Performance Evaluation of In-Memory Database with Cache

Anurag Chakraborty
Cheriton School of Computer Science
University of Waterloo
a8chakra@uwaterloo.ca

Jun Qing Lim
Cheriton School of Computer Science
University of Waterloo
jq3lim@uwaterloo.ca

***Abstract*—An in-memory cache is an intermediate data storage layer that usually resides between applications and databases and functions to provide fast responses for same queries over the data. In-memory databases are database management systems (DBMS) that rely on main memory instead of disk based persistent storage for saving user data. The later is optimized for faster read and write operations compared to traditional disk based DBMS due to its in memory based operations compared to slower disk based access operations. This document aims to compare the system performance of read-and-write operations in an in-memory database only environment and an in-memory database with cache environment. The purpose of this system evaluation is to determine whether an extra caching layer is really required in an architecture that takes advantage of an in memory database for faster querying of data. For test evaluation, *Apache Ignite (GridGain)* has been chosen as the in memory database and *Redis* has been chosen as the in memory caching system. The comparison has been made between two scenarios: in the first case only Apache Ignite has been used to serve user queries over the database stored in RAM, while in the second case, an additional Redis layer along with the database layer has been used. A number of parameters such as total number of rows, total number of queries and total number of threads being used have been varied across the test evaluation to determine the system performance. For evaluation of performance the following parameters are considered: latency, throughput, cache hit ratio (in the second scenario) with varying number of concurrent threads and total queries.**

***Index Terms*—in-memory database, in-memory cache, query processing, distributed database, cluster**

I. Introduction

In memory caches have been an existing solution for slow read write disk based databases for a long time. The traditional architecture for any large scale system involving serving user queries from the backend usually always involve an intermediate caching layer for faster response time and improving user experience. In memory caching can vastly improve the latency and improve online application performance which is a critical parameter for user engagement in today's systems. Studies have shown that 53% of users leave a site if it takes more than 3 seconds to load [1] indicating how critical speed is for good site engagement.

There have been other approaches to deal with slow access pattern of disk based databases such as the *anti-caching* approach [2] that uses eviction policies to determine which tables to push from the in memory storage to the disk storage when the in-memory storage of the database gets filled up. Since this approach has an eventual possibility of falling back to disk based storage, it has greater chance of degrading performance with increase in size of data. Distributed databases provide a suitable alternative to this approach since they allow for multiple systems of commodity hardware to be added in a cluster of nodes so that the risk of the main memory getting filled up is lower. This way the in-memory access can be maintained for a longer amount of time and with increase in size of data, more nodes with appropriate memory can be added eventually to the system easily. The overhead in this approach however is the network communication that is added for the nodes to communicate with each other for aggregation of intermediate or final results [10]. To reduce the latency of query serving due to this added overhead of communication over the network, an intermediate caching layer is a good approach since it reduces the chances of extra query execution for the same query from the user.

Apache Ignite is an in memory, horizontally scalable, distributed and fault tolerant database for high performance computing with in memory speed. The memory architecture of Ignite supports storing and processing data and indexes both in memory as well as on disk [11]. However for the purpose of this project, only the in memory architecture has been considered. It supports a wide range of developer APIs which have been used in this evaluation for native SQL queries. Similar to a disk based database, it allows creating a table, adding rows to this table, querying this database and then modifying this data with standard SQL data definition language (DDL), data query language (DQL) and data manipulation language (DML). Redis (*Re*mote *Di*ctionary *S*ervice) is an open source key-value in-memory database that is frequently used as a caching layer by many systems. Unlike traditional database systems, Redis allows users to model their data in the form data structures such as *string*, *bitmap*, *hash*, *list*, *map* etc. Redis also allows users to place eviction policies such as LRU (least recently used), maximum cache time-to-live (ttl) and cache size as part of the system configuration. Redis also supports a distributed architecture which has been leveraged in this system evaluation where multiple primary nodes and secondary nodes can be added as part of a wider cluster. This enables Redis to support a high availability (HA) feature in its clusters.

Redis does not support native querying with SQL commands, given it is a key value store. For caching the Apache Ignite query results which are composed of tabular results, the tables and rows have been remodeled as Redis data structures [5]. Primarily, the *Hash map* data structure have been used to remodel the tabular results. The approach has been explained in section II.

II. Design Scheme for Performance Evaluation

A. Overview of the Apache Ignite cluster

This section describes the cluster architecture for Apache Ignite. Figure 1 is an example architecture of an Ignite cluster. There are two types of nodes present in any Apache Ignite cluster, server nodes and client nodes. Server nodes participate in caching, query computation, processing of data, and other important tasks related to the actual execution of the queries. In other words, they are the base computational and storage unit in the cluster.

Client nodes act as intermediate nodes connecting the users to the server nodes. Usually the term of *Thick Clients* are reserved for these nodes which act as the intermediaries. They are connection endpoints and gateways from the user application to the main server nodes. There is another concept of *Thin Clients*, which have been used for this system evaluation project. Unlike Thick clients, Thin clients do not actually form the nodes in the actual Ignite cluster. They are lightweight clients that directly connect to the server nodes from the user application process.

The *advantage* of Thin clients over Thick clients is that since they do not exclusively take up an extra host as part of the cluster to launch a separate Client node, more hosts can be dedicated to launch server nodes for the cluster. The *disadvantage* of this approach however, is that Thin clients do not receive updates on the network topology of the Ignite cluster, meaning if a server node has left or a new node has joined the cluster. Due to this, there is more network communication since a Thin client has to first find out which specific server node owns a particular data set and then forward the request to that specific node. The decision to use Thin Client approach was based on the fact in this evaluation there were lower chances of network topology updates and the advantage of more scalability due to more server nodes.

Apache Ignite has two modes for data distribution in the cluster: *Partitioning* and *Replication*. In the partitioned mode, the data is the cluster is partitioned and split equally between all server nodes. In other words, the large sets of data is divided into smaller chunks and distributed among all the server nodes in a balanced manner. This mode is the more *scalable* distributed mode since it allows us to store as much data that can fit in the total memory available across all the nodes. In the replicated mode, all the data is replicated *at each and every node*. Meaning if there is a database of size 1GB then it will be copied across all the server nodes in the cluster. This mode provides more availability since the data is available on every node. However, every data update is propagated to every other node in the cluster that can performance to degrade. Also, the replicated mode is not suitable if the data is quite large since it can take a lot more of the total cluster memory due to such high replication. For the purpose of this system evaluation, *Partitioning* mode is used to distribute our data across the server nodes for better scalability.

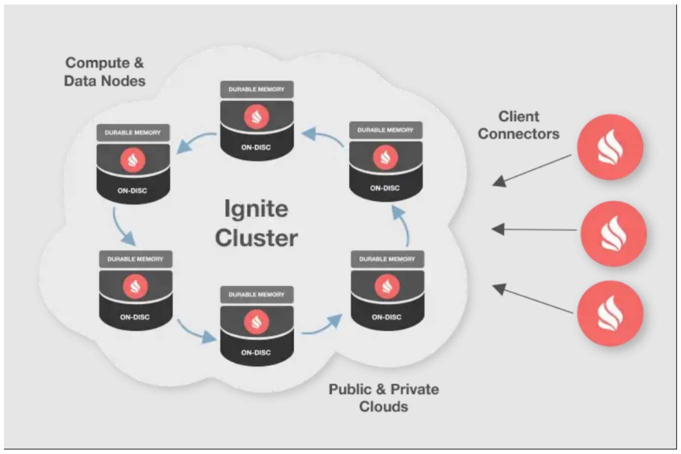

Fig. 1: Architecture of an Apache Ignite cluster [6]

B. Overview of the Redis cluster

This section describes the cluster architecture for Redis. Figure 2 is an example architecture of a Redis cluster. In the cluster mode, Redis spreads the data stored by the user across multiple machines using the sharding technique. Each Redis instance in a cluster holds a shard of the data as a whole. To determine which instance is holding the shard of a key, Redis uses algorithmic sharding to find the shard for a given key. A hash function is used to internally hash the key and generate a hash value. Then this value is *mod* against the total number of shards. To ensure a particular key is always mapped to the same shard, a deterministic hash function is used internally by Redis.

For this project's system evaluation, 3 primary nodes and 3 secondary nodes have been used in the cluster. Redis uses *asynchronous replication* [12] to replicate the updates between the primary and secondary nodes. Every primary instance of Redis has a *replication ID* and an *offset*. The offset is a count for every action that Redis performs on any primary. Meaning the offset gets incremented for every operation that Redis performs on the primary node. When the Redis secondary instance is a few offsets behind the primary instance, it receives the remaining operations to perform from the primary. The secondary node will replay these operations on the same replicated data until it is synced with the primary. The replication ID is a placeholder for cases where some primary node leaves the cluster and a secondary node is upgraded to primary status. In this case the secondary gets a new replication ID, and has to do a full re-sync with other secondary nodes that do not match with the new ID. Internally, Redis uses the Gossip protocol to communicate among the nodes in the cluster. If a shard is available to serve requests then the request gets forwarded to it. All *read* requests to a Redis cluster can be served by *both* the primary and the secondary nodes in the cluster, but the *update* requests can only be served by the primary nodes of a cluster.

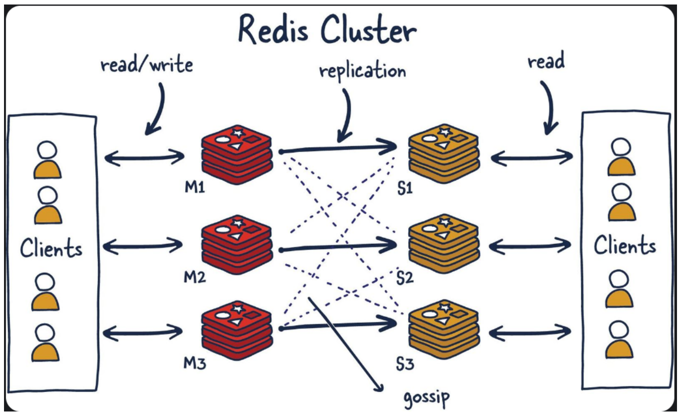

Fig. 2: Architecture of a Redis cluster [7]

For the purpose of this system evaluation, we have enabled *diskless replication* in our Redis cluster. Redis supports disk based mode of persistence also, but since the aim of this project is system evaluation only for in-memory mode, it was not enabled. This gives our cluster much more speed and scalability for higher size of datasets but lower durability due to lack of persistence. With diskless replication, the primary instance directly sends the data over the network to the secondary nodes hosting the replicas, without using disk as an intermediate storage. The $repl-diskless-sync$ configuration parameter is used to enable this mode in the cluster and the $repl-diskless-sync-delay$ parameter is used to control the delay between an operation and replication trigger between the nodes. The value for this delay was set as *5 second* in our cluster. The $maxmemory$ configuration sets the upper value for the Redis cluster, we set this value as (1/10)th of the total size of the particular dataset we're querying against. For example for our 500,000 rows dataset which was approximately 100 Megabytes (MB) in size, we set the cache size as 10 MB for the cluster. The key eviction policy used in the configuration was $allkeys-lru$ which uses LRU policy to remove keys from the cache.

As mentioned in the previous section, Redis does not natively support querying over keys in SQL format. Due to this reason, the query result sets received from Apache Ignite (in tabular format) had to be converted in a key-value format into Redis data structures. The Hash data structure in Redis that stores key value pairs for a particular hash key. The hash value is thus associated with a set of field-value pairs provided by the user. This data structure is thus ideal for storing and retrieving the records of a relational table. The $HMSET$ command is used to store hash keys and their corresponding field-values. For example, the SQL query to insert a row in a database table and the corresponding Redis command are as follows:

```
INSERT into Table(id, col1, col2, col3)
VALUES(1, 'ZXYW', 'ABCD', 300)

HMSET Table:1
      col1  'ZXYW'
      col2  'ABCD'
      col3  300
```

The SQL command is inserting into the table *Table* for the columns *id*, *col1*, *col2* and *col3* the values *1, 'ZXYW', 'ABCD'* and *300*. For Redis, we create a *hash key* combining the table name and the primary key $< table\_name >:< id >$, so it is *Table:1*. Then, every column (attribute) of the table is made into a field and the attribute value for the row is the corresponding value. So for the *col1* field, *1* is the value, and so on. Next, we look at an SQL query that looks up rows for a particular id and the corresponding Redis query which uses the $HGETALL$ command:

```
SELECT * from Table WHERE id = 1

HGETALL Table:1
```

In this query, all the attributes for the row that has id equal to 1 is looked up from the table. In Redis, $HGETALL$ followed by the hash key which in case will be $< table\_name >:< id >$ so *Table:1* is given, which returns all the field value pairs. To get a specific attribute value instead of all columns, the $HGET$ command is used followed by the field required.

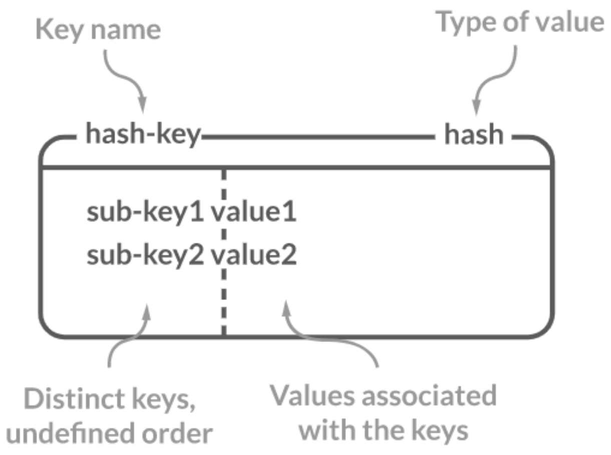

Fig. 3: Hash Data Structure in Redis [8]

In this manner, all SQL queries received by our test set up first parses and converts the query into the equivalent Redis query and makes a cache lookup. If the value is not found from the cache, then the SQL query is executed on Ignite.

The result set received is then inserted into Redis row by row using $HMSET$. The other choice for caching data in Redis would be to directly cache the SQL query as key and the result set as the value. The disadvantage of this approach is that if some query's response is a subset of another query then even for that query the execution will be repeated, since the cache lookup will be done on the basis of the SQL query as the key.

III. Evaluation Procedure

This section discusses the procedures/steps involved in evaluating the performance.

A. Hardware Specification

The following table I illustrates the specification of the machine that will be used for our evaluation.

TABLE I: Hardware Specification

Aspect	Description
System Memory	64GB
No. of CPUs	6
No. of Thread per CPU	1
Processor	Intel(R) Xeon(R) CPU E5-2609 v3 @ 1.90GHz
Architecture	x86 64

Note: the command lshw and lscpu are used to obtain the hardware specifications of the machine

B. Setup Configurations

The Ignite cluster consists of six servers. Each server has the hardware specification illustrated above and a cluster is formed using TCP/IP Discovery Mechanism [3], where each server node knows the IP addresses of all other servers to discover each other in forming a cluster as per Figure 4.

```
<property name="discoverySpi">
<bean class="...tcp.TcpDiscoverySpi">
<property name="ipFinder">
 <bean class="...TcpDiscoveryVmIpFinder">
  <property name="addresses">
   <list>
    <value>Server 1's IP</value>
    <value>Server 2's IP</value>
    ...
   </list>
  </property>
 </bean>
</property>
</bean>
</property>
```

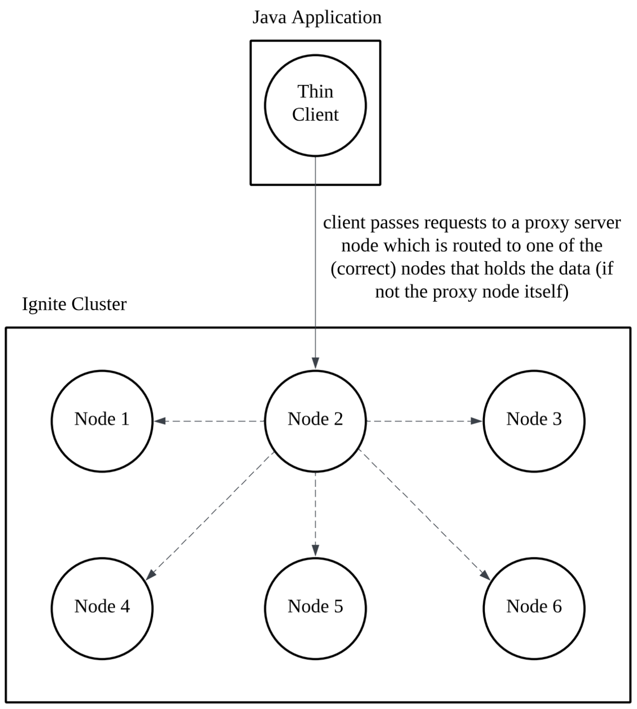
Java Application
Thin Client
client passes requests to a proxy server node which is routed to one of the (correct) nodes that holds the data (if not the proxy node itself)
Ignite Cluster
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6

Fig. 4: Apache Ignite Thin Client Setup [9]

The Redis cluster (3 primary nodes, 3 secondary nodes) is launched from the command line with the $redis-cli$ utility and the $redis.conf$ file having the cluster configuration as follows:

```
bind <Each Node's IP Address>
protected-mode no
port 7000
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 15000
appendonly no
save ''''
daemonize yes
maxmemory <0.1 times the size of dataset>
repl-diskless-sync yes
repl-diskless-sync-delay 5
...
```

The Redis process is launched as a daemon process initially for all the nodes in the cluster and then using $redis-cli$ from any of the nodes with all the node IP addresses specified, the cluster gets started. The $appendonly$ feature of Redis is disabled to use it purely as a cache, and this instructs Redis to not persist any of the data. Once the cluster is restarted, all of the data gets removed. The $save''''$ instructs Redis to disable RDB (Redis database) snapshotting completely so no persistence is ensured.

A Java Thin Client will then be used to interact with the Ignite cluster as illustrated in Figure 4, while for the with-cache scenario an additional request to check with Redis will be made.

C. Dataset Specification

This section discusses the specifications of the dataset used in the evaluation and how they were generated. *Sysbench* was the tool primarily used for generating the SQL workloads [4]. The *oltp* option in Sysbench generates workloads (both datasets and queries) depending on the *read* or *read-write* option passed. First the test table with below schema is created in Apache Ignite:

```
CREATE TABLE `table_<TOTAL_ROWS>` (
`id` int(10) unsigned NOT NULL,
`k` int(10) unsigned NOT NULL,
`c` char(120) NOT NULL,
`pad` char(60) NOT NULL,
PRIMARY KEY (`id`),
KEY `k` (`k`));
```

There are 4 columns in the table, the first column id is the primary key and the second column k has an additional secondary index built for it. The secondary index is created for testing concurrent update & read performance for this attribute. For the read-only query workload, any of the following SQL queries may be executed:

Point queries

```
SELECT c FROM table_<total_rows>
WHERE id=N
```

Range queries

```
SELECT c FROM table_<total_rows>
WHERE id BETWEEN N AND M
```

Range $SUM(k)$ queries

```
SELECT SUM(K) FROM table_<total_rows>
WHERE id BETWEEN N and M
```

Range order by queries

```
SELECT c FROM table_<total_rows>
WHERE id between N and M ORDER BY c
```

Range distinct queries

```
SELECT DISTINCT c FROM table_<total_rows>
WHERE id BETWEEN N and M ORDER BY c
```

In the read-write workload, along with the above mentioned read based queries the following update / insert queries maybe present:

Updates on index column

```
UPDATE table_<total_rows> SET k=k+1
WHERE id=N
```

Updates on non − index column

```
UPDATE table_<total_rows> SET c=N
WHERE id=M
```

Update for range on index column

```
UPDATE table_<total_rows> SET k=k+1
WHERE id BETWEEN N AND M
```

Updates for range on non − index column

```
UPDATE table_<total_rows> SET c=N
WHERE id BETWEEN N AND M
```

Insert queries

```
INSERT INTO table_<total_rows> VALUES (...)
```

D. Multi-threading Configurations

Each set of query execution is executed with 5, 10, 15, 20, 25, 30 threads where the number queries are distributed among the threads. As Apache's Ignite JDBC object of $Connection$ is not thread-safe, thus the number of JDBC opening connections will be set to the number of threads that are running. That is, each thread will hold one unique JDBC $Connection$.

Algorithm 1 Thread Work Execution

```
minLatency = MAX_VALUE, maxLatency = MIN_VALUE
totalSucceed = 0
threadId ← id of the current thread
totalThreads ← total number of threads
queries ← list of queries to be executed
for i = threadId  i < size of queries  i += totalThreads do
    startTime ← get epoch current time
    query ← get the queries of index i
    if query contains "SELECT" then
        cached = executeQuery(query)
        if cached then
            cacheHit++
        end if
    else
        executeUpdate(query, redis)
    end if
    if query succeeded then
        totalSucceed++
    else
        totalFailed++
    end if
    endTime ← get the epoch current time
    latency ← endTime - startTime
    minLatency = Min(minLatency, latency)
    maxLatency = Max(maxLatency, latency)
    totalLatency += latency
end for
avgLatency ← totalLatency / totalSucceed
return minLatency, maxLatency, avgLatency, totalSucceed,
cacheHit
```

Algorithm 1 illustrates the work executed by one thread. Each set will be executing a number of queries that are distributed across the total number of threads in the pool. If the SQL query to be executed contains "SELECT", we execute the query accordingly from cached result, then we increase the cache hit count to keep track of. Otherwise, the query is a non-SELECT statement (i.e., INSERT, UPDATE, etc.), and cache hit count is not incremented here because update queries are not cached. The minimum latency, maximum latency, average latency of the queries, the total number of queries succeeded and cache hit count were then returned as part of the thread work's response.

E. Main Algorithm for Database-only Environment

Algorithm 2 Algorithm for database-only execution

```
function EXECUTEQUERY(sql)
    Statement stmt = conn.createStatement()
    stmt.executeQuery(sql)
end function

function EXECUTEUPDATE(sql)
    Statement stmt = conn.createStatement()
    stmt.executeUpdate(sql)
end function
```

The thread work execution above is executed for database-only environment with the Algorithm 2 above for executing the query and update.

F. Main Algorithm for Database-With-Cache Environment

Algorithm 3 Algorithm for database-with-cache execution

```
function EXECUTEQUERY(sql)
    if redis.checkRedis(sql) then
        return true
    else
        Statement stmt = conn.createStatement()
        ResultSet rs = stmt.executeQuery(sql)
        redis.putRedis(rs, sql, conn)
        rs.close()
        return false
    end if
end function

function EXECUTEUPDATE(sql)
    Statement stmt = conn.createStatement()
    stmt.executeUpdate(sql)
end function
```

Similar to Algorithm 2, the above Algorithm 3 $executeQuery$ function is modified to access cached data if present or query from database otherwise. The $checkRedis$ function parses and converts any SQL query to its equivalent Redis cache lookup query using either the $HGET$ or $HMGET$ redis function:

```
SELECT c, pad FROM <table_name>
WHERE id=%d
```

```
HMGET table_name:primary_key c pad
```

For $sum(K)$ queries, a unique key was created with the range specified in the query to lookup from the cache if previously the aggregation has been computed:

```
SELECT SUM(k) FROM table_name
WHERE id BETWEEN N and M

GET table_name:sumK:N:M
```

For range fetch of a particular attribute, the lower bound and upper bound for the SQL query would be parsed and identified. Then $HMGET$ would be used for each key in the form of $HMGET$ $table\_name : primary\_key$ to fetch the particular attribute required.

In case of a write query (update / insert) the $checkRedis$ function is bypassed since it is a cache miss. The read query could also be a cache miss. In both the cases, we proceed to executing the query on Ignite. The $ResultSet$ received is iterated over and the cache is updated using the $HSET$ and $HSETNX$ (for single attribute update):

```
SELECT c, pad FROM table_name
WHERE id=%d

HSET table_name:id c <value_of_c>
 pad <value_of_pad>
```

For update query, the corresponding Redis query would be as follows:

```
UPDATE table_name SET c='%s' WHERE id=%d

HSETNX table_name:id c <value_of_c>
```

G. Overall Algorithm

Algorithm 4 Algorithm for Full-Simulation

```
connectionUrl ← set up JDBC ignite cluster url string
conn ← DriverManager.getConnection(connectionUrl)
threads ← [5,10,15,20,25,30]
queryFiles ← get the list of query file names
for each queryFile do
    queries ← get the list of queries from queryFile
    for each threadSize do
        createTable(...)
        insertData(...)
        executorService ← create pool of threadSize
        startTime ← record the current time
        execute all threads and wait for completion
        endTime ← record the current time
        compute min_latency, max_latency, avg_latency, total_time, events_per_sec
        dropTable(...)
    end for
end for
```

Algorithm 4 illustrates the full simulation of all the threads. For every table size and set of queries, it is run against the full simulation and the results are then gathered and discussed in the following section IV.

IV. Evaluation & Result Discussion

There are three main metrics for which the performance evaluation has been performed: throughput measured in terms of *events per second*, latency measured in terms of round trip time to service a request by the system, and finally the cache hit ratio (in the case where the Redis cache exists above the Ignite database).

A. CPU Throughput Graphs

Each query executed by the system is recorded as an *event* and we count the total number of such events executed for every evaluation run. The total end to end time for the execution for all the queries is also tracked and finally total events divided by the total time gives an effective measure of the Throughput.

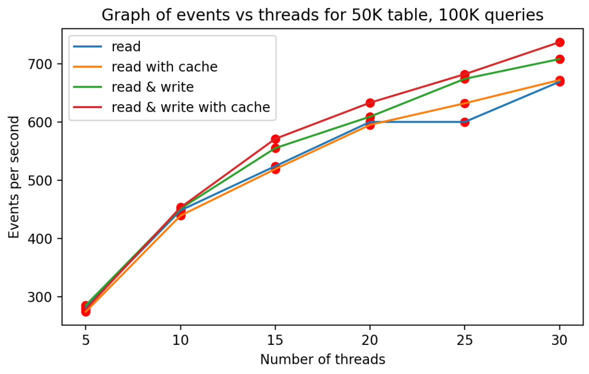

Fig. 5: CPU Throughput for 50k rows and 100k queries

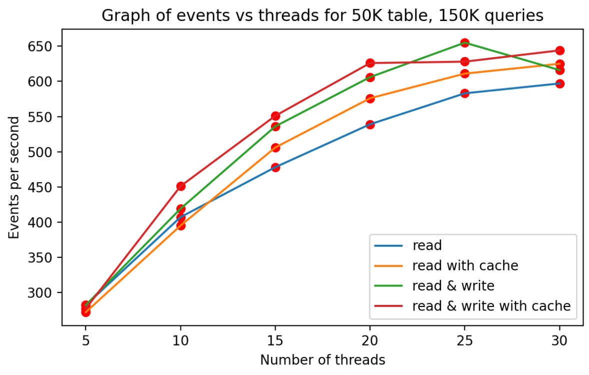

Fig. 6: CPU Throughput for 50k rows and 150k queries

The above two figures, Figure 5 and 6 measure the throughput for a dataset of 50k rows and two sets of workloads: 100k queries and 150k queries. For each set of query workload (50k and 100k) there is a read only workload and a read-write workload with both read and write queries. For 100k queries and low concurrency (5, 10 threads) the throughput more or less remains constant with cache and without cache. The difference is more noticeable with high concurrency (25, 30 threads) where the throughput is more with cache (red & yellow line) compared to the set up without cache.

For 150k queries the throughput for read-write workload is always more with cache compared to without cache (except for 25 threads setup). Read-only with cache always has more throughput compared to without cache for 50k rows and 150k queries.

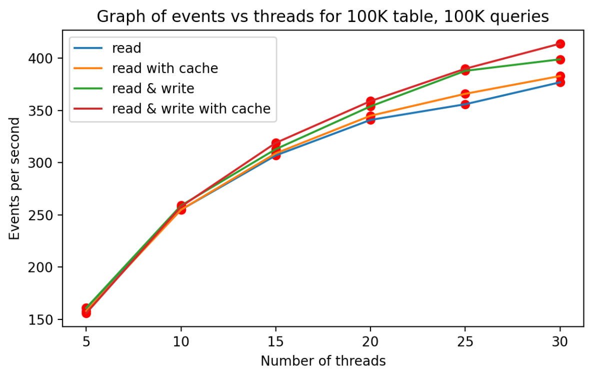

Fig. 7: CPU Throughput for 100k rows and 100k queries

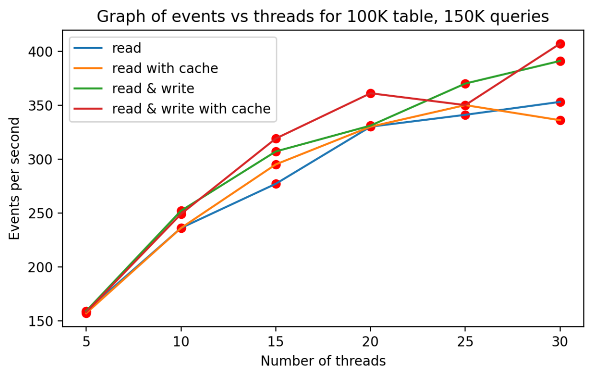

Fig. 8: CPU Throughput for 100k rows and 150k queries

Figure 7 and 8 measure throughput for 100k table with total 100k & 150k queries. 200k queries was also benchmarked but this workload was either timing out (execution not ending with 100% CPU utilization for extended time) or the cluster was running out of memory (throwing OutOfMemory Java exception). The throughput upper limit is considerably lower compared to 50k rows (reaching 650-700 events per sec) while in this case it is around 400 events per sec.

For 100k queries and 100k rows, there is not much throughput difference between without cache v/s with cache scenario, with the difference only noticeable at 30 threads for read-write workload and at 20 threads for read-only workload.

For 150k queries, throughput for read-write with cache is always more compared to without cache (25 threads set up is the exception). Read-only throughput is mostly more with cache except for 30 threads v/s without cache setup.

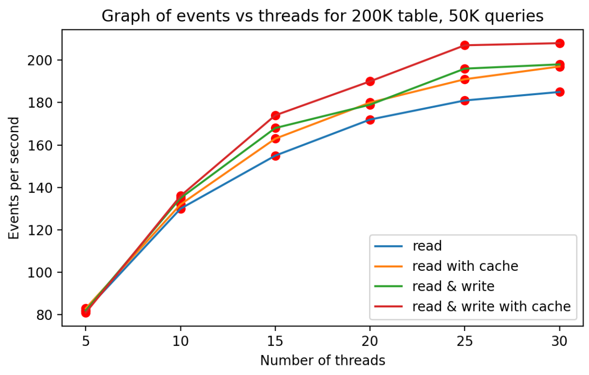

Fig. 9: CPU Throughput for 200k rows and 50k queries

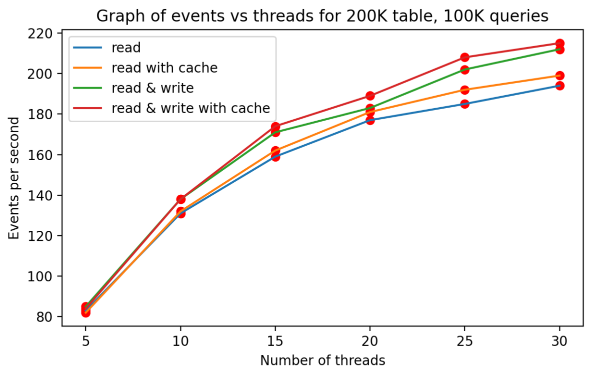

Fig. 10: CPU Throughput for 200k rows and 100k queries

Figure 9 and 10 measure throughput for 200k rows with 50k queries & 100k queries. The upper limit for the throughput is almost halved with this set up compared to 100k rows where we noticed around 400 events per sec. With 200k rows, even though query workload comprised of much lesser queries the limit was reached around 200-220 events per sec. For >100k queries workload we faced the same issues of non termination of workload while benchmarking with either 100% CPU utilization for extended time or cluster running out of memory. 50k queries workload has consistently the with-cache setup having more throughput compared to without cache. Same goes for 100k queries workload.

Figure 11 and 12 measure throughput for 500k rows with 25k queries and 50k queries. 500k rows table for the database was observed to be the limit for the Ignite cluster, 1 million and 2 million rows was also tested but there were either issues with loading the table into the database or the query workload had to be significantly reduced (less than 1000 queries). Increasing concurrency did not help with the issue, so 500k rows was taken as the final table size to be tested for this system evaluation.

From Figure 11, with 25k queries the observed behavior till now gets reversed. For read-only workload it is more noticeable where the throughput without cache is greater compared to the with-cache setup, only with 30 threads they get closer. In case of the read-write workload, the with-cache setup has either lesser throughput or comparable throughput to the without-cache setup. This behavior is most likely an affect of lower number of cache hits due to much lesser total number of queries. Sysbench generates workloads with uniform distribution meaning a mostly random access pattern, with less number of queries thus guaranteeing that there will be lesser cache hit ratio.

For 50k queries (Figure 12), we return to the previous behavior of the with-cache scenario where it has more throughput compared to the without-cache setup. With more queries, there was greater chance of cache hit ratio due to this the first setup performed better here compared to without having a cache.

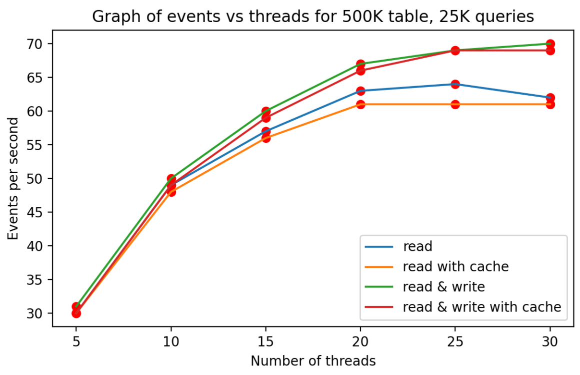

Fig. 11: CPU Throughput for 500k rows and 25k queries

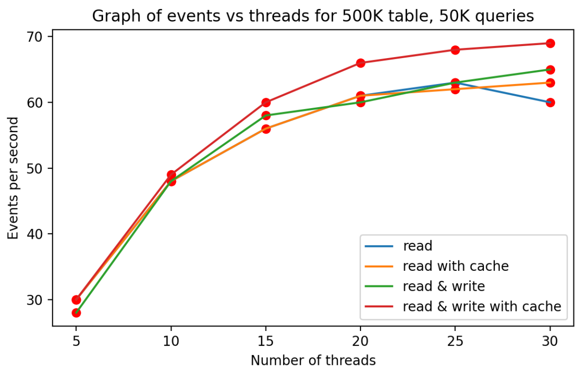

Fig. 12: CPU Throughput for 500k rows and 50k queries

B. Latency Graphs

The Latency of an event was measured as the round-trip-time taken to serve a query request. The average latency was calculated after aggregating the latency tracked for all the events and then dividing them by the total number of queries / events in the workload. In general for all tables and query workloads, with increased concurrency the latency increase was observed (along with increase in throughput from previous section).

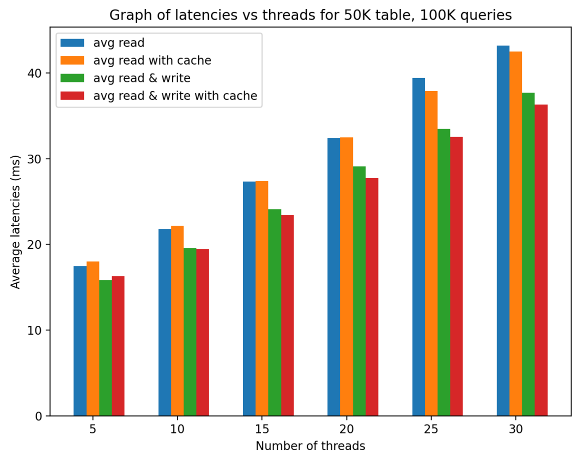

Fig. 13: Average Latency for 50k rows and 100k queries

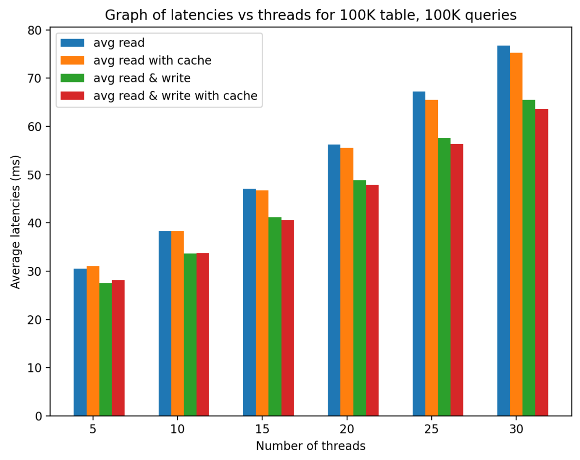

Fig. 15: Average Latency for 100k rows and 100k queries

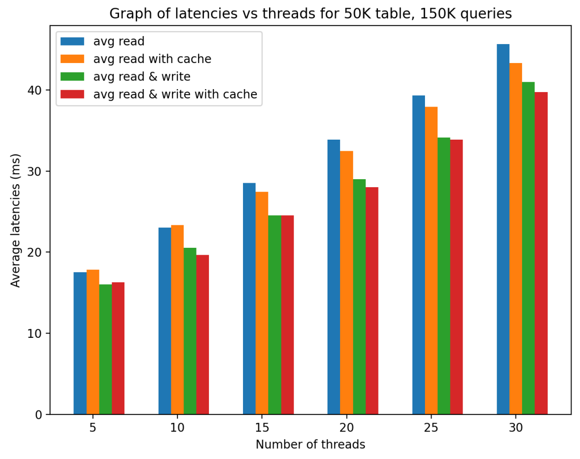

Fig. 14: Average Latency for 50k rows and 150k queries

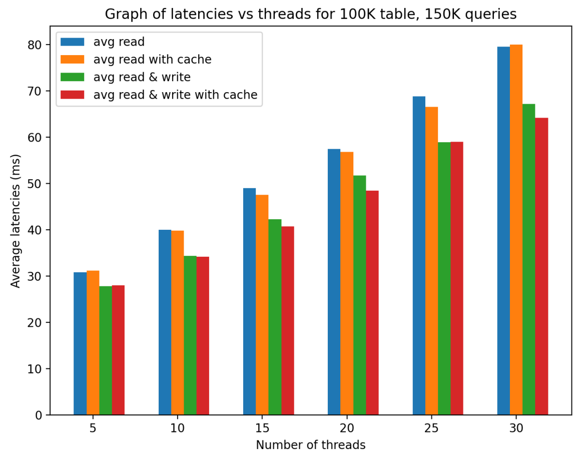

Fig. 16: Average Latency for 100k rows and 150k queries

Figure 13 and 14 display average latency for 50k total rows table with workload of 100k & 150k queries respectively. With low concurrency, the with-cache setup has higher average latency compared to without cache. As the concurrency is increased, the cache environment helps to reduce the latency. For 30 threads the difference between the two setups is noticeable, where the cache setup outperforms the without cache set up for both the read-only and read-write query workloads.

Figure 15 and 16 display display latency for 100k rows with 100k queries and 150k queries workload. Compared to the 50k rows setup, the average latency for this workload is almost double (∼40ms with max concurrency for 50k and ∼80ms for 100k). Similar observation as the previous setup of 50k, the latency with cache is more or at least the same as without cache setup. When the concurrency is increased however, the with cache setup displays lesser latency. The only exception to this is in Figure 16 where for 30 threads, the average latency with cache for read-only workload is slightly greater than without cache setup.

For 200k rows workload with 50k queries & 100k queries (Figure 17 and Figure 18), the latency is considerably higher for the without cache setup compared to with cache. The average latency has also doubled here compared to the 100k rows workload (∼160ms for highest concurrency). For 5 or 10 threads, there is not much observable latency difference between the two setups, and similar to the other workloads becomes more prominent for greater number of threads.

For the final set of workloads of 500k rows and 25k, 50k queries (Figure 19 and 20) the latency with cache was higher in some cases compared to without cache. With 30 threads, the 25k queries workload displays more latency with cache for both the read-only and the read-write workload. The same is observed in the 50k queries workload where the read-only workload has more average latency (for read-write case the latencies are almost comparable so not much difference).

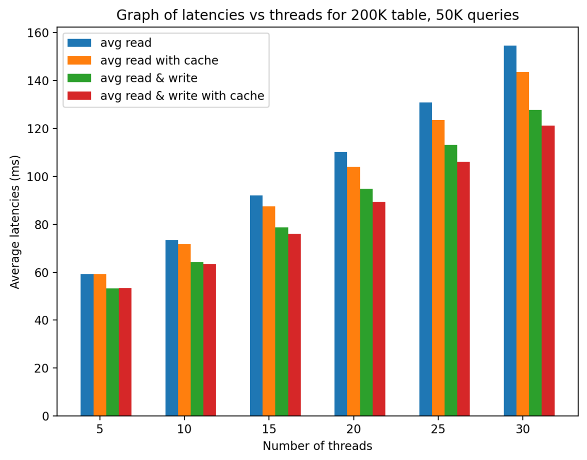

Fig. 17: Average Latency for 200k rows and 50k queries

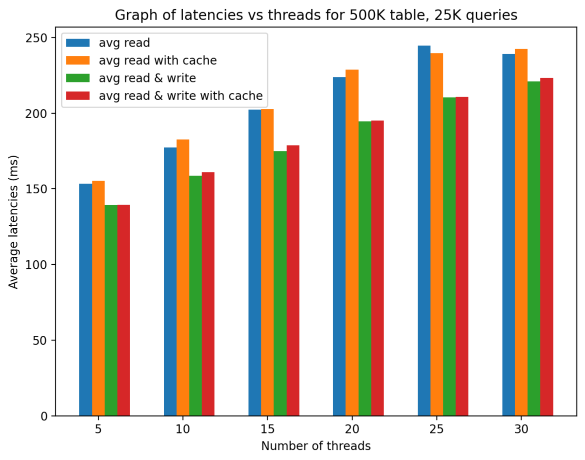

Fig. 19: Average Latency for 500k rows and 25k queries

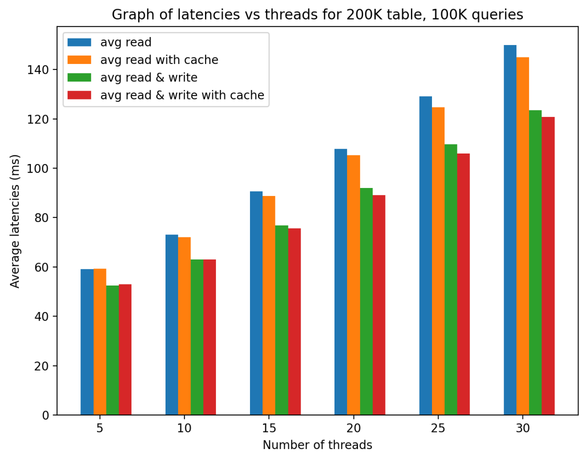

Fig. 18: Average Latency for 200k rows and 100k queries

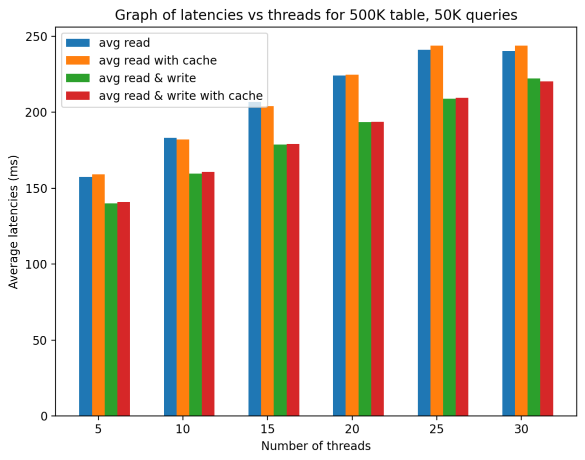

Fig. 20: Average Latency for 500k rows and 50k queries

With 50k queries (Figure 20) in the read-only workload, there is clearly more average latency in the highest concurrency (30 threads) case. The latency remains more or less remains the same with read-write workload across different concurrency in this evaluation setup.

C. Cache Hit Ratio

This section highlights the cache hit ratio with varying total rows in the table and total number of queries run. This metric was measured in the second setup (with Redis cache). If the cache lookup before query execution is successful, the cache hit counter got incremented. The final ratio was obtained by dividing the total number of cache hits by the total number of queries run.

Figure 21 (next page) highlights the cache hit ratio with increasing total number of queries. There is very high cache hit ratio increase with more number of queries being run. With more queries getting run, there was a greater chance of the query accessing a previous row already accessed from a previous query. Also the higher total number of queries workloads (100k, 150k queries) were only run for lower total number of rows (50k, 100k rows) meaning the total query space for rows to access was much lesser. Better cache performance is thus observed for more queries run and smaller size of database table.

Figure 22 highlights the cache hit ratio with increasing total rows in the table. The cache hit considerably decreases with this increase given that the total number of queries for the set of rows was also lesser.

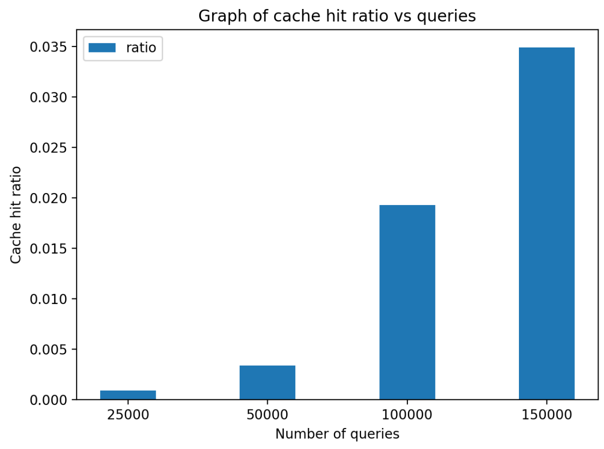

Fig. 21: Cache Hit Ratio v/s Total Queries

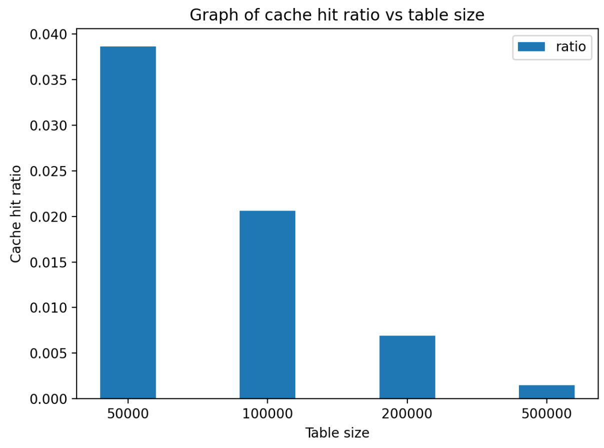

Fig. 22: Cache Hit Ratio v/s Total Rows

V. Conclusion

While there are certain edge cases, in general the expected behavior of better throughput with a cache layer in front of the database setup was observed. The throughput was almost always better in the with-cache environment compared to without-cache, especially with higher concurrency (25, 30 threads). However the upper threshold for throughput was observed to be reached around 30 threads after which the throughput generally dropped with more threads. This is consistent with Amdahl's Law which places a theoretical upper threshold for maximum speed up possible for a workload with increasing concurrency. Better performance was also observed in all cases for the read-write workload compared to the read-only workload, across both the throughput and latency graphs. This indicates that Apache Ignite might not be as suitable for read heavy workloads (OLAP).

The throughput gain with an extra Redis layer however is not prominent in many cases such as when the total threads were lower where the with-cache and without-cache setups almost had the same throughput. Also the average latency for requests was greater with higher concurrency, with latency consistently increasing for greater table size. If more threads are chosen then definitely more throughput is achieved but with the extra overhead of greater average latency. For latency, the cache is causing performance to degrade in some cases since more average latency is present (for example in the 500k total rows case).

Thus, it is difficult to make a sweeping generalization that a cache layer (Redis) is truly giving a performance boost over querying & accessing data directly from an in-memory database (Apache Ignite). Our evaluation numbers suggest that it is highly workload dependent, and the slight boost in one parameter (such as throughput increase) may cause greater performance degradation in the latency of the query requests. Instead of aiming for highest concurrency with highest throughput but lowest average latency, a more balanced concurrency level with similar throughput levels and better latency is a better alternative.

If the usecase is more latency sensitive, then having a cache maybe beneficient since the latency graphs mostly displayed lower latency in the with-cache setup compared to without-cache. If more throughput is the aim, then keeping a cache layer might not be as beneficient and maximum concurrency possible (until hitting the upper throughput threshold) should be the aim.

References

[1] https://www.blog.google/products/admanager/increase-speed-of-your-mobile-site-wi/

[2] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik, "Anti-caching: A new approach to database management system architecture," Proc. VLDB Endowment, vol. 6, pp. 1942–1953, 2013.

[3] https://ignite.apache.org/docs/latest/clustering/tcp-ip-discovery#multicast-ip-finder

[4] https://github.com/akopytov/sysbench

[5] https://redis.com/blog/get-sql-like-experience-redis/

[6] https://ignite.apache.org/docs/latest/memory-architecture

[7] https://architecturenotes.co/redis/

[8] https://redis.com/ebook/part-1-getting-started/chapter-1-getting-to-know-redis/1-2-what-redis-data-structures-look-like/1-2-4-hashes-in-redis/

[9] https://ignite.apache.org/docs/latest/thin-clients/getting-started-with-thin-clients

[10] H. Zhang, G. Chen, B. C. Ooi, K. -L. Tan and M. Zhang, "In-Memory Big Data Management and Processing: A Survey," in IEEE Transactions on Knowledge and Data Engineering, vol. 27, no. 7, pp. 1920-1948, 1 July 2015, doi: 10.1109/TKDE.2015.2427795

[11] https://ignite.apache.org/docs/latest/memory-architecture

[12] https://redis.io/docs/management/replication/