KELI: A Key-value-with-Links In-memory Store for Realtime Applications

Minh Duc Le*, The De Vu*, Duc Hieu Nguyen*, Tien Hai Ho*, Duc Hai Nguyen*, and Tran Vu Pham
Faculty Computer Science and Engineering
Ho Chi Minh City University of Technology - VNU-HCM
Ho Chi Minh City, Vietnam
contact email: {haind, ptvu}@cse.hcmut.edu.vn

ABSTRACT
The emergence of advanced network technologies and decreasing DRAM costs have provided great opportunities for data centers to deploy high performance key-value in-memory stores for real-time applications. Although key-value model helps in-memory stores to simplify their query processing and data management mechanisms, its simplicity have applications to generate multiple item lookups to perform simple requests such as constructing HTML files. Such lookups are often executed sequentially and seriously affect the overall performance. In this paper, we introduce KELI, a novel in-memory key-value store that employs Key-value-with-Links, a simple but quite efficient data model, to solve this issue. On the top of the key-value model, KELI establishes links between items so instead of invoking expensive item lookups, it could simply follow those links to retrieve needed data. KELI also uses RDMA Read to implement link chasing to minimize searching overhead. Some experiments have been conducted to examine the performance of KELI. Their results indicate that KELI runs three times faster than Redis for data lookups. Furthermore, KELI runs seven times faster than Redis when handling real workloads generated by a practical application.

CCS Concepts
• Information systems → Database design and models; Data structures;

Keywords
in-Memory Stores; Key-Value; RDMA; InfiniBand

1. INTRODUCTION
The emergence of Big Data era and real-time Web-based applications makes reducing the latency of processing data requests become crucial. In order to provide services that could manipulate a huge amount of data within tight deadlines, using traditional (i.e. on-disk) databases is inappropriate due to the limitation in the speed of secondary storages. Moving data to main memory (e.g. RAM) is a common approach. In-memory stores have become important parts of many applications including social networking [11], big data analytics, and high performance computing [7].

Because of strict requirements in latency and throughput, most of in-memory stores organize their data by using key-value model to avoid expensive operations such as join or sort, which are popular in traditional (e.g. relational) models. However, the simplicity of key-value model has applications to decompose the original request into a lot of items lookups. For example, in order to construct an HTML file, Facebook has to create about 130 data lookups in average [13]. Similarly, Amazon requires about 100 to 200 internal requests for each of its pages [4]. As those requests are often executed sequentially, their cumulative latency is the major factor that limits the overall performance.

Since most of in-memory stores are deployed over distributed environment to handle big datasets, communication is another problem affecting their performance. Different from High Performance Computing (HPC) centers, most of data centers use traditional interconnection technologies (i.e. Ethernet) to connect data nodes due to their low prices. This prevents in-memory stores from reaching their full potential because even though DRAM is extremely fast, the slow speed of data transmission makes applications to spend a lot of time waiting for the results. To tackle this issue, some data centers have begun to employ high performance network technologies such as RDMA and InfiniBand. The emergence of these technologies is a great opportunity for us to build ultra-low latency services.

In this article, we introduce KELI (KEy-value-with-Links In-memory store), an in-memory cache that relies on state-of-the-art network technologies to offer low latency caching services for real-time applications. KELI employs a new data model, named Key-Value-with-Links, to manage data. Using Key-Value-with-Links model, KELI organizes items as key-value pairs and inside the value, there could be links to other items. The main advantage of this model over the traditional key-value is that it could reduce significantly the amount of data requests when processing items that having relationships with each other. In such cases, instead of decomposing the original request into multiple lookups, applications could have KELI to chase the links between related items.
Assume that each device generates one signal for every 30 seconds, we have to process more than 20 thousand signals per second if only 10% of seven million vehicles in Ho Chi Minh City send data to the system. Such kind of workload cannot be handled efficiently by traditional databases and in-memory approach must be considered. However, with our current data layout, using traditional key-value in-memory stores such as Memcached or Redis [3] results about 30 data lookups in average for each new GPS signal. Which means we have to generate 600 million requests per second with mentioned workload. Such requirement is almost impossible for the average computer systems that we currently have. Therefore, a new solution is necessary.

3. BACKGROUND

To solve problems mentioned in the previous section, we have developed KELI. The design of KELI is inspired by the two key technologies: InfiniBand and RDMA. Thus, before discussing KELI’s design and implementation, let us first briefly introduce those technologies.

InfiniBand is the standard interconnection technologies in High Performance Computing community thanks to its high throughput and low latency. With the emergence of real-time and online analytic applications, the need of high performance network infrastructure is visible. Especially, decreasing in InfiniBand NIC prices in recent years makes data centers start to consider this technology as a replacement for the traditional Ethernet technology. One of the most important features that help InfiniBand achieve very high performance is its communication interface. In InfiniBand networks, nodes exchange data through messages using verbs API instead of treating data as a stream of byte and sending it through socket interface like traditional Ethernet. This interface allows applications to interact directly with the devices without interfering the Operating System. Bypassing the kernel yields very high communication efficiency as it avoids many sources of overhead such as waiting for system calls, buffer copying, etc.

The Verb API allows applications to access InfiniBand NICs through four types of verbs: RDMA Read, RDMA Write, Send, and Receive. Those verbs are posted by applications to send queue and receive queue stored inside the InfiniBand NIC. The NIC uses the information posted in its queues to perform communication tasks. There are two types of verb semantics: channel semantics and memory semantics. Send and Receive verbs have channel semantics: in order to get a message, the receiver must first explicitly post a Receive request to the queue. After that, the sender posts a send request for sending the message to the queue before actually transferring the data to the receiver. This is similar to the “read()/” operation used in socket implementation. RDMA Read and RDMA Write verbs, on the other hand, have memory semantics. Instead of explicitly invoking the operation, these verbs allow local machine to perform memory accesses (i.e. read and write) over the remote machine without letting the CPU on the remote machine to be aware of these operations.

Read and write data without interfering the CPU of the remote machine makes RDMA Read and RDMA Write have many performance advantages over Send/Receive verbs. The absence of remote CPU during read and write process makes systems use RDMA Read and RDMA Write scale out more easily since their performance does not bound by remote
4. KELI DESIGN

4.1 System Architecture

Our objective when designing KELI is to provide a lightweight in-memory store for hardly-changed datasets stored in complicated databases. That is, KELI copies items stored by the database to memory and lets applications to access them through its interface instead of sending requests directly to the database. The design of KELI also assumes that update occurs very rarely and a few changes do not cause serious impact on application performance and correctness.

The overall architecture of KELI is illustrated in Figure 1. Data is originally stored permanently on disk to ensure durability and availability. KELI is deployed entirely in memory. After starting up, KELI accesses data on disk and loads them into memory. It then reconstructs items’ structures based on its data model (i.e. key-value-with-links) and organizes them on dedicated memory regions. After KELI finishes loading data from disk, applications could create connections to it to get needed data instead of accessing the database. KELI does not support update operations (e.g. modify, write, and delete) so if applications want to change the content of data, it still has to send those requests to the database. Updates occurring at disk do not take effect immediately to the in-memory store. KELI, however, reloads the content of data on disk after predefined and fixed intervals.

4.2 Key-Value-with-Links Data Model

Similar to other caches, KELI treats records as pairs of key and value. Key of each record is a block of bytes with variable length and is used as its identifier. Record’s value is its payload. Unlike some key-value stores such as RAMCloud [13] or memcached [2], KELI considers record’s value as a tuple consisting of multiple attributes. Those attributes could be numbers, strings, arrays, and even links to other records.

Figure 2 shows an example of how KELI represents data from real life by its data model. Suppose we have a database storing the information about universities and students in Ho Chi Minh City. Each university (or student) is a single key-value pair whose key is its (his/her) ID. The value contains multiple attributes representing information about the student or university. It may contain links or a list of multiple links which can be used as pointers to other items. For examples, the “university_id” field of the student whose ID is 001 is a link to another item (e.g. a university) whose ID is QSB. Therefore, from this student, we can directly access that university by just following “university_id” link.

Storing the link to other items inside item’s value makes it easier to reason useful information that involve retrieving multiple records. For examples, in Figure 2, given an ID of a student, suppose that user wants to know if his hometown is the neighbor of the city where he is studying. KELI must first access the record whose key equals to this ID. From this, it then travels across the link to his university and then goes to the city where this university located. At the same time, from the student’s record, KELI goes through the link to his hometown and checks if the list of its neighbor contains the link to his university’s location to make the final answer. This process only requires some link chasing while in relational models, we must perform multiple cumbersome joins to produce the answer. In the case of traditional key-value data model, the application must decompose the request into multiple item lookups, each lookup is equivalent to one link chasing. However, with the emergence of RDMA technology, item lookups are much more expensive than just following links so using traditional key-value model takes more time than applying key-value-with-links model.

4.3 Data Layout

Since RAM capacity is much smaller than that of secondary storage, utilizing memory space effectively is a crucial requirement. To do so, avoiding or reducing fragmentation is necessary since this is the primary source of low memory utilization. [15] showed that standard memory allocators such as malloc in C do not handle this problem well. Therefore, to avoid fragmentation, we do not allocate space for new items at the time they are created but we reserve many contiguous memory slots in advance and then fill them up sequentially by the content of new items.

KELI updates its content in batch-style. When the update process is triggered, it first allocates new memory regions for new items then fills them up with the content of data stored in the disk-based database. After that, KELI deallocates the memory regions of old data and use new memory regions for answering upcoming requests from applications. To ensure applications do not access old items after update takes place, KELI halts all active connections from clients and have the clients to reestablish those connections to the server to obtain the new content.

Similar to other in-memory key-value, KELI employs a hash table to index items by their key. The implementation of the hash table is based on Cuckoo hashing [14] since this technique ensures constant complexity in the worst case. Hash table does not hold the content of hashed items but it instead stores the pointer to the actual data. This means a given key, access to the hash table would return null (i.e. item not found) or a pointer to the value of the item owning this key. For each new item, KELI fist finds a room in allocated memory regions for it and write its content on this region. After that, the key of this item and the pointer to its content are added to the hash table.

Some store such as HBase keeps items in memory in form
of memory objects to simplify data management tasks. However, this approach often requires the server to serialize those objects to an array of bytes before transferring them to the client. Serialization adds significant overhead to request processing especially in small ones like item lookups. KELI avoids serialization by converting the content of item into an array of bytes right after it being put into memory.

For each lookup request, KELI first uses the hash table for searching the item using the key included in the request. If it finds out the pointer to needed item, KELI transfers the content of memory region pointed by this pointer back to the client. The client then uses the data schema to interpret the meaning of the result.

### 4.4 Request Processing

KELI takes the advantages of advanced network technologies such as RDMA and InfiniBand to deliver very low latency data service. Figure 3 shows how KELI process data requests from client. KELI’s communication modules are built upon IB verb programming model. Given a key, the client asks for its value by sending a request using "ib_send" operation. The request is received at server side by a listener which has responsibility for receiving any incoming requests from the client. In order to maximize KELI performance, we let the listener continuously ask input queue for new requests instead of passively waiting for the queue to inform the client. In order to maximize KELI performance, we let the listener continuously ask input queue for new requests instead of passively waiting for the queue to inform the client. The new message like traditional techniques. Although this approach wastes a lot of CPU cycle for polling input queue, it makes KELI response to the new request very quickly.

When listener discovers that there is a new request in the input queue, it then pops this request out and forwards it to one of worker threads in worker pool. Worker thread is randomly chosen to ensure the workload scatters evenly between threads. After receiving a request from the listener, worker thread then searches the needed item by accessing the hash table. If the required item is not found, a response with empty payload is sent back to the client using "ib_recv" operation. If worker thread can locate the needed item via the pointer returned by the hash table then it puts the content of the item to the payload of the response and sends it back to the client. "ib_recv" is also used to transmit data.

The client has responsibility for interpreting the meaning of the payload of the response received from the server. If the needed item contains links to other items and application also wants to retrieve them, the client does not make another lookup request but using RDMA Read to directly read the content of linked items from the server. The reason of using RDMA Read instead of creating a new lookup request is that item lookups involve in executing APIs associated with InfiniBand verbs interface. Those functions require both client and server to participate in processing the request. In contrast, RDMA Read allows the client to directly access server’s memory without informing server’s CPU. Therefore, executing RDMA Reads is much cheaper than invoking a full item lookup request.

### 5. PERFORMANCE EVALUATION

This section illustrates experiments we have conducted to evaluate the performance of KELI. In order to examine the benefits of using KELI, we compare its performance with Redis. All experiments are carried out on a small cluster. Data nodes are equipped with CPU Intel Xeon E5-2670 and 32GB memory. Data nodes are connected via InfiniBand 40 Gbps network. We use Redis version 4.0.3 in all experiments.

Figure 4 shows the data schema we use in experiments. This schema is constructed according to the motivation application we mentioned in Section 2. There are four datasets we have to maintain in memory: street, node, segment, and cell. All of them are generated based on OpenStreetMap’s...
Table 1: Size of datasets used in experiments.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>street</td>
<td>288669</td>
<td>Represent streets on digital map</td>
</tr>
<tr>
<td>segment</td>
<td>4204072</td>
<td>Lines that can be put together to form streets</td>
</tr>
<tr>
<td>node</td>
<td>4067026</td>
<td>Segments’ endpoints</td>
</tr>
<tr>
<td>cell</td>
<td>145857</td>
<td>Disjoint areas used to group segments</td>
</tr>
</tbody>
</table>

Figure 5: Item lookup latency of KELI in comparison with Redis.

Redis is used as a cache (logging mode is disabled) which means data will be lost when it stops. Therefore, we save datasets permanently on disk in form of JSON files and push them to Redis each time it starts. On the other hand, KELI lets MongoDB keep its datasets permanently and each time it starts, KELI reloads data from MongoDB to main memory. Since MongoDB manages data in form of documents, KELI must transform them to key-value format and link related items together before using them.

5.1 Latency

Let us first examine the latency of item lookups of KELI and Redis. To do so, we create a client and make it generate multiple item lookups. The client then sends the lookup requests to each store to have them search for the value of the keys contained in the requests. If there exist items whose key matches that contained in the request then stores will send its value back to the client. The client chooses key randomly from the set of keys available in datasets to ensure that there is request causes the server to create a response with a null payload.

Figure 5 shows the 99th percentile latency of KELI and Redis. Clearly, KELI is three times faster than Redis. The main reason for this is that while KELI implements InfiniBand verbs to transmit both requests and responses, Redis still uses traditional communication mechanism (e.g. Socket). This decision makes Redis suffer from many disadvantages. Firstly, exploiting InfiniBand verbs allows KELI to communicate directly with Network drivers without invoking system calls to the kernel (kernel bypass). Secondly, kernel bypassing helps KELI to move data directly from its local memory regions to NIC's buffer without using intermediate buffers provided by Operating System as Redis does when using Socket interface (zero copying). Therefore, KELI could reduce significantly communication overhead caused by system calls and copy operations.

5.2 System Throughput

Using the same configuration with experiments in the previous section, we evaluate the throughput of KELI and Redis by changing the number of clients sending the request to each store. Figure 6 shows the experiment results. Clearly, KELI has better throughput than that of Redis when there are just a few clients. However, KELI does not perform lookup requests well when the number of clients increases. Particularly, its throughput almost does not change when the number of clients is greater than four. On the other hands, Redis scales very well. Its throughput keeps increasing as the number of clients increases. Although using InfiniBand verbs helps KELI to utilize the network must better than Redis. Its communication modules' implementation is still relatively naive. Handling so many tasks at server side makes KELI’s performance be easily bounded by server’s CPU. Also, in order to bypass the kernel, memory regions holding KELI's items must be pinned to physical memory to avoid page swapping. Since the amount of data to be pinned is very large compared to the page size, the page table that maintains the mapping information between virtual pages and physical addresses cannot be held entirely on the NIC driver. Actually, it can just cache some parts of the table in its local memory. Naive implementation and choosing keys randomly make KELI suffer from heavy cache misses leading to poor performance.

5.3 Experiments on Real Datasets

In this section, we will show that despite scaling poorly, KELI still works well with real workloads by letting both KELI and Redis handle workloads generated by practical applications. In our experiment, we deploy the traffic application that we described in Section 2 at the client side and let another machine push GPS signals continuously to this application. To improve overall throughput, we do not send data to the application one-by-one but by a group of 10000

Figure 6: Throughput of KELI and Redis when changing the number of clients generating lookup requests.
signals at a time. Also, application splits the input data into 32 sub-groups and letting 32 worker threads to handle them individually. Those threads process data by retrieving map datasets from KELI or Redis and combining them with GPS signals to generate traffic information.

Figure 7 shows the amount of time application spends when cooperating with KELI or Redis. Apparently, using KELI makes the application run seven times faster than using Redis. This is because KELI uses link chasing to retrieve items needed by the application while Redis gets data through multiple lookup operations. For a single GPS, application must generate more than 2000 internal requests in average. As we have explained in Section 4, link chasing consumes much less time than item lookups do. Therefore, the cumulative latency of using Redis is much higher than that of using KELI.

Also note that the lookup operation of KELI is not well-implemented. As we have shown in the previous section, when the number of clients is high, KELI is outperformed by Redis. However, if the application requires items that have links to each other, KELI could avoid those expensive operations by chasing those links to reduce overall latency. As Figure 7 shows, even though handling requests from 32 clients – the case that Redis outperforms KELI in previous experiment – chasing links helps KELI to boost its performance significantly and finish its job seven times sooner than Redis.

6. RELATED WORK

Exploiting fast network technologies such as RDMA and InfiniBand to improve the performance of key-value in-memory stores is an attractive research topic. Similar to KELI, Pilaf [12] also uses Cuckoo hashing for item lookups. However, it lets the client use RDMA Reads to perform lookup operations instead of processing the request inside the server as KELI does. This decision makes lookups do not suffer from server’s CPU bound but since a single lookup requires about two to three RDMA Reads in average, their cumulative latency is much higher than that of KELI as pointer chasing requires only one RDMA Read. FaRM [5] also utilizes RDMA Read for item lookups and suffers from the same issues as Pilaf.

HERD [10], on the other hand, is implemented in an unconventional way: its communication layer is constructed upon unreliable connection types. Since unreliable transports reduce a lot of communication overhead, HERD could perform lookup operations at very high speed. This design, however, does not ensure reliable transmission which is highly required by modern data centers.

RAMCloud [13] and MICA [11] also bypass the kernel to achieve ultra-low latency request processing. Those stores, however, do not utilize RDMA like other systems listed above. They instead interact directly with the NIC to ensure their service still work well on commodity hardware. Although those stores could process item lookups at very high speed, using traditional key-value model still makes them sensitive to cumulative latency.

Together with new designs, other noticeable efforts have been made to improve the performance of existing systems. Jithin Jose et. al. [9] redesigned Memcached communication layer for RDMA capable networks. Bin Fan et. al. [6] enhanced Memcached’s performance by combining optimistic cuckoo hashing, a compact LRU-approximating eviction algorithm, and comprehensive implementation of optimistic locking. More recently, J. Huang et. al. [8] modified communication modules of HBase [1] to improve its performance over high performance networks. Most of those works only change the internal organization of well-known stores, they try to avoid or minimize modification on stores’ APIs and data model. Therefore, those solutions are still subjected from the cumulative latency of performing multiple lookups.

7. CONCLUSION AND FUTURE WORK

This paper describes KELI, a novel in-memory key-value store that relies on state-of-the-art network technologies to offer low latency caching services for real-time applications. The principle underlying KELI design is to allow items to be linked to each other. Doing so helps reduce the cumulative latency significantly when breaking the original request into multiple item lookups. KELI also exploits high performance network technologies such as RDMA and InfiniBand to deliver very low latency data service. Experiment results show that KELI works much better than Redis on real datasets.

However, there are still works to be done. Firstly, KELI does not scale well due to naive communication implementation. Improving communication modules is necessary to ensure KELI handle requests from multiple clients efficiently. Secondly, KELI is currently running on a standalone data server. However, since the amount of data is rapidly increasing, distributed approach should be taken into consideration. Finally, KELI is constructed upon the assumption that changes on the datasets rarely occurs and those changes do not severely affect the lookup results. In the future, in order to make KELI work well with a wide range of applications, a flexible update mechanism should be implemented.

8. REFERENCES


