query processing. A Dendrite employs an order-preserving in-memory data structure to maintain three different variables named Predicate Interval Tree (PrInT), Lease Manager Interval Tree (LeMInT), and Buffered Write Interval Tree (BuWrInT).

PrInT indexes the cached predicates and their result sets. The lower and upper bounds of a predicate identify an *interval*. It is associated with the key of the cached result-set. PrInT stores this pairing by indexing the interval, facilitating fast lookups when the predicate is issued again. Given a predicate, PrInT may identify other cached entries that either overlap or contain this predicate. These can be used to compute the results of a predicate, see discussions of Section 8. To simplify discussion and without loss of generality, the following describes processing predicates issued repeatedly with $\beta = 100\%$.

With writes (insert, delete, update), the Dendrite interface requires the application to specify the interval [lower-bound, upper-bound] impacted by the write's range predicate. If the predicate is exact-match (i.e., an equality predicate such as id=C where C is a constant) then its specified interval is a point ([C,C]). RangeQC uses this interval to index into PrInT and identify the impacted cached entries. With the write-around policy, the Dendrite deletes the impacted cached entries. With the write-through and write-back policies, the Dendrite invokes the input "Cache update function" to update each impacted cached entry. Both write-around and write-through update the data store by invoking the provided DML execute function with the input DML command.

With the write-back policy, a Dendrite maintains a Buffered Write Interval Tree (BuWrInT). This tree associates the interval impacted by the write with the key of its buffered write stored in the cache. The buffered write maintains changes by the write and is applied to the data store asynchronously[1].

RangeQC uses BuWrInT to process a predicate that observes a cache miss as follows. First, it uses the interval referenced by the predicate to identify the buffered writes that overlap it. Next, it applies these writes (if any) to the persistent store and removes them from BuWrInT. Finally, it issues the predicate to the persistent store for processing, caches the result set, and associates it with the interval corresponding to the predicate in the PrInT.

With concurrent threads, a Dendrite preserves consistency of the cached entries using a Lease Manager Interval Tree, LeMInT. This tree maintains Shared (read) and eXclusive (write) leases on intervals that are read and written, respectively. A get obtains a Shared lease on its referenced interval while an insert/delete/update obtains an eXclusive lease on its referenced interval. The maximum number of entries in LeMInT is dictated by the number of concurrent threads processing predicates.

There are a variety of ways to implement Dendrites: In a client library, a data store, a CMI, a middleware that uses a data store, in a trusted mobile[2] client with intermittent

[1]Order of applying writes may be important, see Section 4.

[2]An untrusted mobile client may open possibilities for a Denial of Service (DoS) attack or data corruption.

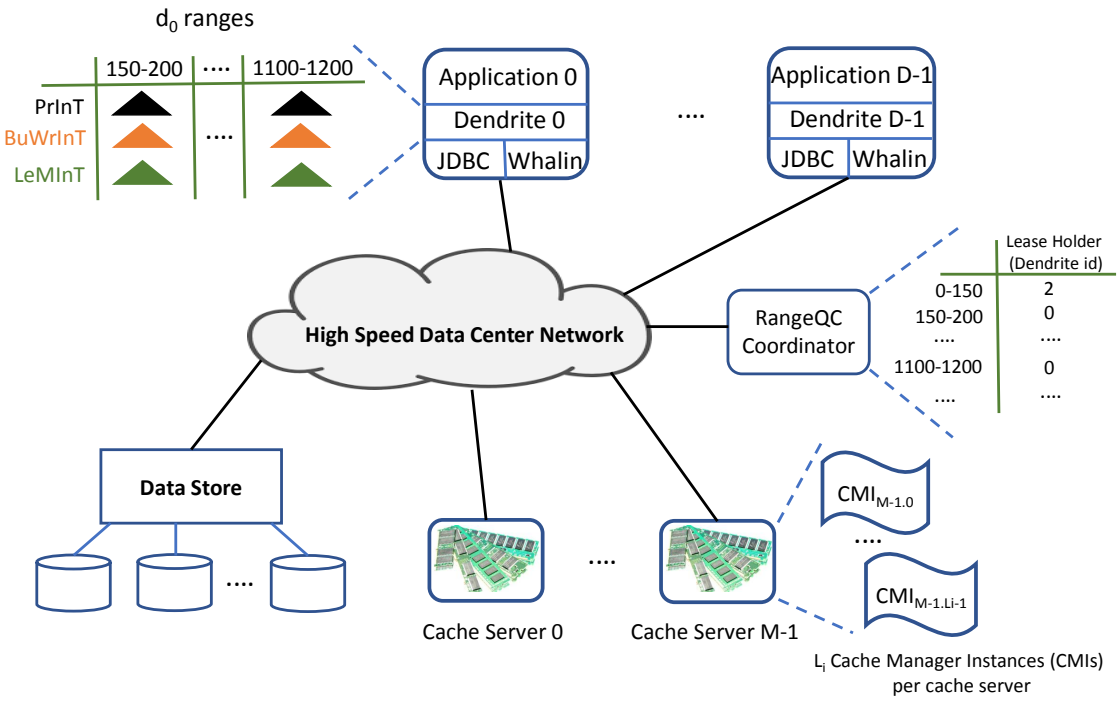


Figure 2: A scalable multi-node implementation of RangeQC with Dendrite as a library. Alternative architectures are possible such as separating a Dendrite to be a standalone process that an application invokes using the network, or pushing the memory of cache servers to be managed by a Dendrite.

network connectivity, among others. Next section describes one implementation as a client library. Section 8 compares this implementation with an alternative that implements a Dendrite in a CMI.

3 A Scalable Implementation of RangeQC

This section describes an implementation of RangeQC using IQ-Twemcached [16] as the CMI and either MySQL or MongoDB as candidate data stores, see Figure 2. It consists of D Dendrites and one Coordinator. The Dendrite may be a standalone process or a library component. Figure 2 shows the latter where a Dendrite is a library integrated in a Client. This figure shows the Coordinator as a stand-alone process. There may be shadow Coordinators that resume the responsibility of the Coordinator once it fails, similar to RamCloud's [24] Coordinator.

The domain of the attribute (property) referenced by the predicate is range partitioned across D Dendrites. Dendrite D_i is assigned d_i ranges. A Dendrite obtains its ranges from the Coordinator. The Coordinator assigns a range to at most one Dendrite by granting it a lease with a fixed life time. A Dendrite may renew its lease to continue serving requests for the range [6, 29].

Given D Dendrites and d_i ranges assigned to each Dendrite D_i, there are $\sum_{i=0}^{D-1} d_i$ instances of PrInT, LeMInT, and BuWrInT. Each implemented using an interval tree [19, 28].

When Dendrite D_i fails, the coordinator assigns its d_i ranges to other Dendrites with the objective to balance its load across the remaining Dendrites. When a Dendrite D_r is assigned a range of D_i, it must re-construct PrInT and BuWrInT of this range prior to processing requests that reference it. This requires D_r to discover the cache entries (to reconstruct PrInT) and buffered writes (to reconstruct BuWrInT) of its newly assigned range. This is trivial if a cache manager implements interfaces for D_r to enumerate entries of a range. Such an interface is not available with an off-the-shelf cache manager such as memcached or Redis. In this case, a Dendrite generates *log* records for each range and stores them in a CMI. A log record pertains to insertion and deletion of an interval in an instance of PrInT or BuWrInT.

RangeQC log records are configurable. They may be co-located with their referenced key-value pairs, replicated across multiple CMIs, pinned in volatile memory to prevent eviction by the cache replacement technique, persisted to secondary storage of a CMI to tolerate cache server failure, or persisted to NVDIMM-N to implement durability in the presence of data center power failure. See next section for details.

In our implementation, the key that identifies a cached entry is a function of the collection-name, domain-name, and the specified lower and upper bound for a range predicate, see Figure 3a. When a Get for a predicate is issued to Dendrite D_i, D_i constructs its key and looks it up in the cache server S_j, $j = h(key)$. If it finds the results then it produces it as output for consumption by the application. With a cache miss (or a write such as insert, delete, or update), Dendrites cooperate with one another by processing these requests on behalf of one another. Only the Dendrite(s) with the assigned range overlapping the predicate may process either a cache miss or a write action.

When a Client invokes a Dendrite D_i to process a cache miss, D_i looks up the range partitioning information to determine which Dendrite should service this request. If no range contains or overlaps this predicate, D_i contacts the Coordinator for a lease on this predicate. The Coordinator either (a) provides D_i with updated range partitioning information that identifies another Dendrite responsible for the range or (b) grants D_i a lease on a range that contains the referenced predicate. With the former, D_i forwards the predicate to the Dendrite responsible for this range. With the latter, D_i assumes responsibility for the range by constructing its PrInT, LeMInT and BuWrInT. While LeMInT starts as an empty tree always, D_i retrieves log records for PrInT and BuWrInT from the IQ-Twemcached servers to reconstruct the content of these trees.

Once these trees are constructed, D_i processes a cache miss by obtaining a Shared lease on the interval referenced by the predicate in LeMInT. D_i executes the function that queries the database to compute the key-value pair (k_j, v_j) representing the cache entry. D_i pairs k_j with the interval corresponding to its referenced range. It generates a log record for this insertion and stores it in server S_c, i=h(k_j). Next, it stores this (k_j, v_j) in the IQ-Twemcached server S_c. It inserts pairing of interval with k_i in PrInT. Finally, D_i releases its Shared lease on the referenced interval in LeMInT.

The above applies to write-around and write-through. Write-back is similar with one difference. Prior to querying the data store, it uses BuWrInT to identify buffered writes that overlap the interval referenced by the predicate. It applies these buffered writes to the data store and then executes the function that queries the database for the (k_j, v_j) pair to store in the cache. This provides strong consistency with write-back.

Dendrite D_i processes a write, such as insert, delete, or update by requiring it to specify

```
Get(lb, ub, QFunc)
1: key= computeKey(lb, ub).
2: value= cache.get(key).
3: if (cache hit) then
4:     return value.
5: cache.acquireILease(key).
6: LeMInT.acquireSharedLease(lb, ub).
7: if (policy== write-back) then
8:     buffKeys = BuWrInT.getKeys(lb, ub).
9:     buffWrites = cache.readAndAcquireQLeases(buffKeys).
10:    db.apply(buffWrites).
11:    cache.deleteAndReleaseQLeases(buffKeys).
12:    BuWrInT.delete(lb, ub).  // generates a log record.
13: value= db.get(rangeQuery).
14: cache.setAndReleaseILease(key, value).
15: PrInT.insert(lb, ub, key).  // generates a log record.
16: LeMInT.releaseSharedLease(lb, ub).
17: return value.
```

(a) Get operation.

```
Update(lb, ub, DML cmd, DML Func, Cache Update  Func)
1: LeMInT.acquireExclusiveLease(lb, ub).
2: overlappers= PrInT.overlappers(lb, ub).
3: for each interval in overlappers  do
4:     key= interval.getKey() from PrInT.
5:     cache.readAndAcquireQLease(key).
6:     if (policy== write-back or write-through) then
7:         Execute Cache Update Func using key as its input.
8:     if (policy== write-around) then
9:         cache.delete(key).
10:        PrInT.delete(lb, ub).  // generates a log record.
11:    cache.releaseQLease(key).
12: if (policy== write-around or write-through) then
13:    db.apply( DML cmd, DML Func).
14: if (policy= write-back) then
15:    buffKeys, buffWrites = generte(DML Func, DML cmd).
16:    cache.set(buffKeys, buffWrites).
17:    BuWrInT.insert(lb, ub, buffKeys).  // generates a log record.
18: LeMInT.releaseExclusiveLease(lb, ub).
```

(b) Update operation.

Figure 3: Pseudo-code for Get and Update. Insert and Delete pseudo-code is similar to Update. Inhibit and Quarantine leases [16] are provided by IQ-Twemcached.

its referenced interval [lb, ub], see Figure 3b. The write specifies the same value for the lower and upper bounds when specifying a point, i.e., an equality predicate such as id=5 is represented as [5,5]. D_i obtains an eXclusive lease on the interval in LeMInT. Next, D_i looks up the overlapping and contained intervals in PrInT to identify the impacted cached entries. With write-around, D_i deletes the impacted key-value pairs from the cache along with their intervals in PrInT. In contrast, with write-through and write-back policies, D_i executes its input *Cache update function*, see Section 2 and Figure 3b. In our implementation, this function updates the cached entry using a read-modify-write operation.

Next, with write-around and write-through, D_i applies the write to the data store. With write-back, D_i generates the buffered write and its log record. It appends the log record first and then inserts the buffered write in the caching layer, replicating these records at least 3 times and pinning them in the cache. It inserts the (interval, key) pairing in BuWrInT. Finally, D_i releases its exclusive lease of its interval in LeMInT.

3.1 Log Records

RangeQC employs configurable log records to recover from a Dendrite failure. Consider a range of a failed Dendrite D_i assigned to D_r. D_r must re-construct D_i's PrInT and BuWrInT for this range prior to its failure in order to process requests. PrInT is required to decide whether a write impacts a cached entry. BuWrInT is required with the write-back policy only. It identifies buffered writes and their intervals.

A Dendrite generates log records for each interval it either inserts or deletes from PrInT or BuWrInT. The configuration of a log record dictates whether it is co-located with its referenced cache entry, replicated for high availability, pinned in memory, persisted for durability, or a hybrid of these.

PrInT log records are configured to be co-located with their referenced cache entry in a CMI, pinned in its memory, and have one replica. A CMI failure destroys both the cache entry and its log record because CMI memory (DRAM) is volatile.

BuWrInT log records are also configured to be co-located with their referenced buffered writes and pinned in memory of a CMI. However, they have 3 or more replicas across CMIs assigned to different cache servers, see Figure 2. This enhances availability in the presence of cache server failures. (Buffered writes are also replicated 3 or more times on the same CMIs as the log records.) Finally, the log records (along with their referenced buffered writes) are configured to persist to NVDIMM-N to implement durability in the presence of data center power failure, see Section 6. Both buffered writes and their log records cannot be lost because they pertain to acknowledged writes.

A log record contains the following information:

- insert:LSN:lowerbound:upperbound:key
- delete:LSN:lowerbound:upperbound

The first entry of the log record is the action, either insert or delete. It is followed with the Log Sequence Number (LSN), a monotonically increasing number for the log records corresponding to a range. The lower and upper bounds identify the interval impacted by

the action. While insert maintains the identity of the impacted key, this information is not required for the delete log record.

An application write may impact multiple intervals of PrInT. With write-around, D_i deletes cache entries corresponding to each interval. It also deletes each interval from PrInT and generates its "delete" log record. These log records are hash partitioned across IQ-Twemcached servers using their referenced key.

With insert into either PrInT or BuWrInT, its log entry is generated and appended first. This results in two KVS operations: Append to the log record, insert the key-value pair in the cache. When deleting an interval from either PrInT or BuWrInT, D_i deletes the cache entry from IQ-Twemcached server CMI_j synchronously and generates its log records to be appended asynchronously. This does not compromise either strong consistency or correctness because the existence of an interval in the interval-tree does not mean the corresponding key must exist in the cache (e.g., it may have been evicted by IQ-Twemcached's LRU policy). This does enhance performance by minimizing the number of round-trip messages issued by D_i to a cache server for deletes from PrInT and BuWrInT.

Our implementation of asynchronous generation of delete log records maintains M buffers, one for each cache server. When a delete for an interval is issued, we hash partition on the key to append its delete log record to buffer j=h(key). Once the buffer reaches a threshold, say α, one append[3] is issued for the α log records, as long as the maximum value size is not violated.

Batching of delete log entries may cause them to be out of order with the inserts. When applying the log records to reconstruct an interval tree, a Dendrite sorts the entries using their LSN prior to applying them to construct the interval tree.

Multiple Dendrites may generate log records for the same CMI concurrently. To minimize contention for a single cache entry (key-value pair), a Dendrite partitions the log records across multiple cache entries on a CMI.

Many inserts and deletes may reference the same lower and upper bounds. We compact the log records by eliminating the inserts and deletes that nullify one another, i.e., reference the same lower and upper bounds.

4 Write-back and Dependent Buffered Writes

With the write-back policy, multiple writes may impact either the same interval or overlapping intervals. RangeQC's synchronization primitives (LeMInT and IQ leases) dictate their serial execution to the cache entries. The background threads must apply the buffered writes to the data store in the same serial order.

RangeQC maintains the serial order of writes by extending a buffered write to maintain the identity of one or more buffered writes that it depends on. This results in an acyclic graph. Prior to applying a buffered write to the data store, all buffered writes it depends on must be applied first. This may navigate multiple buffered writes recursively. The termination condition is when a buffered write depends on no other buffered write or all buffered writes it depends on do not exist (because they have been applied to the data store

[3]We extended IQ-Twemcached with a command that appends multiple values to a key.

and deleted). At that point, the buffered write is applied and processing of buffered writes falls out of recursion.

The dependency graph between buffered writes is constructed incrementally as RangeQC executes writes that reference overlapping intervals one by one. Processing of a write inserts its referenced interval associated with its buffered write in BuWrInT. If this insert results in one or more overlapping intervals then there are buffered writes it depends on.

Example 4.1. Consider two writes W_1 and W_2 where W_2 is serialized after W_1. W_1 sets the salary of those Employee documents with id in the interval [10,12] to 80K. W_2 gives a 500 dollar salary raise to those Employee documents with id in the interval [9,11]. Processing of W_1 generates a buffered write bw_1 associated with [10,12] in BuWrInT. When processing W_2, its buffered write bw_2 is associated with [9,11] in BuWrInT. The interval of W_2 overlaps the existing interval of W_1 in BuWrInT and processing of W_2 makes bw_2 depend on bw_1. When a read R_1 for employee id 9 observes a cache miss, it queries BuWrInT and finds an overlapping interval [9,11] associated with bw_2. Because bw_2 depends on bw_1, R_1 applies bw_1 followed with bw_2. It deletes bw_1 and bw_2 from the cache, and deletes their intervals [10,12] and [9,11] from BuWrInT.

5 Strong Consistency

RangeQC uses leases instead of locks for strong consistency. Leases granted on a data item have a fixed lifetime. If the lease expires (due to the grantee failing), its data item is released making it available for processing once again.

RangeQC uses two types of leases: LeMInT leases granted by a Dendrite on an interval, and IQ leases granted by an IQ-Twemcached CMI on a key-value pair. These leases serve different purposes. LeMInT leases prevent read-write and write-write race conditions on cached intervals. Since LeMInT is an order preserving interval tree, it detects conflicts between a reader/writer and a writer acquiring Shared/eXclusive leases referencing intervals that overlap, causing one to wait for the other to either commit or abort. The IQ leases implement the concept of a *session* [16]. An application implements its read and write actions using a session that acquires either an Inhibit or a Quarantine lease on a key-value pair. Once a write is granted a Q lease on a key k_i, IQ-Twemcached constructs a copy of v_i and applies all updates of this session to this copy. Once the application commits a session, it releases its Q lease and switches copy of v_i with v_i atomically. Should the application fail prior to committing, its Q lease times out and its copy of v_i (that may have been updated) is discarded.

We provide a formal proof of strong consistency of RangeQC for different write policies. We assume the data store provides strong consistency.

Definition 5.1. A range predicate is represented as an interval r_i defined as $[l_i, h_i]$ where $h_i \geq l_i$.

Definition 5.2. Two intervals $r_i, r_j, i \neq j$ overlap if $l_i \in [l_j, h_j]$ *or* $h_i \in [l_j, h_j]$*or* ($l_i \leq l_j$ *and* $h_i \geq h_j$) *or* ($l_i \geq l_j$ *and* $h_i \leq h_j$). The latter defines the interval r_i contained inside r_j as overlapping.

Definition 5.3. The result set of a range predicate r_i is represented as a key-value pair (k_i, v_i) where k_i=l_i:h_i and v_i = result set.

Definition 5.4. A buffered write for an interval r_j is represented as a key-value pair (k_j^{bw}, v_j^{bw}) where k_j^{bw}='BW':l_j:h_j and v_j^{bw} = changes made by writes that impacts interval r_j.

Theorem 1. *RangeQC serializes concurrent conflicting requests that reference overlapping intervals.*

There are five cases that result in conflicts:

- Two or more concurrent writes that reference overlapping intervals.
- Two or more concurrent writes that reference no overlapping intervals and reference at least one key-value pair in the cache.
- At least one read and one write executing concurrently and referencing overlapping intervals.
- Two or more concurrent reads observing a cache miss and their intervals reference at least one common buffered write.
- Applying buffered writes of two or more write requests with overlapping intervals to the data store in the same order they are applied to the cache.

Generalization of each handles more than two concurrent requests. These result in the following five lemmas. After the proof of each lemma, we illustrate it using an example.

Lemma 1.1. *RangeQC serializes two concurrent write requests, W_j and W_l, that reference overlapping intervals.*

Proof. By definition, interval r_j of W_j overlaps with interval r_l of W_l. LeMInT serializes W_j and W_l by granting an eXclusive lease to one write, say W_j. The other write, W_l, must back-off and try again until W_j releases its lease by either committing or aborting. If W_j's lease expires, W_j must abort and retry. ■

Example 5.1. Assume there is a Dendrite for range predicates referencing attribute *age* of *Employee* table (collection). W_j updates the salary of employees with age [20,24] to be 100K. W_l updates the salary of employees with age [19,22] to be 90K. The two intervals [20,24] and [19,22] overlap on age from 20 to 22. Assuming W_j is granted its eXclusive lease on [20,24] using LeMInT, LeMInT detects W_l's request for an eXclusive lease on [19-22] as a conflict, causing W_l to backoff and retry until W_j commits or aborts. W_l is serialized after W_j, resulting in salary of employees with age from 20 to 22 to be 90K. Otherwise, if W_l acquires an eXclusive lease on [19,22] first, W_j back-offs and retries on acquiring an eXclusive lease on [20,24] until W_j commits. W_j is serialized after W_l, resulting in salary of employees with age from 20 to 22 to be 100K.

Lemma 1.2. *RangeQC serializes two concurrent write requests, W_j and W_l, that reference two or more non-overlapping intervals and reference at least one common cached key k.*

Proof. By definition, r_i of cache entry (k_i, v_i) overlaps r_j of W_j and r_l of W_l in PrInT. As a result, both W_j and W_l race to acquire a Q lease on k_i. With the write-through and write-back, Q leases that reference the same key are incompatible. One write is granted the Q lease on k_i while the other must back-off and try again until the lease holder either commits or aborts. This serializes the two writes.

With write-around, Q leases are compatible because deleting the same key multiple times produces the same result, i.e., no (k_i, v_i) in the cache. Since both writes W_j and W_l update the data store, their order is dictated by the data store synchronously. ■

Example 5.2. Assume a cache entry (k_i, v_i) for the range predicate [18,22] of the age attribute of the Employee table (collection). PrInT uses its interval [18,22] to index k_i. Assume W_j updates salary of employees with age 20 to be 100K and W_l updates salary of employees with age 22 to be 90K. Both writes acquire eXclusive leases on their intervals (i.e, [20,20] and [22,22] in LeMInT) successfully because their referenced intervals are mutually exclusive. Each looks up PrInT to find [18-22] associated with k_i overlaps with its interval. Hence, they race to acquire Q leases on k_i to update (or delete) its value. With write-through or write-back, one has the Q lease granted while the other aborts and retries, dictating their serial order. With write-around, both acquire Q leases successfully, delete k_i, apply their write to the data store, and commit to release their Q leases. The data store dictates serial order W_j and W_l with write-around.

Lemma 1.3. *RangeQC serializes two concurrent read and write request, R_i and W_j, that reference overlapping intervals.*

Proof. Let (k_i, v_i) denote the referenced cache entry of R_i. There are two cases:
Case 1: v_i is in the cache. Since W_j and R_i reference overlapping intervals, W_j deletes k_i with write-around and updates v_i with write-through and write-back. With the latter, the write acquires a Q lease on k_i and constructs a copy of v_i, v_i^c, leaving the original copy of v_i to be read by R_i. If R_i reads v_i then it is serialized before W_j. Once W_j commits, it releases its Q lease and swaps v_i^c and v_i atomically with write-through and write-back and deletes k_i with write-around. With write-through and write-back, if R_i observes the new updated value v_i^c, then it is serialized after W_j. With write-around, R_i observes a cache miss and is serialized after W_j per Case 2 below.
Case 2: When v_i is missing from the cache, LeMInT serializes R_i and W_j. If R_i acquires the Shared lease on its interval r_i first, it is serialized before W_j. This is because W_j's eXclusive lease is not compatible, forcing it to back-off and try again until R_i releases its lease by either committing or aborting. R_i queries the data store, computes (k_i, v_i), and inserts it in the cache. This serializes R_i before W_j. Otherwise, if W_j acquires its eXclusive lease on its interval r_j first, forcing R_i to back-off and re-try until W_j commits or aborts. This serializes R_i after W_j. ■

Example 5.3. A read R_i uses the range predicate [18,22] on the age attribute of the Employee table (collection). A write W_j uses the predicate age=20 to update salary to 100K. The interval for the write is [20,20]. If the cached key k_i of interval [18,22] exists, R_i may consume its value v_i. If v_i includes the changes made by W_j, R_i is serialized after W_j. Otherwise, R_i is serialized before W_j.

With a cache miss for the interval [18,22], an I lease is granted on k_i=18:22 and R_i acquires a Shared lease on the LeMInT interval [18,22]. W_j aborts and retries when acquiring an eXclusive lease on its interval [20,20] since it overlaps with the interval [18,22]. In this case, R_i does not observe the changes made by W_j and is serialized before W_j. If W_j already acquired an eXclusive lease on [20,20], R_i aborts and retries until W_j commits. With write-around and write-through, R_i queries the data store to compute the value that includes the changes made by W_j, inserts it in the cache, releases the I lease and the Shared lease on its interval in PrInT. With write-back, R_i applies the buffered write to the data store before querying it to get the changes made by W_j. With all policies, R_i is serialized after W_j.

Lemma 1.4. *RangeQC serializes two concurrent read requests, R_i and R_j, that observe cache misses and their intervals reference at least one common buffered write (k_i^{bw}, v_i^{bw}).*

Proof. Without loss of generality, assume R_i acquires a Q lease on k_i^{bw}, if its value v_i^{bw} is non-idempotent, converts it to be idempotent and releases the Q lease. In the presence of failure, when the lease of one request expires, the other may acquire a Q lease on k_i^{bw} and applies the idempotent changes one more time without losing consistency. One read, assuming R_i acquires the Q lease on k_i^{bw}, applies its buffered write v_i^{bw} (now becomes idempotent) to the data store, deletes k_i^{bw} from the cache and releases the Q lease. R_j acquire Q lease on k_i^{bw} and find no buffered write, then it releases the Q leases. As a result, changes of buffered write v_i^{bw} are only applied once to the data store. ■

Example 5.4. A write W updates the salary of employees whose age is 20 to be 100K. With write-back, a buffered write key k^{bw} is generated to store the changes of this write (idempotent) and is associated with interval [20,20] inserted in BuWrInT of the Dendrite for attribute *age* of *Employee* table (collection). Two reads R_i and R_j queries the employees whose age is [19,20] and [20,21], respectively. They acquire the I leases on their keys and the Shared leases on their intervals ([19,20] and [20,21]) on LeMInT successfully. They look up BuWrInT and find that [20,20] overlaps with their intervals and is associated with k^{bw}. They acquire Q leases on k^{bw} to buffer their changes in its value. Only one may proceed at a time, dictating their order. If R_i acquires the Q lease first, it applies the changes to the data store, deletes k^{bw} and releases the Q lease. R_j acquires the Q lease and release it just to find no buffered write to apply, then queries the data store to compute the latest value. Because R_i already applied the changes of W to the data store, R_j also observes the changes that all employees with age 20 have salary 100K.

Lemma 1.5. *RangeQC applies buffered writes of overlapping intervals to the data store in the same order they are applied to the cache entries.*

Proof. Consider two buffered writes W_j and W_l that reference overlapping intervals. If they are concurrent then their writes to the cache are serialized per Lemma 1.1 and 1.2. Assume RangeQC applied them to the cache in the order W_j followed by W_l. BuWrInT detect their overlapping interval and makes buffered writes of W_l dependent on those of W_j. To apply buffered writes of W_j to the data store, buffered writes of W_l is applied first.

■

Example 5.5. Consider two writes W_j and W_l where W_l is serialized after W_j. W_j sets the salary of those Employee documents with id in the interval [10,12] to 80K. W_l gives a 500

salary raise to those Employee documents with id in the interval [9,11]. Processing of W_j generates a buffered write key k_j^{bw} to store its changes v_j^{bw} and is associated with interval [10,12] in BuWrInT. Processing of W_l looks up BuWrInT with its interval [9,11] and finds k_j^{bw}. It generates its buffered write key-value pair (k_l^{bw}, v_l^{bw}) and makes it depend on k_j^{bw}. As a result, RangeQC applies v_j^{bw} before v_l^{bw} because v_l^{bw} depends on v_j^{bw}.

6 Durability of Writes

With the write-back policy, any loss of buffered writes causes RangeQC to lose its acknowledged writes, compromising durability. Log records of BuWrInT are also required to reconstruct BuWrInT in the presence of Dendrites failures. Without BuWrInT, an application is not able to discover buffered writes to apply to the data store. Hence, we replicate both buffered writes and log records of BuWrInT to enhance availability in the presence of CMI failures. Each buffered write or log record has three replicas on three CMIs assigned to different cache servers. With a write, all replicas of its buffered write and the log record for BuWrInT must be stored successfully before it is acknowledged as success. Failure to store a replica results in executing the write using the write-through policy (after applying all relevant buffered writes.) Key-value pairs for buffered writes and log records are pinned in memory, preventing their eviction, see Sections 3 and 7.1 for details.

Replication by itself is not enough to survive data center power failure that causes all the pending buffered writes and log records of BuWrInT to be lost. These key-value pairs should be stored on non-volatile memory such as NVDIMM-N [27]. Its DRAM may survive shortly after power failure using a backup power source that lasts long enough to flush its DRAM content to flash for persistence.

7 Evaluation

This section evaluates RangeQC using two different data stores based on different data models, MySQL and MongoDB. Our objective is to demonstrate the tradeoffs associated with RangeQC and to quantify these tradeoffs to highlight its benefits and limitations. The key lessons learned from this evaluation are as follows:

1. RangeQC with the write-back policy provides the highest performance benefit.

2. RangeQC scales horizontally (with write-back) to support a higher throughput as a function of its cache servers.

3. The percentage improvement observed with RangeQC depends on the size of cached entries. This size depends on the selectivity factor of the range predicate and its number of projected attributes. When cached entries are large, RangeQC may provide a performance inferior to a data store by itself.

We compare RangeQC with how Redis implements processing of range predicates. With a read heavy workload, RangeQC is superior to Redis by utilizing clients to maintain PrInT. With a write heavy workload, while our implementation with Redis is significantly faster,

it produces anomalies. This highlights the need for LeMInT to prevent undesirable race conditions that produce stale data.

We compare 0% and 100%, the extreme values of β, showing $\beta = 0\%$ is superior with a write-heavy workload. With a read-heavy workload, $\beta = 100\%$ is superior.

We use Emulab [30] d430 nodes with 10 Gbps network connectivity for all experiments. Each node is a Dell poweredge R430 server configured with two 2.4 GHz 64 bit 8-Core Haswell processors, 64 GB of DRAM, 200 GB of Intel SATA SSD, and two 1 TB SATA hard disk drive. We vary the number of nodes used for the caching layer from 1 to 8. Each node hosts 16 IQ-Twemcached instances each configured with 3 GB of memory. There are 16 YCSB clients issuing requests. Each is configured with a Dendrite instance of RangeQC. A single node is dedicated to a data store, either MySQL or MongoDB configured to use 48 GB of DRAM. To maximize the performance of each data store, we used the SATA SSD in all our experiments. We have conducted experiments using the hard disk drive and observe RangeQC to provide a significantly higher performance benefit compared with a data store by itself.

Presented results are obtained using YCSB databases consisting of one and ten million records. The size of each record is 1000 bytes. Unless stated otherwise, the ten million record database is used for evaluation purposes.

We consider two different YCSB workloads with uniform access pattern. Both generate a fixed percentage of scans and updates. The scan fetches α records (α=5 unless specified otherwise). The update modifies columns of a record using its primary key value. The difference between the two workloads is the number of columns fetched by the scan and modified by the update. While scan/update fetches/updates all ten 100 byte columns of the qualifying records with $YCSB^*$, it fetches/updates only one 100 byte column of the qualifying record with $YCSB^1$. We refer to a workload using its percentage of scan, e.g., f=10% scan. The remainder are updates, e.g., 1-f=90%.

7.1 Log Records, Pinned Memory, and Durable Writes

We extended IQ-Twemcached to implement the concept of pinned key-value pairs which is used for log records and buffered writes. When a Dendrite inserts a key-value pair as pinned, IQ-Twemcached may not evict it using its replacement policy. Instead, the Dendrite must specifically either unpin or delete this key-value pair to free memory.

If pinned key-value pairs exhaust cache space, it returns a memory full error. A Dendrite that observes this error (1) switches to write-through by applying a buffered write to the data store synchronously, and (2) stops generating PrInT log records by not caching predicate result sets.

In our implementation, the size of a log record is 18 bytes plus the key size. The latter is determined by the specified collection and domain names with Initialize. Table 1 shows the size of BuWrInT log records with different number of partitions for a write-heavy (10% scan) and a read-heavy (99% scan) workload. With both, the total occupied size is in the order of tens and hundreds of megabytes. The size is significantly smaller with the read-heavy workload because there are fewer buffered writes. Compaction benefits both workloads by reducing the total memory occupied by the log records. Moreover, increasing the number of partitions has an insignificant impact on the total size of log records.

Table 1: Size of BuWrInT log records in MB before and after compaction for write-heavy and read-heavy workloads with different number of partitions. Scan cardinality is 5.

% Scan	32 Partitions		64 Partitions		128 Partitions	
	Before	After	Before	After	Before	After
10	265.39	166.21	288.82	159.26	297.57	151.56
99	55	0.63	56.37	0.24	56.07	0.27

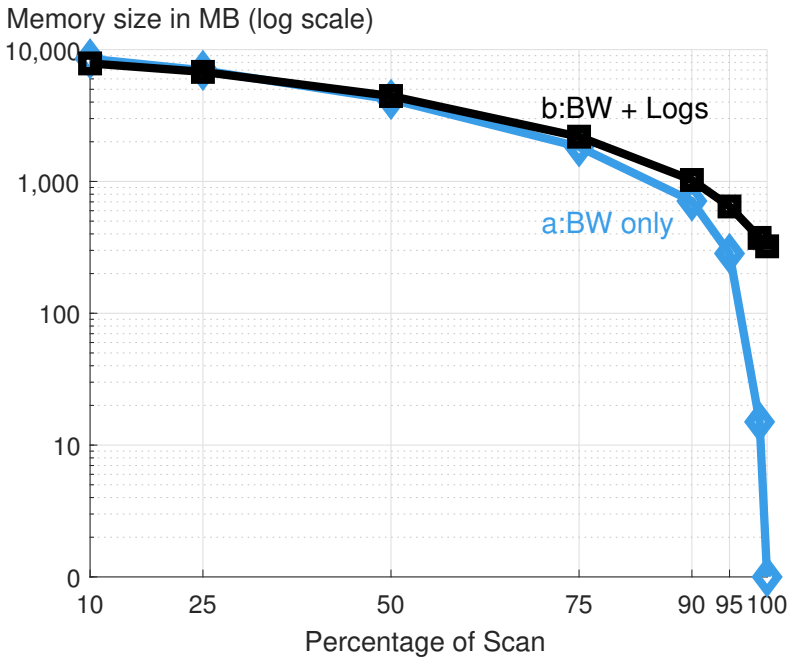


Figure 4: Pinned cache memory (MB) for Buffered writes only and for Buffered writes + log records for BuWrInT and PrInT.

With BuWrInT, the amount of pinned memory includes the buffered writes. Figure 4 shows the total size of memory occupied with (a) buffered writes only and (b) with buffered writes, log records for BuWrInT, and log records for PrInT. With write-heavy workloads (scan frequency of 50% and lower), the difference between these two is not significant because the size of buffered writes dominates. With read heavy workloads, the size of log records starts to dominate.

In Figure 4, the memory size with buffered writes only is higher with the write-heavy workload, 10% scan. This counter-intuitive result is because its throughput is higher as it performs less work on behalf of each request. Hence, it generates and pins more buffered writes in memory.

Figure 5 shows how the size of pinned memory behaves with read heavy workloads. With 99% scan, the buffered writes are typically applied to the data store immediately by both the background worker threads and the scans that observe a cache miss for intervals with pending buffered writes. This explains why the size of pinned memory is either 0 or close to 0 most of the time. The spikes are due to MongoDB checkpoints performed periodically. During a checkpoint, all writes to MongoDB must wait, causing writes to buffer in memory

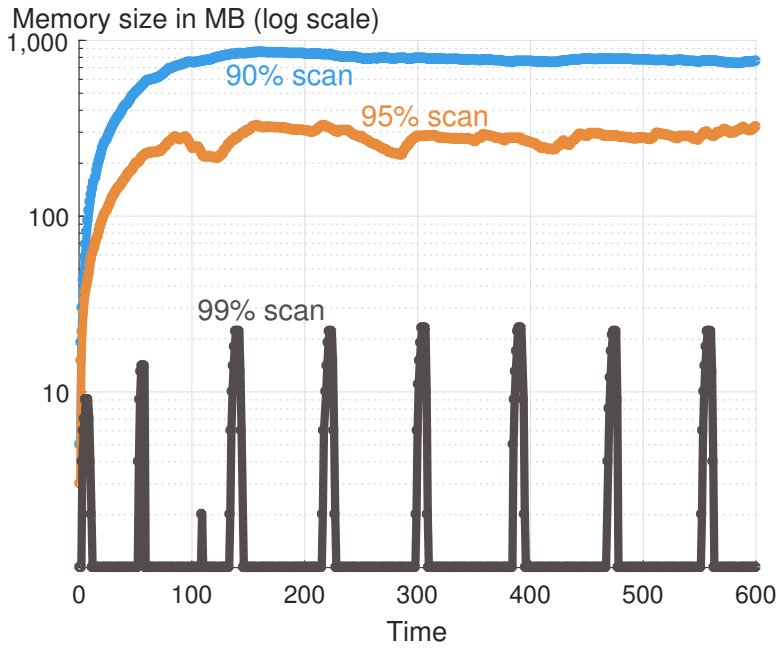


Figure 5: Pinned cache memory (MB) for Buffered writes with read heavy workloads.

and the size of pinned memory to spike. With 90% and 95% scan, the size of buffered writes stabilizes for two reasons. First, multiple buffered writes for the same record are represented as one idempotent write. Second, background worker threads (and gets that observe a miss) apply buffered writes to the data store at the same rate the YCSB workload generator issues writes. With these workloads, spikes due to MongoDB checkpoints are not visible due to use of log scale for y-axis.

Table 2: MongoDB throughput (× 1000 actions/sec) with 1 and 3 degree of replication for buffered writes and their log records with a configuration consisting of 8 cache servers. Scan cardinality is 5.

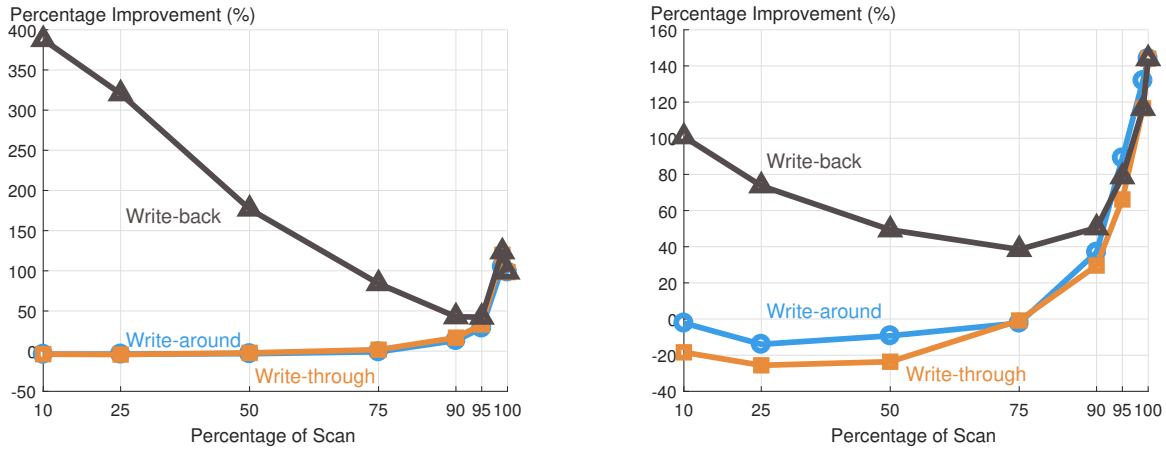
% Scan / Num Replicas	10	25	50	75	90	95	99
1 Replica	240	290	429	724	1,072	1,270	1,491
3 Replicas	212	258	376	687	1,068	1,268	1,496

Write-Back Policy and Durability: With the write-back policy, one may configure RangeQC to replicate both the buffered writes and the log records generated for BuWrInT. Table 2 shows the throughput of an eight cache server configuration with 1 and 3 replicas for buffered writes and log records.

Table 2 highlights the following lessons. First, when the scan constitutes more than 95% of workload (last two columns), the overhead of replicating buffered writes and log records on 3 cache servers is insignificant. Second, this overhead becomes more noticeable with

write-heavy workloads. Even with 10% scan, the impact of maintaining 3 replicas on system throughput is 15%.

7.2 Alternative Write Policies



(a) MySQL. (b) MongoDB.

Figure 6: Percentage improvement observed with alternative write policies using $YCSB^*$ and scan cardinality 5.

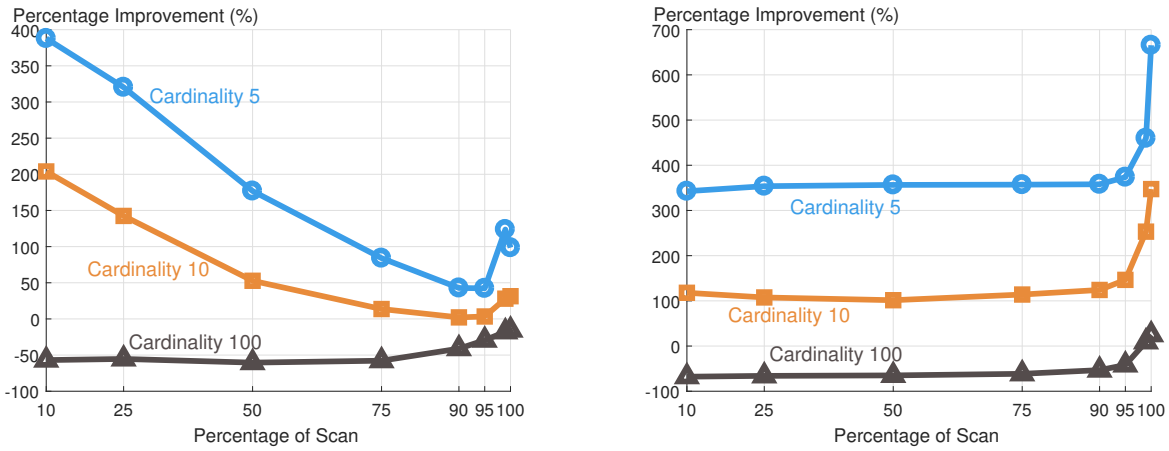
Figure 6 shows the percentage improvement with RangeQC and alternative write-policies when compared with a data store by itself. The x-axis of this figure shows different mixes of update and scan commands. The y-axis reports the percentage improvement in throughput obtained by measuring the throughput of the data store by itself (T_{DS}) and once extended with RangeQC using one IQ-Twemcached server ($T_{RangeQC}$), $\%Improvement = \frac{T_{RangeQC} - T_{DS}}{T_{DS}}$. A y-axis value below 0 means RangeQC degraded the performance of a data store instead of enhancing it.

Figure 6 shows the write-back policy outperforms the other two policies by a wide margin with a write-heavy workload, 10% scan. It is able to do so by buffering writes in the cache, eliminating the data store from the critical path of the write operations. Both write-around and write-through perform the update to the data store synchronously. They incur the delay attributed to a sustained queue of requests on the SSD of the data store due to the high frequency of updates. Moreover, their cache misses incur the same delay. With the write-back policy, a scan that observes a cache miss applies the pending buffered writes for its referenced interval (if any) prior to querying the data store. Ten Background threads apply the buffered writes to the data store. This enables the write-back policy to provide a superior performance with write-heavy workloads.

The write-around and write-through policies enhance performance when scan constitutes more than 90% of the workload, i.e., a read-heavy workload. These workloads eliminate the sustained queue of requests on SSD of the data store, causing the network bandwidth of the cache server to dictate the overall system performance. Hence, all write policies provide comparable performance benefits.

7.3 Size of Cached Entries

Size of cached entries impacts the performance benefit observed with RangeQC significantly. There are two reasons for this. First, with read-heavy workloads, the network bandwidth is the bottleneck resource and a small cached entry increases the number of simultaneous requests processed (throughput) by a cache server. Second, both write-through and write-back update a cached entry using a read-modify-write operation. Its time is dominated by the network transmission time which is a function of the cached entry size, number of impacted keys, and the available network bandwidth.



(a) $YCSB^*$. (b) $YCSB^1$.

Figure 7: Percentage improvement using write-back policy with different scan cardinalities and number of projected columns (MySQL).

Size of cached entries is dictated by the number of records fetched by a scan (i.e., scan *cardinality*) and its number of projected columns. Figure 7 shows the percentage improvement with RangeQC and MySQL when compared with MySQL by itself[4]. It presents results from the two workloads, $YCSB^*$ and $YCSB^1$, using the write-back policy and different scan cardinalities. The percentage improvement with $YCSB^1$ is dramatically higher because it retrieves only one column of each record, i.e., for a given scan cardinality, the size of its cached entry is ten times smaller.

Table 3 shows the cache hit rate with write-back, 99% scan workload, different scan cardinalities for $YCSB^*$ and $YCSB^1$ with 1 Million and 10 Million records. With $YCSB^*$ and scan cardinality of 100, cache hit rate is low due to duplication of records across different cached result sets, e.g., the record with id 500 may appear in 100 different cached result sets. This is due to our use of $\beta = 100\%$. As we increase the size of YCSB database from 1 million to 10 million records, the number of unique predicates increases. This causes a drop in the cache hit rate because the available memory is fixed at 48 GB. This explains why we observe no benefit with $YCSB^*$ and scan cardinality of 100, see Figure 7a.

$YCSB^1$ projects one column of each record, reducing the size of a cached entry by a factor of 10. This enables RangeQC to provide a significant benefit when the scan fetches either 5

[4]Similar results are observed with MongoDB.

Table 3: Cache hit rate with write-back, 99% scan workload, different scan cardinalities, and $\beta = 100\%$.

Scan cardinality	1 Million $YCSB^*$	1 Million $YCSB^1$	10 Million $YCSB^*$	10 Million $YCSB^1$
5	100%	100%	70.54%	100%
10	100%	100%	35.81%	100%
100	37.33%	100%	3.80%	28.28%

or 10 records. The performance benefit observed with 100 records remains either negative or marginally positive due to the large size of the payload. Moreover, an update may impact a record that is duplicated across multiple cached entries, resulting in read-modify-write of multiple cached entries.

Compression reduces the size of a cached entry, enhancing cache hit rate and reducing network bandwidth usage. For example, with cardinality of 100, use of zlib [3] reduces result set size by 50% with YCSB* and 80% with YCSB[1]. With YCSB[1], cache hit rate is enhanced by almost a factor of two. This enhances the observed throughput with RangeQC specially with read-heavy workloads. For example, with YCSB[1] and 99% scan, the throughput of MySQL with RangeQC is 71% higher than the throughput of MySQL by itself.

7.4 Scalability

Table 4: MongoDB throughput (×1000 actions/sec) as a function of the number of cache servers for different workloads. Scan cardinality is 5.

% Scan / Num Servers	10	25	50	75	90	95	99	100
1	37	39	47	64	116	169	254	268
2	59	72	106	181	270	319	375	391
4	122	152	214	364	538	636	743	783
6	188	224	323	547	775	955	1,120	1,173
8	254	298	424	725	1,072	1,270	1,497	1,562

RangeQC scales horizontally as a function of the number of cache server allocated to it. Table 4 shows the throughput of MongoDB with RangeQC with different mixes of scans and updates as a function of the number of cache servers. These numbers are gathered with the write-back policy using MongoDB[5], preventing the data store from limiting the scalability of the framework.

Table 5 shows system scalability normalized to a configuration consisting of 2 cache servers. As we scale the system horizontally by increasing the number of servers four folds from two to eight, the observed improvement in throughput is almost 4 folds. This holds true for all workloads.

[5]Similar trends are observed with MySQL.

Table 5: Scalability relative to two cache servers.

% Scan / Num Servers	10	25	50	75	90	95	99	100
2	1	1	1	1	1	1	1	1
4	2.06	2.09	2.02	2	1.99	1.99	1.98	2
6	3.17	3.08	3.05	3.01	2.86	2.99	2.99	3
8	4.27	4.09	4	3.99	3.96	3.97	3.99	3.99

We do not report scalability numbers relative to one cache server as the trends are misleading. With one cache server, there is not sufficient memory for the entire data, resulting in cache evictions. With two or more cache servers, there are no evictions. Hence, with one cache server, the system utilizes the data store to process a fraction of scans. With two or more cache servers, almost all scans are processed using the caching layer. Since the network bandwidth is the bottleneck with both a read-heavy (100% scan) and a write-heavy workload (10% scan), the cache misses cause the one cache server configuration to provide a higher throughput, resulting in sub-linear scalability. With one cache server and other read and write mixes, say 75% scan, the cache CPU exhibits a high utilization due to evictions. With 2 or more cache servers, the cache network bandwidth becomes highly utilized, resulting in super-linear scalability numbers. Reporting scalability numbers relative to two servers avoids these idiosyncratic results.

7.5 A Comparison with Redis

Redis provides ZADD and ZRANGE methods for processing range predicates using Zindex [4]. The idea is to invoke ZADD for every inserted key-value pair using the attribute value referenced by the range predicate: key = attribute value, value = key of the impacted cache entry. This builds an order preserving index (similar to a secondary B+-tree index) that can be queried using ZRANGE by specifying an inclusive range of values. ZRANGE returns keys of the qualifying cache entries (similar to record identifiers, RID, of a B+-tree index). Subsequently, a multi-get is issued for these keys to obtain the value of the qualifying records. In sum, this implementation processes a range predicate using two Redis calls: A ZRANGE followed with a multi-get.

We compare Redis Zindex with RangeQC using an experimental setup consisting of two servers: 1 cache server hosting 16 Redis CMIs and one server hosting the YCSB client. With RangeQC, there are three servers: 1 cache server hosting 16 IQ-Twemcached CMIs, 1 server hosting the YCSB client, and 1 server hosting MongoDB. MongoDB is not in the critical path of processing requests because we use the write-back policy with a warm cache that provides a 100% hit rate.

To provide an apple-to-apple comparison, we modified RangeQC to store only points (instead of intervals) in its caching layer. With a scan that fetches q documents (records), we transform the result into q individual cache entries and register them with PrInT. To process a range predicate, RangeQC looks up PrInT to identify the qualifying keys and issues a multi-get for them.

With writes, we wanted to eliminate the data store from impacting our comparison. Hence, we assumed YCSB client for Redis maintains an in-memory hash table for the modified data. A YCSB update simply applies its change to its in-memory hash table and sets the value of modified attributes using Redis SET command. The YCSB update impacts an attribute different than the indexed attribute. Hence, it does not update the Zindex. Updating a local in-memory hash table gives Redis an added advantage compared with RangeQC. With write-back, RangeQC must buffer the writes in the cache, incurring a network round-trip time.

RangeQC processes an update by inserting two key-value pairs in IQ-Twemcached: one for the buffered write and a second for its log record[6]. In addition, it invokes the Dendrite client library to set a lease in LeMInT on the interval impacted by the update, indexes the buffered write using BuWrInT, and checks PrInT to identify impacted cache entries. This last Dendrite operation identifies a cache entry (due to a 100% cache hit rate). RangeQC reads, modifies, and writes the cached-entry using IQ-Twemcached, adding two more IQ-Twemcached calls. In sum, RangeQC performs four IQ-Twemcached operations for an update.

The simple Redis implementation fails to provide strong consistency and produces anomalies. To prevent anomalies, one must extend it with LeMInT and IQ leases of IQ-Twemcached. Thus, this evaluation quantifies the number of anomalies eliminated by these synchronization primitives and their overhead.

Figure 8 shows the observed throughput with Redis and RangeQC for a YCSB workload consisting of a different mix of updates and scans. When scans constitute 10% of the workload, Redis outperforms RangeQC more than 6x because it implements the writes very efficiently using its in-memory data structure and one Redis SET. RangeQC performs four IQ-Twemcached operations along with the overhead of LeMInT lease manager for each write. The Redis implementation generates a few hundred anomalies in each experiment due to read-write race conditions. For example, with 90% scan, 300 reads (0.0044% of total reads) observe inconsistent data. Although the number of anomalies is small, our implementation of Redis must be extended with LeMInT and IQ leases of RangeQC to provide strong consistency.

With 100% scans, RangeQC outperforms Redis two folds because it uses two different nodes to process a predicate instead of one with Redis. With these experiments, the CPU of the cache server is the bottleneck resource dictating the overall throughput. With Redis, ZRANGE and multi-get method invocations are processed by the cache server. With RangeQC, the PrInT look up is performed using a Dendrite hosted on a server different than the one hosting IQ-Twemcached. Looking up the index from a Dendrite reduces the load imposed on the cache server, enabling RangeQC to outperform Redis.

With Redis, we considered an implementation using its Lua scripting [11] capability. With this implementation, the script (similar to stored procedure of a relational database management system) invokes the ZRANGE and multi-get methods, reducing the network round-trips times from the client to one. This implementation does not improve system throughput because the CPU of the cache server is the bottleneck resource.

[6]There are no replicas because there is only one cache server.

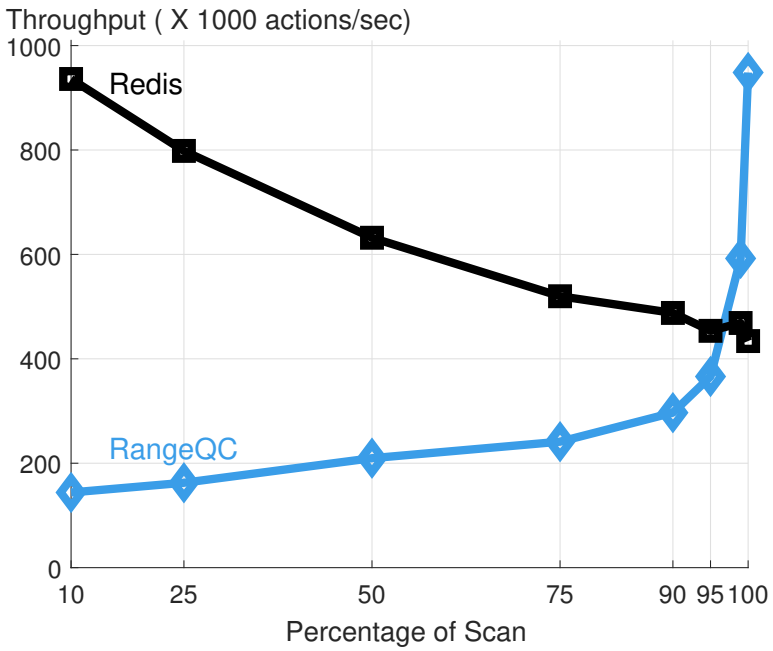


Figure 8: Redis vs. RangeQC with $\beta = 0\%$.

7.6 $\beta = 0\%$ versus $\beta = 100\%$

Cache entries of RangeQC of Section 7.5 are mutually exclusive as each represents one YCSB document (record). Hence, its $\beta = 0\%$. We used this implementation to compare with $\beta = 100\%$ assumed by Sections 7.1 to 7.4. In the following experiments MongoDB is not in the critical path of processing requests because we use the write-back policy and warm-up the cache to provide a 100% hit rate.

Figure 9 shows $\beta = 0\%$ provides a higher throughput than $\beta = 100\%$ with write-heavy workloads. An update impacts one document and each cache entry with $\beta = 100\%$ is five times larger than $\beta = 0\%$ since the result set contains five qualifying YCSB records. This slows down an update with $\beta = 100\%$ significantly because it must read and write[7] a cache entry that is five times larger than the one with $\beta = 0\%$. Hence, $\beta = 0\%$ outperforms $\beta = 100\%$ four folds. As we increase the frequency of reads, $\beta = 100\%$ starts to outperform $\beta = 0\%$. Its throughput is 40% higher with 99% read and almost two folds higher with 100% reads. With $\beta = 100\%$, reads observe a cache hit with one lookup for the predicate. With $\beta = 0\%$, reads must issue a multi-get for five keys. This multi-get is slower than one get with IQ-Twemcached, enabling $\beta = 100\%$ to outperform $\beta = 0\%$.

It is important to note that as we increase the cardinality of scan from 5 to 10 and beyond, $\beta = 0\%$ outperforms $\beta = 100\%$ by a wider margin with write-heavy workloads. Higher cardinality increases cache entry size for read-modify-write with $\beta = 100\%$. At the same time, $\beta = 0\%$ closes the gap with $\beta = 100\%$ for read-heavy workloads since the network of the cache server starts to become the bottleneck with higher cardinality.

[7]This read and write is a part of read-modify-write (RMW) to update one entry.

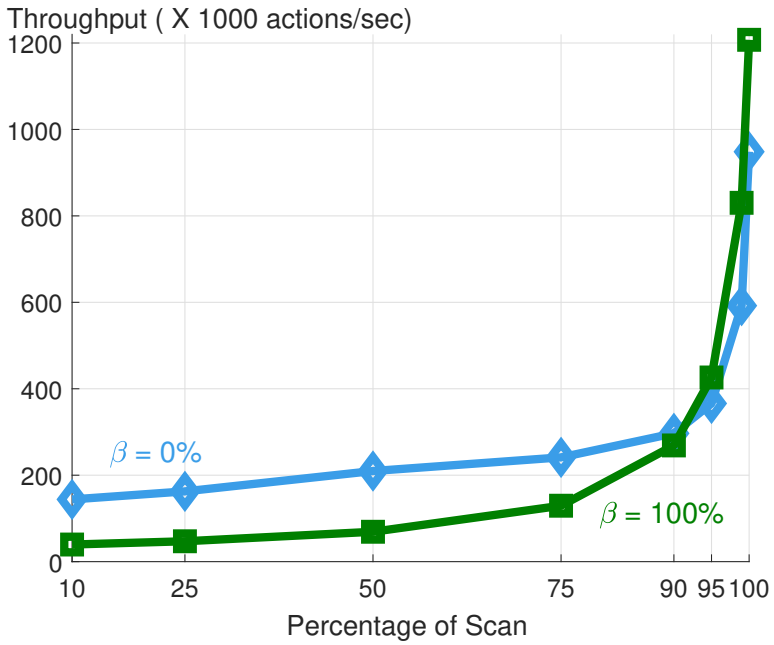


Figure 9: $\beta = 0\%$ vs. $\beta = 100\%$.

8 Discussion

Fusing multiple cache entries with $\beta < 100\%$: Section 7.6 highlights impact of β on RangeQC. Results presented in other sections change with β values less than 100%. For example, the cache hit rates of Table 3 increases to 100% with $\beta = 0\%$ by reducing each interval in PrInT to a point. This minimizes the number of cached result sets impacted by an insert, delete, and update.

Alternative architectures: Implementation of Section 3 is one possibility for use in a data center. There are other possible implementations for other applications and different environments. We discuss one below.

With Redis, one may implement PrInT, BuWrInT, and LeMInT using its Zindex: ZADD and ZRANGE methods. Their entries would be range partitioned across Redis CMIs similar to cached range predicates. This pushes functionality of a Dendrite into a Redis CMI, eliminating log records (because failure of a Dendrite and its corresponding CMI is the same). The buffered writes and its BuWrInT entries must be replicated 3 or more times to tolerate CMI failures. This can be realized using a backup Zindex of BuWrInT and buffered writes. With LeMInT, the entries would have a time to live consistent with the lease time. One must extend Redis with Inhibit and Quarantine leases to provide strong consistency.
Bulk loading: It is possible to load data in bulk using data store scripts. Subsequently, RangeQC would populate the Dendrites and CMIs. One may use a prefetching script to populate Dendrites and CMIs after loading data into the target data store. The effectiveness of such a script depends on how well it predicts future predicate references. If it is 100% accurate then the prefetched entries will be referenced at some point in the future. In the worst case scenario, the prefetched entries are neve accessed, rendering prefetching as an

overhead.

Conjunctive and disjunctive predicates: One may use RangeQC interfaces of Section 2 as a building block of more complex query processing. For example, a design may maintain a RangeQC instance on two different attributes of a document, say age and salary. Given a boolean conjunctive predicate such as "18 < age < 20 and 100K < salary < 200K", one may maintain statistics on the selectivity of each RangeQC instance. Assuming the "age" index is more selective then the design would process the predicate "18 < age < 20" to fetch the qualifying records. Subsequently, it would analyze each record to discard those entries that do not satisfy "100K < salary < 200K". With a disjunctive predicate, one would lookup the RangeQC instance for the age predicate and the RangeQC instance for the salary attribute. Their union with duplicate elimination produces the final result. Whether this form of processing is more beneficial that standard query processing using the data store depends on the selectivity of the predicates and the size of their result sets.

Cardinality of cache predicates: Section 7.3 highlights size of result sets as an important consideration for caching predicates. RangeQC may establish the threshold for this size by quantifying the service time of fetching a cache entry from the data store versus the cache during off-peak system loads. It may accumulate the different intervals issued without caching them. During low system loads, it may issue these predicates to the data store and cache their results, establishing their cardinality. It would fetch these cache entries and query the data store to build a histogram. Using this histogram, it would estimate a threshold on the cardinality of a predicate beyond which caching is not beneficial.

9 Related Work

Our implementation of RangeQC uses an interval tree [19, 28] to implement PrInT, BuWrInT, and LeMInT. This in-memory centralized index structure preserves order. Any order preserving indexing structure including a skip-list or a B+-tree could be used in its stead. Skip graphs [10] and its extensions [18, 9, 15, 14, 22] are decentralized data structures for processing range predicates in a *peer-to-peer* network of devices. Skip graphs extend the skip list by adding redundant connectivity, realizing a collection of up to D skip lists (assigned to different peers) that share some of their lower levels. They incur network traffic to maintain state in the presence of peers failing, leaving and joining the network. Its extensions [18, 9, 15, 14, 22] focus on data to balance load across peers, enhance path locality properties of the design, provide archiving of data, optimize energy usage, and consider non-cooperating peers.

This paper focuses on an implementation of RangeQC for use in a data center. It is novel because it maintains a cache of range predicate result sets computed using a data store. It maintains these entries consistent in the presence of updates, inserts, and deletes to the database. It implements alternative write policies, employs log records to tolerate failures, replicates buffered writes to enhance the likelihood of durable writes, controls placement of data such that a cache server failure loses both cached entries and their log records (due to use of volatile memory), and provides strong consistency. These concepts are absent from Skip graphs and their use case in peer-to-peer networks.

As discussed in Section 8, one may implement RangeQC for different applications. With

mobile environments and Internet of Things (IoT), RangeQC's client and coordinator may by re-designed by borrowing concepts from Skip graph and its extensions.

TxCache caches the result of a range predicate and invalidate it every time a DML command is issued to the table referenced by the predicate [26]. RangeQC invalidates (write-around) cached result sets every time a DML command impacts a row of its result set. In addition, RangeQC supports write-through and write-around policies.

MySQL InnoDB memcached plugin [2] enables an application program to access the InnoDB storage engine directly, bypassing the SQL layer to issue memcached commands such as get and set to a table. A recent addition is a range predicate ($@ \leq M$) as a memcached command. Alternative InnoDB caching policies include innodb_only, caching, and cache-only. The innodb_only policy passes information back and forth with InnoDB tables, bypassing memcached memory. The other two policies are designed to use memcached. However, in our experiments, we observed no performance difference between the alternative caching policies. The range result set is not cached in memcached when issuing a range predicate. We plan to investigate an implementation of RangeQC using MySQL.

Redis is an in-memory key-value store that processes range predicates [4, 5]. One may stage records referenced by different range predicates in a Redis cache server and issue the predicates for processing using Redis. This may substitute for the points and intervals maintained by PrInT with β=0%. However, it would still require BuWrInT to implement the alternative write policies, LeMInT and leases (Inhibit and Quarantine) for strong consistency.

10 Future Work

Our future research directions are two folds. First, we are investigating techniques that decide the optimal value of β dynamically. Section 7.6 highlights the tradeoffs between $\beta = 0\%$ and $\beta = 100\%$. A low β value reduces duplicated data in cache (maximize utilization of cache space), while a high β value creates coarser cache entries with duplicated data between two or more entries. A future work is to develop an algorithm to decide the optimal value of β between 0% and 100%. This algorithm may consider available cache space and network latency and bandwidth. The algorithm would dynamically adjust β to enhance metrics such as service time or throughput.

Second, we are evaluating alternative Dendrite implementations. For example, a Dendrite might be implemented in a CMI or a middleware. We speculate different implementations will provide different benefits for a given workload and application use case scenario. An understanding of the tradeoffs enables a system designer to select the most appropriate implementation for a given workload and application use case scenario.

References

[1] MongoDB. https://docs.mongodb.com/v3.4/core/ wiredtiger/#storage-wiredtiger-checkpoints (Accessed on 02/02/2018).

[2] InnoDB memcached Plugin, https://dev.mysql.com/doc/refman/8.0/en/ innodb-memcached.html (Accessed on 02/06/2018).

[3] zlib, https://docs.oracle.com/javase/7/docs/api/java/ util/zip/ Deflater.html (Accessed on 02/06/2018).

[4] ZRANGE, https://redis.io/commands/zrange (Accessed on 02/06/2018).

[5] Secondary Indexing with Redis, https://redis.io/topics/indexes (Accessed on 02/06/2018).

[6] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer: Auto-Sharding for Datacenter Applications. In *OSDI*, 2016.

[7] Y. Alabdulkarim, M. Almaymoni, and Ghandeharizadeh. Polygraph: A Plug-n-Play Framework to Quantify Anomalies. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018.

[8] J. Arulraj, J. J. Levandoski, U. F. Minhas, and P. Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. In *VLDB*, 2017.

[9] J. Aspnes, J. Kirsch, and A. Krishnamurthy. Load Balancing and Locality in Range-queriable Data Structures. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 115–124, 2004.

[10] J. Aspnes and G. Shah. Skip Graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 384–393, 2003.

[11] J. L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[12] D. Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Cloud Computing*, 2010.

[14] P. Desnoyers, D. Ganesan, H. Li, M. Li, and P. J. Shenoy. PRESTO: A Predictive Storage Architecture for Sensor Networks. In *Proceedings of HotOS'05: 10th Workshop on Hot Topics in Operating Systems, June 12-15, 2005, Santa Fe, New Mexico, USA*, 2005.

[15] P. Desnoyers, D. Ganesan, and P. J. Shenoy. TSAR: A Two Tier Sensor Storage Architecture using Interval Skip Graphs. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems, SenSys 2005, San Diego, California, USA, November 2-4, 2005*, pages 39–50, 2005.

[16] S. Ghandeharizadeh, J. Yap, and H. Nguyen. Strong Consistency in Cache Augmented SQL Systems. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 181–192, New York, NY, USA, 2014. ACM.

[17] S. Ghemawat and J. Dean. LevelDB, 2011, http://code.google.com/p/leveldb.

[18] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 9–9, 2003.

[19] H. V. Jagadish. On Indexing Line Segments. In *VLDB*, pages 614–625, 1990.

[20] N. B. Lamoureux, M.G. Deterministic Skip List Data Structures: Efficient Alternatives to Balanced Search Trees. In *Proceedings of the 19th Annual Mathematics and Computing Science Days (APICS)*, 1995.

[21] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *Trans. Storage*, 13(1):5:1–5:28, Mar. 2017.

[22] R. Morales, B. Cho, and I. Gupta. AVMEM - Availability-Aware Overlays for Management Operations in Non-cooperative Distributed Systems. In R. Cerqueira and R. H. Campbell, editors, *Middleware 2007, ACM/IFIP/USENIX 8th International Middleware Conference, Newport Beach, CA, USA, November 26-30, 2007, Proceedings*, volume 4834 of *Lecture Notes in Computer Science*, pages 266–286. Springer, 2007.

[23] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[24] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *Operating Systems Review*, 43(4):92–105, 2009.

[25] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[26] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional Consistency and Automatic Management in an Application Data Cache. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 279–292, 2010.

[27] A. Sainio. NVDIMM: Changes are Here So Whats Next. *In-Memory Computing Summit*, 2016.

[28] R. Sedgewick and K. Wayne. *Algorithms (Fourth edition deluxe)*. Addison-Wesley, 2016.

[29] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction processing. *PVLDB*, 8(3):245–256, 2014.

[30] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. volume 36, pages 255–270, New York, NY, USA, Dec. 2002. ACM.