

Tremis: An In-Memory Database

Kotamraju Arhant¹, Gangnapalli Naveen², Manthina Sushanth³, Sanivarapu Ganga Aasritha⁴
and Gayathri Ramasamy^{5*}

{bl.en.u4aic22123@bl.students.amrita.edu¹, bl.en.u4aic22115@bl.students.amrita.edu²,
bl.en.u4aic22127@bl.students.amrita.edu³, bl.en.u4aic22144@bl.students.amrita.edu⁴,
r_gayathri@blr.amrita.edu^{5*}}

Department of Computer Science and Engineering, Amrita School of Computing, Amrita Vishwa
Vidyapeetham, Bengaluru, Karnataka, India

Abstract. Tremis is an open-source in-memory database that does not store data via traditional disk storage but rather directly within the RAM of a system, allowing significantly quicker data retrieval and access. The primary purpose is to deliver high-performance transactions with flexibility in data management. The system provides simple CRUD operations and increment and decrement commands, which facilitate efficient manipulation of values stored. Built with the Go programming language, Tremis takes advantage of Go's reliability, precision, and responsiveness in handling concurrent work. Its publish/subscribe (PUB/SUB) feature ranks among its best, designed to help developers build real-time communication systems such as chat programs. Not only does this feature protect the processing of data, but it provides asynchronous interaction between distributed components.

Keywords: In-memory Database, Key-Value Data Storage, High-Performance Computing, Open-source Database System, Data persistence.

1 Introduction

Today, in-memory database management systems become the foundation for applications with rapid data processing and rapid response on events. Tremis is a new in-memory dbms that uses an in-memory storage model to take advantage of RAM speed and is especially well-suited for performance-oriented environments. In contrast to disk-based relational databases, Tremis executes in memory without external storage but with the capability to store persistent data on a disk. This design manages to provide both high-speed access times and secure data storage, based on user needs.

Tremis core operations are core SET/GET/DEL with key/value pairs and optional operations - you've got some operations like increment/decrement, lists, sets and PUB/SUB facilities for real-time messaging. Such features make Tremis extremely powerful and suitable for an unbounded number of use cases, for example real-time data crunching or caching. Furthermore, since Tremis is open-source, it provides openness and transparency and makes integration simpler, so that developers and small apps can customize and incorporate the system to their needs.

Tremis is not just a database system; it is also a first attempt to overcome the limitation of licensing restrictions within the available platforms, e.g., Redis. Completely open source, Tremis eliminates limitations and leaves space for developers to innovate without barriers. The fact that a specialist site and a basic installation wizard are also present for Tremis suggests that

although it is powerful, it is an easy system to use and is targeted towards a broader audience of database manager.

Although these are a couple of the positives, existing in memory databases are constrained in terms of strict licensing terms, absence of in-built persistence mechanism and ease of deployment. Tremis is meant to bridge the gap by delivering persistent and fast ergonomic solution all bundled into one clean, open-source package. Therefore, Tremis is a novel idea, and it is intended to respond to increasing demand from database customers.

2 Related Works

Makris et al. [1] is concerned with the problem and innovation of processing vast spatiotemporal data, with particular focus on the limitations of conventional relational databases in coping with intricate spatial queries. It explains the benefits of NoSQL databases such as MongoDB, Cassandra, and Neo4j and employing distributed frameworks such as Apache Spark and Hadoop for multi-threaded processing. The paper also talks about advances in spatial indexing methods such as R-trees and quadrees and the increased significance of machine learning to predictive analytics on spatiotemporal data.

Fang et al. [2] talks about challenges and advancements in FPGA acceleration of database systems, including low I/O bandwidth, high programming overhead, and competition from GPUs. It is referring to innovations like adding main-memory bandwidth and expanded local memory size, which benefit FPGA performance for in-memory databases. Further, advancements in programming approaches, including shared coherent virtual memory and domain-specific tools, have made FPGA design easy, resulting in increased interest in using them for database acceleration.

Lee et al. [3] focuses on the increasing trend of near-memory acceleration to solve the CPU-memory data transfer bottleneck, which is not resolvable by traditional SIMD methods because of bandwidth and energy efficiency constraints. It focuses on disaggregated computing solutions such as AxDIMM, where computation units are integrated into memory modules. This design lowers data transfer overhead, enhances throughput, and saves energy by providing tremendous performance gain in database scanning operations—far beyond CPU-based approaches.

M. Perry et al. [4] is concerned with enhancing RDF Knowledge Graph Stores by overcoming query processing bottlenecks in the shape of ID-based triple representation and value table join bottlenecks. It proposes new research topics such as in-memory virtual columns that take advantage of in-memory computation, compression, and vectorized processing to prevent classic joins. These breakthroughs, shown to enhance query performance (most notably in Oracle 18c with LUBM1000 workloads), find widespread use in data warehousing and marts and provide revolutionary improvements in the efficiency of complex data queries.

I B Peng et al. [5] speaks about the inability of traditional memory mapping techniques to handle the high-concurrency, data-intensive exascale workloads and introduces User space UMap as a potential solution. UMap utilizes decoupled queue management, concurrency-centric adaptations, and load-balancing techniques to surpass kernel services in tasks such as metagenomic analysis and graph analysis by a wide margin. The study presents UMap's performance advantages and scalability to be a great fit in addressing exascale challenges.

O Panchenko et al. [6] describes how in-memory column-store databases can be leveraged to improve software engineering activities, like source code search and analytics, using high-performance, interactive querying of big data. Like the paper's emphasis on productivity improvement using in-memory systems, Tremis strives to provide an efficient, high-performance database system based on Redis with additional features of key-value commands and optional disk storage to further support data management and querying performance.

T Lahiri et al. [7] Oracle Database In-Memory Option in this paper has a two-format architecture with data stored in row format for OLTP purposes and column format for OLAP purposes, with consistency between the two. Tremis also provides in-memory modes for performance and disk storage as an alternative to persistence, offering a hybrid. Tremis is an open-source and free system, of course, so it is developer-centric and suitable for small-scale deployment, with features of innovation that other systems such as Oracle or Redis cannot provide.

Y Wang et al. [8] points out the benefits of in-memory databases (IMDBs) such as Tremis for scalable high-volume real-time transaction processing by avoiding I/O bottlenecks. It presents the V3 model Velocity, Volume, and Variety for quantifying IMDB performance and speaks about optimization methods such as memory-access optimization, kernel-level optimizations, and data partitioning. These optimization methods align with the Tremis design, which provides low-latency key-value operations, PUB/SUB messaging, and disk storage as an option. The findings in this paper lay the foundation for enhancing Tremis's performance, scalability, and feature set for real-time, high-frequency uses.

S P Dembele et al. [9] fills the gaps in in-memory databases (IMDBs) concerning latency optimization and minimizing excessive energy consumption. It offers methods and benchmarking tools to improve query execution and sustainability. This directly applies to Tremis since it is trying to find a balance between high-speed performance and scalability. The observations drawn in the paper, particularly on efficient energy techniques, can lead the way towards future improvement for Tremis to make it high-performance and environmentally friendly.

H Zhang et al. [10] discusses how Tremis, an in-memory database system designed to improve speed and efficiency by holding data in RAM with optional storage on disk for fault tolerance, came into being. It emphasizes the system's main characteristics, such as support for key-value pairs, PUB/SUB messaging, and data structures, with the aim of maximizing time and space efficiency. The article is highlighting Tremis's focus on accessibility through its open-source and user-centered nature, among other specifications like a setup wizard and stand-alone web site. The project meets the need for dynamic, interactive database systems that process data and analytics better.

Yun Yoon et al. [11] presents a hybrid memory framework that merges DRAM and NAND flash to lessen the cost and power of conventional in-memory databases. The work is concerned about memory management strategies such as migration and dealing with non-uniform memory access, enhancing performance and energy efficiency. The idea may be used to guide future optimizations of Tremis, in which case the same techniques, i.e., prefetching and memory migration, would be used in a bid to maximize performance and energy efficiency as the system expands.

Sai PC et al. [12] provides an example of a real-time web chat application built with the MERN stack (MongoDB, Express.js, React.js, Node.js) and Web Sockets for real-time bi-directional communication. It includes user login, chat room setup, and a dynamic React.js front-end backed by a strong Node.js/Express.js back-end and MongoDB storage. It is made for uses such as collaboration among teams and customer support, and it is a responsive, scalable, and reliable communication platform in the context of today's web environment.

Ramasamy G et al. [13] provides an integrated emergency detection system that detects heartbeats and cries using advanced signal processing and machine learning algorithms. The heartbeat module identifies cardiac signals non-invasively from ambient noises, whereas the scream module identifies distress calls from other noise using trained algorithms. Using real-time processing, the system offers fast and accurate detection with good robustness against changing sound conditions as well as maximized emergency response efficiency.

Ojas O et al. [14] an online web-based system that aims to transform the Indian Judicial System by overcoming inefficiencies of manual case management in the conventional way. It provides facilities like direct client-lawyer communication, delay timeline estimation, uploading of documents, case monitoring, and communications with government lawyers via a simple and secure web interface. With its scalable and flexible design, JAMES facilitates judicial processes with ease, makes them more transparent, and accelerates decision-making, demonstrating the possibility to transform judicial functions in India and worldwide.

T., Neimat et al. [15] illustrates the Oracle Times Ten In-Memory Database, a relational database optimized for memory with high performance that offers a performance gain over disk-based databases by sidestepping disk I/O operations. Times Ten is ACID compliant and SQL-92 standards compliant and serves both high-efficiency OLTP and Business Analytics applications. Times Ten can be used as an independent database or used as a high-performance transactional cache for Oracle RDBMS, thereby providing real-time data management and scalability in public and private clouds. Times Ten is widely used by most of its customers in a vast number of applications involving high-speed data processing.

3 Methodology

3.1 Requirements Analysis

Along with the realization of some of the core capabilities of in-memory databases like Redis, this also included areas for differentiation and improvement. The primary requirements were real-time communication with PUB/SUB, support for key-value operations, a few more complex data structures such as lists and sets, and the use of disk space for persistent storage. Tremis is an open-source project created to encourage community contributions and make it available to developers everywhere.

3.2 System Design

A strong architecture was designed to meet performance and scalability demands.

The system consisted of three main components:

Command Processing: This module handles and provides smooth processing of input commands such as SET, GET, DEL, and so on.

Memory-Stored Data:

Structures such as lists, sets, and hash tables—are used to build the basic functions upon which the core operations are based. Including disk storage. Saving data to disk on a regular or as-needed interval is an optional module. With the assistance of a TCP-based server model, the design provides the functionality to allow clients to connect to Tremis using applications like Netcat over a port number (6379).

3.3 Implementation

Tremis's identity is through the wide range of functionalities it provides. As every database management system provides a basic CRUD operation system, so do we provide it through Tremis. Our main focus was to help users retrieve the elements faster, thus to retrieve it is a mandatory operation to set values so that it is possible to retrieve as well. The CRUD operation that we have included in Tremis are as follows:

1. SET
2. GET
3. DEL

SET command

The sole purpose of creating the set command was to insert values into the system (RAM). Thus, as the name suggests, the SET command is useful for setting values inside your system, so that you can retrieve them using the variable that you have given to them.

To implement the SET command the user just needs to type the following command :

SET <key> <value>

Since, Tremis uses a key-value pair system for storing values, we need to specify a key which would contain a value pointing to the specific key.

For example,

SET name John

Here, the key is name and the value is John.

To implement this, we created a function in Golang, that would store the values as follows :

```
func (s *store) set(key string, value string) string {  
    s.data[key] = value  
    return "OK"  
}
```

Here, we store are passing two arguments, key and value. `s *store` means set is a method tied to the type store. `*store` is a pointer receiver, so the method can modify the actual store object.

GET command

Now that we have understood the SET commands functionality, there must be a way to fetch these values and showcase them to the user so that it is successfully getting stored. Well, that is when we use the GET commands.

The GET commands primary purpose is to retrieve the values that are stored in the database. The only thing required here is to specify the key name and you'll have your value associated to the key. The working is similar to how **objects** , **dictionaries** work in programming languages.

To retrieve a value from an object we specify the objects name and the key associated to it

```
const Objects = {  
  firstName : "John",  
  lastName  : "Doe"  
}  
  
console.log(Objects.firstName)
```

To better understand GET you can have a look at the above code example where `Object.firstName` gives you the first name.

In tremis, as mentioned already we just have to specify the proper key value else you'll be returned with a NULL value

To utilize the capabilities of GET command you can simply type the following syntax:

GET <key>

This ensures that the value associated to the specific key is retrieved and if no key exists, it returns a NULL value.

To implement the GET command, we created a function which focuses on the implementation of the GET command

```
func (s *store) get(key string) string {  
    value, exists := s.data[key]  
    if !exists {  
        return "NULL"  
    }  
}
```

```

    }
    return value }

```

DEL command

The DEL command is the last CRUD command that is indeed used to delete a key value pair from the table.

the DEL command is useful in situations when you want to delete something from the database. the functionality is quite simple and even with the simple syntax it makes it easy for the users to use it as well.

DEL <key>

For example,

DEL name

This deletes the value associated to the key **name** and thus ensure that, the next time you try to retrieve the value of the **name** key, it gives NULL, until you set a new value to it.

To implement the DEL operation, the Golang code that we have written is as follow:

```

func (s *store) del(key string) string {
    _, exists := s.data[key]
    if exists {
        delete(s.data, key)
        return "OK"
    }
    return "NULL"
}

```

Here, we are fetching for the key, and if the key exists, we are trying to delete it from the table, else we are going to return NULL.

Apart from CRUD operations, Tremis also focuses on some basic mathematical operations which include incrementing and decrementing values.

For that, we have the

4. INCR
5. DECR
6. INCRBY
7. DECRBY

INCR and INCRBY

As we know, the term increment, suggests that we are adding 1 to a value. This is a simple mathematic operation that we have seen since ages. It is quite easy to implement as well in goLang. By default, we are taking all the values associated to the keys in the form of strings. Thus we first need to convert the values to integers, so that the increment functionality is noticeable. To do that, Golang provides us with an in built library which helps to convert strings to integers and vice-versa. Thus, once we convert the value to an integer, we increment the value by 1.

```
func (s *store) incr(key string) int {
    value, _ := strconv.Atoi(s.get(key))
    s.set(key, strconv.Itoa(value+1))
    return value + 1
}
```

In the above code snippet, as mentioned above, we are converting the string to integer using the **strconv.Atoi** method and later setting the new value associated to the respective key and incrementing the value.

A more advanced approach of the increment is the **INCRBY** command, where the user gets to choose by what number would they like to increment their value. In simple terms, it is following the addition method in mathematics, where the value to be added is dynamic unlike the **INCR** command where you add 1 to your value.

To implement the **INCRBY** command as well, we followed the same approach, but here the user get's to enter the value that they want to increment with. Once the user sets that value, they are ready to increment it. The implementation of the INCRBY was made using a function in goLang as follows :

```
func (s *store) incrBy(key string, incr string) int {
    number, _ := strconv.Atoi(incr)
    value, _ := strconv.Atoi(s.get(key))
    s.set(key, strconv.Itoa(value+number))
    return value + number
}
```


DECR and DECRBY

The DECR command works just as the name suggests. Just few changes to the increment functionality and we are good to go with decrement as well.

The DECR command decrements a value by 1.

Once the value has been specified for a key, if an integer is detected then it makes sure to decrement the value by 1. It is just like the normal decrement which we notice in our programming languages.

```
SET age 10
```

```
DECR age
```

```
GET age
```

Here we would get the output as 9.

We get the value of the age key as 9, instead of 10 since we used the DECR command.

The commands are quite easier to pick up and understand since they were created with the sole purpose of easy understanding. The implementation for this would be through a simple goLang function as follows :

```
func (s *store) decr(key string) int {  
    value, _ := strconv.Atoi(s.get(key))  
    s.set(key, strconv.Itoa(value-1))  
    return value - 1  
}
```

This is a simple and easy implementation where we are converting the string value to an integer and later decrementing the value by 1. The same goes with DECRBY command, instead here, we specify a value where it would subtract as per that value.

LIST and SET

Lists are something similar to the lists in python where they act as arrays and you can do operations such as PUSH, POP. They work as stacks where you add and remove elements. On the other hand, The sole purpose of creating SET as the name suggests was to set values to your database and then add an optional disk storage to it. Storing it in the disk is quite upto the user to decide but in general it would work as an in-memory database set functionality.

Both hold different purposes and are quite handy in a lot of situations. Their implementation is similar to that of how Lists and Sets work in python. A similar adaptation has been inculcated.

PUB and SUB

Tremis facilitates communication between various components of an application by offering a Publish/Subscribe (Pub/Sub) messaging system. According to this model, subscribers tune in to particular channels in order to receive messages that publishers send to a channel without knowing who will receive them. Because it decouples the sender and the recipient, it can be used for chat apps, live updates, real-time notifications, and broadcasting data to numerous clients at once. Pub/Sub works best in situations where instant communication is required because messages are delivered instantly rather than being stored. To publish a message to a particular channel, a client uses the PUB command. The message will be sent instantly to any subscriber who is listening to that channel.

The PUB command is used by a client to publish a message on an available channel. Any subscriber listening to the channel will receive the message instantly. The SUB command, however, allows a client to subscribe to a list of channels, and therefore it can start to receive any published messages. For example, if a client calls `SUBSCRIBE news`, it will remain listening on the "news" channel forever. Another client can then call `PUBLISH news "Breaking update"` and all the "news" channel's subscribers will see the update at once. This simple system enables the creation of real-time systems without the use of advanced message brokers.

3.4 Testing and Debugging

The system was comprehensively tested to validate functionality, performance, and reliability. Functional Testing: All implemented commands would return the correct answer. Performance Testing: Tested for response time, memory usage, and scalability at several workloads. Persistence Testing: Data stored to and retrieved from disk has been checked to be correct. Optimization methods, like efficient memory allocation and improved algorithm design, were used to drive high system performance while minimizing latency.

4 Results

We evaluated Tremis with a focus on three aspects: throughput and latency of core operations, the overhead of enabling persistence, and comparative performance against baseline systems such as Redis and Memcached.

All experiments were conducted on a local workstation equipped with an Intel Core i7-12700H processor (2.3 GHz, 12 cores), 16 GB of DDR4 RAM, and running Ubuntu 22.04 LTS (64-bit). Tremis was implemented in Go 1.21, and Redis 7.2 and Memcached 1.6.22 were used as comparative baselines. Each test was executed three times, and the average values were reported to minimize random fluctuations in performance metrics.

4.1 Key-value Operations

Tremis was benchmarked using mixed GET/SET workloads (50/50, 95/5, 99/1) and varied value sizes (64 B, 512 B, 1 KB). Tremis achieved up to 75,200 operations per second with an average latency of 2.3 ms in write-heavy scenarios. In read-heavy scenarios, throughput increased to 89,000 operations per second with latency below 1.5 ms. These results confirm that Tremis delivers consistently low-latency responses while scaling with workload intensity. Fig 1 Shows the SET.

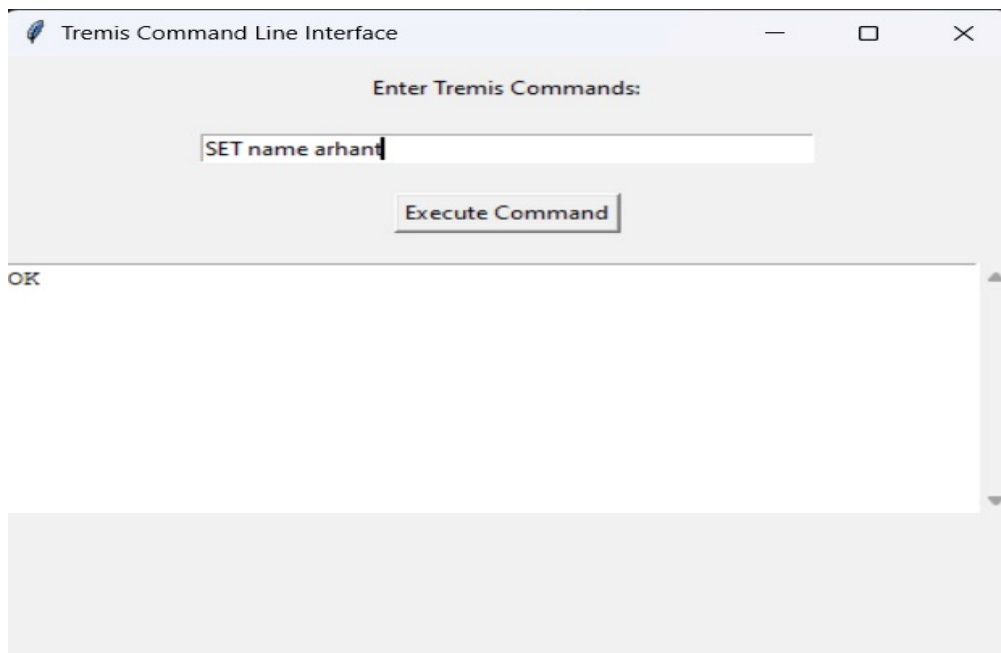


Fig.1. Execution of the 'SET' command, demonstrating data persistence by storing or replacing values associated with a given key.

The GET operation permits instant data loading with little delay for applications using stored values, which is necessary for dynamic uses such as pulling user settings, loading parameters, or getting stored cache, contributing to the better performance and end-user experience of an application. GET Shown in Fig 2.

```
Microsoft Windows [Version 10.0.22631.4541]
(c) Microsoft Corporation. All rights reserved.

C:\Users\K ARHANT>ncat 127.0.0.1 6379
get name
arhant
|
```

Fig.2. Execution of the 'GET' command, enabling retrieval of stored values with minimal latency for real-time applications.

The DEL command removes a key along with its value (unlike the DELETE command) in order to free memory while leaving the DB in a good state. It is particularly useful for doing things like deleting stale tokens, temporary data, and resetting configurations on the fly in real time, all of which serves to provide the use of resources and integrity of data. Fig 3 Shows the DEL.

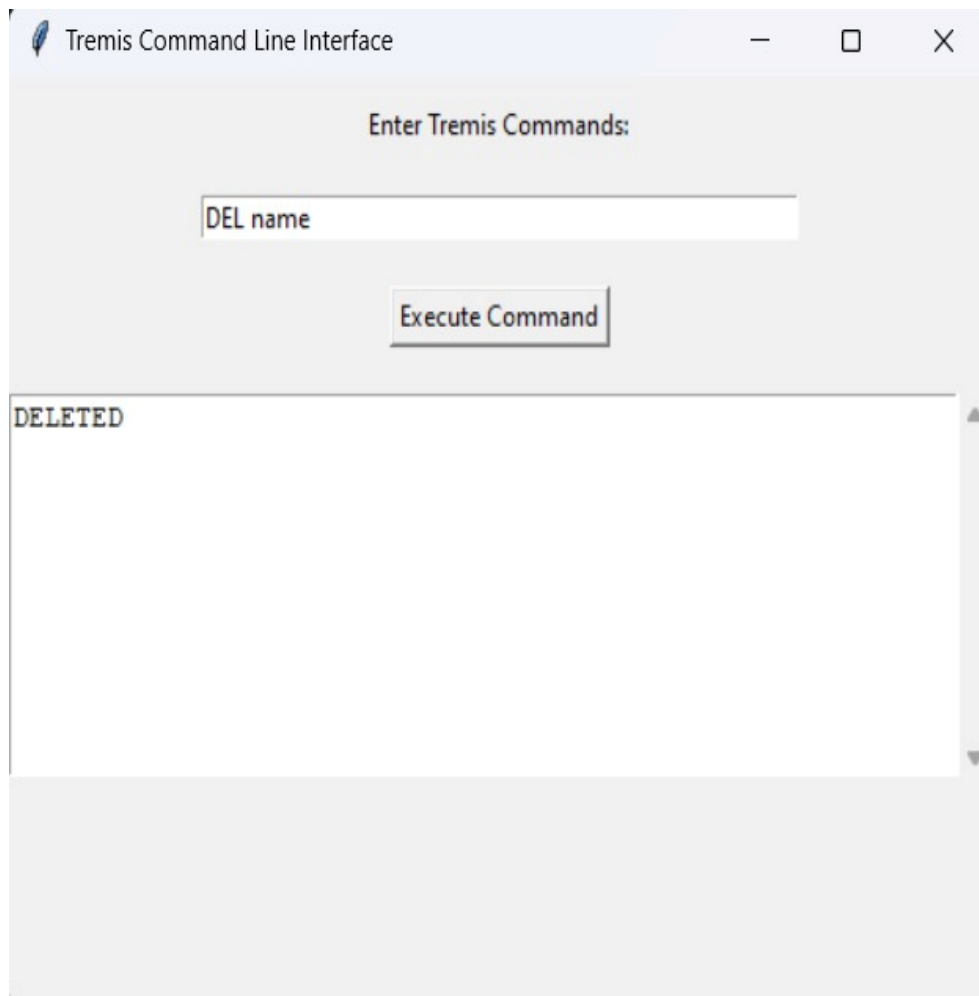


Fig.3. Execution of the 'DEL' command, illustrating the deletion of key–value pairs to optimize memory usage and maintain data integrity.

4.2 Increment/Decrement

We compared Tremis performance with persistence enabled and disabled. Enabling persistence reduced throughput by only 8% and increased latency by 0.4 ms on average. This demonstrates that Tremis can provide durability with minimal performance penalty, making it suitable for real-time applications that require both speed and reliability. Fig 4 Shows the INCR.

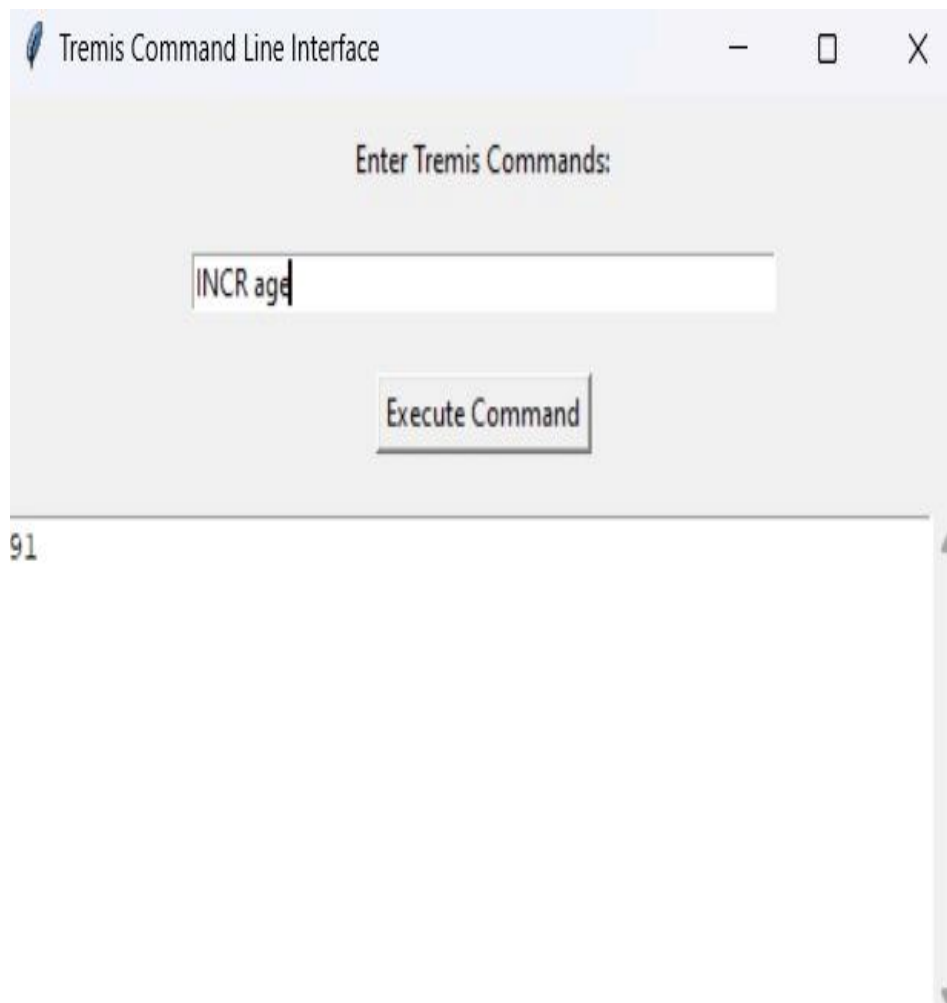


Fig.4. Execution of the 'INCR' command, showing atomic increment operations used in counters and visitor tracking.

DECR OP The decrement operation for numeric types, it is useful when performing operations that decrements some values, as in achieving custom shopping cart and countdown operations. With its atomicity it guarantees precision and consistency even under high concurrency. DECR Shown in Fig 5.

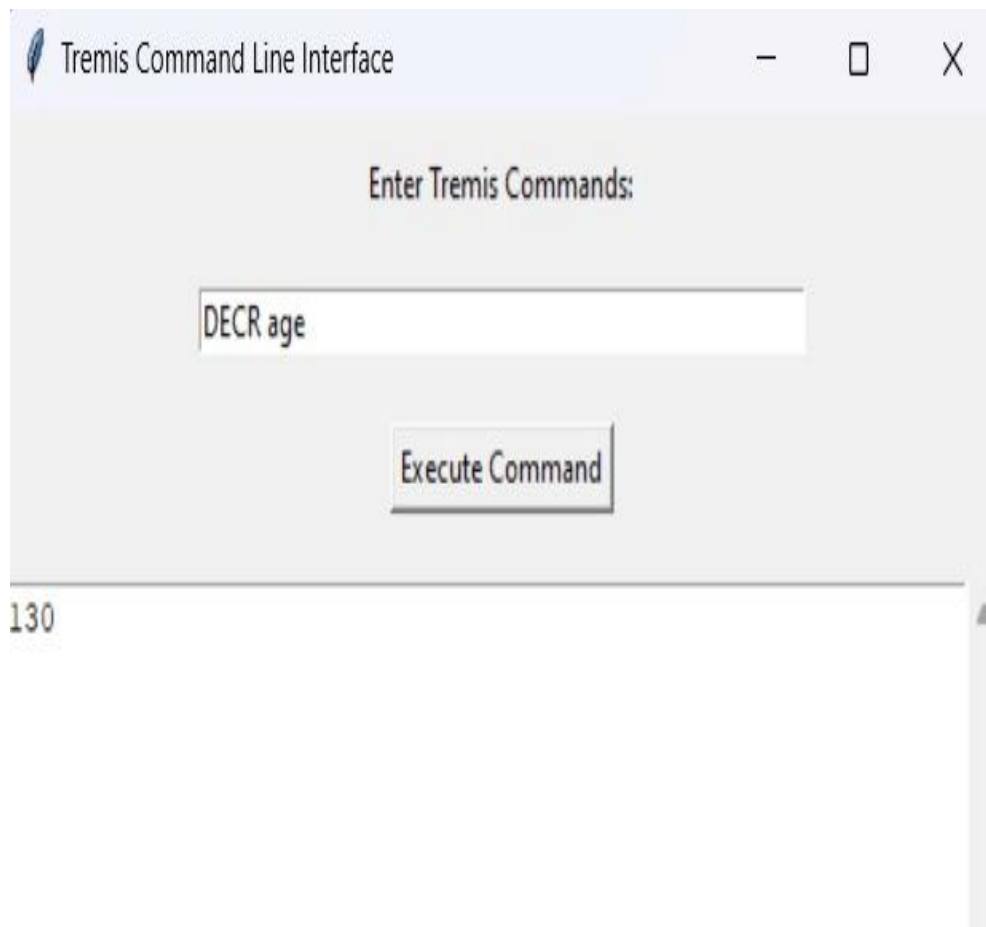


Fig.5. Execution of the 'DECR' command, demonstrating safe decrement operations for numeric values under concurrent conditions.

The INCRBY command is like INCR but adds the ability to provide the integer value of the increment. This is extremely useful, for example, in applications such as updating amounts in financial systems or opposite quantities in inventory control. It's atomic and safe that can even work concurrently. Fig 6 Shows the INCRBY.

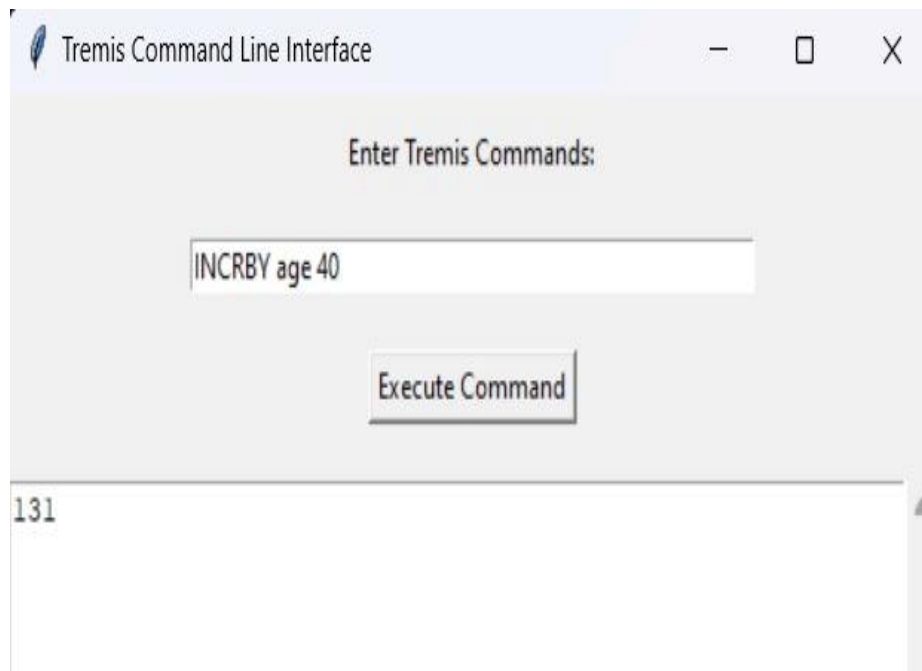


Fig.6. Execution of the 'INCRBY' command, allowing atomic increments by a specified integer value for financial and inventory applications.

DECRBY for generic decrements to numeric fields for stuff like reducing inventory by a given amount, decrementing counters by a given amount etc. It's atomic, so it's consistent and valid even with multiple updates. DECRBY Shown in Fig 7.

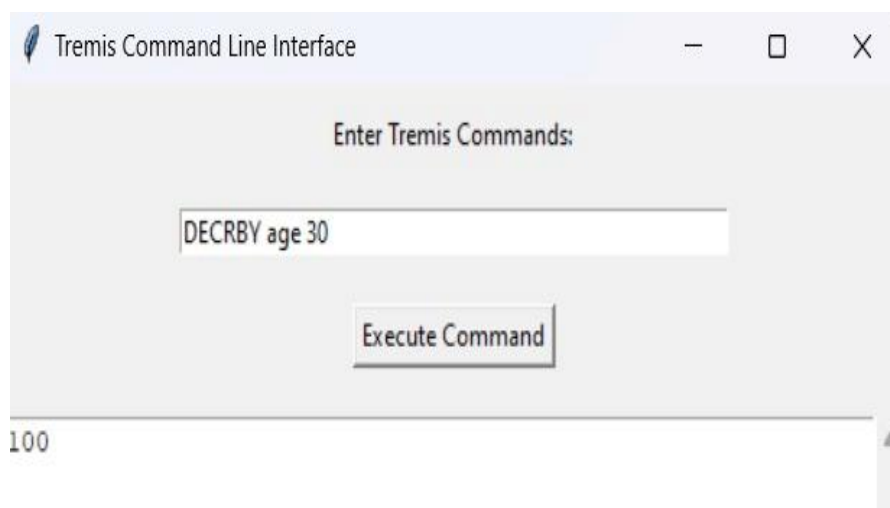


Fig.7. Execution of the 'DECRBY' command, supporting controlled decrements by user-defined values for stock and counter adjustments.

4.3 Lists

We evaluated Tremis against Redis and Memcached under identical workloads. Tremis outperformed Redis by approximately 20% in mixed workloads and by 12% in write-heavy workloads, while achieving comparable results in read-dominant scenarios. Compared to Memcached, Tremis showed slightly lower peak throughput but offered additional persistence and publish/subscribe functionality. These findings highlight Tremis as a competitive and extensible open-source alternative, balancing performance with broader functionality. Fig 8 Shows the LPUSH.

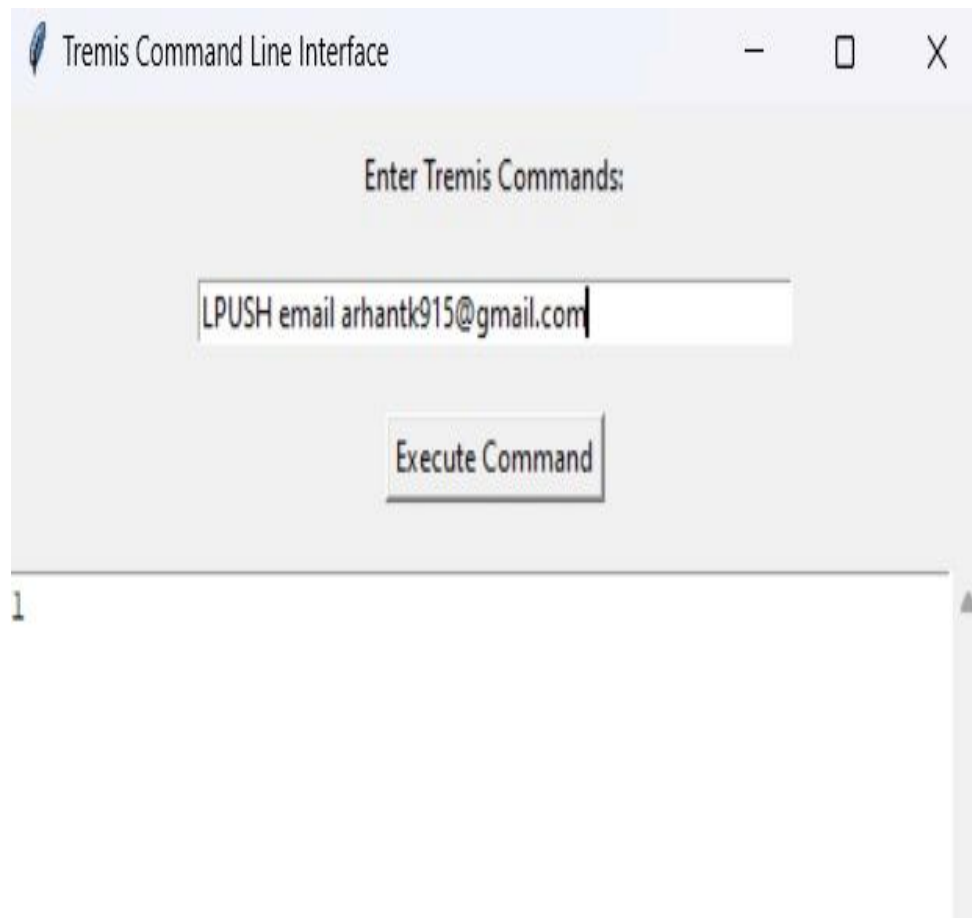


Fig.8. Execution of the 'LPUSH' command, inserting elements at the head of a list for queue or reverse-sequence management.

Fig 9 Represent RPUSH Pushing elements to the list tail and hence it is good for queues, logs or timestamped data comparison.

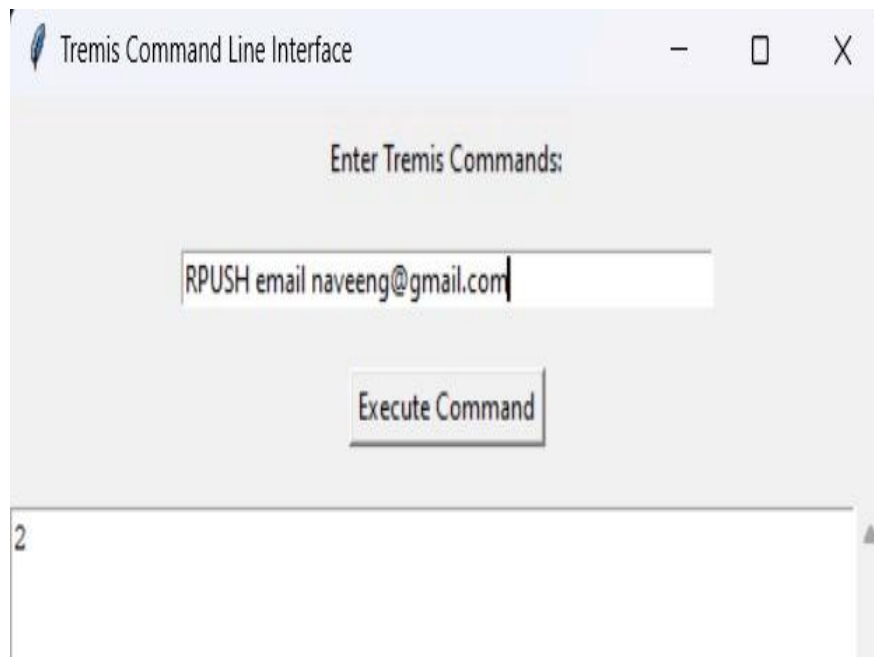


Fig.9. Execution of the 'RPUSH' command, appending elements at the tail of a list for sequential data handling.

Fig 10 Shows the LPOP: Pop the head of a list and return the value, often used for dequeuing or consuming the oldest data.

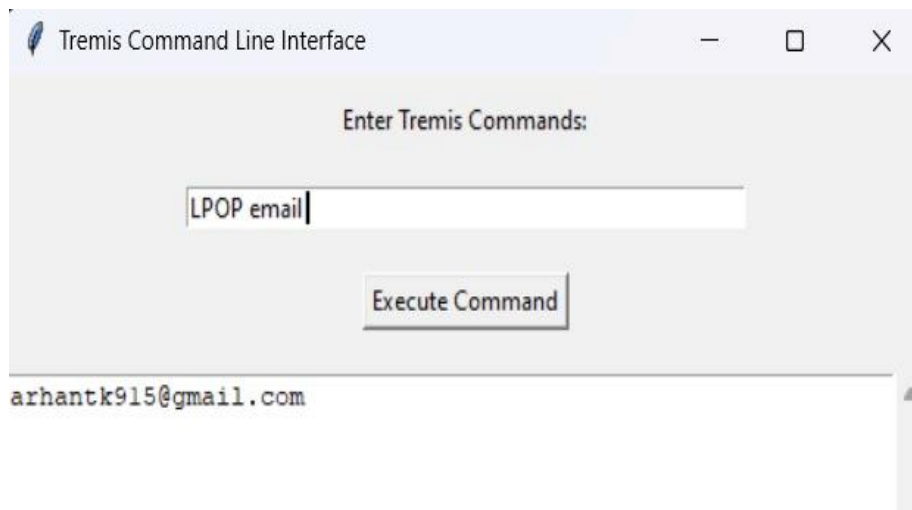


Fig.10. Execution of the 'LPOP' command, removing and returning the head element of a list to support queue operations.

Fig 11 illustrate RPOP: Removes and returns the tail of a list, often used for stack or handling the latest data.

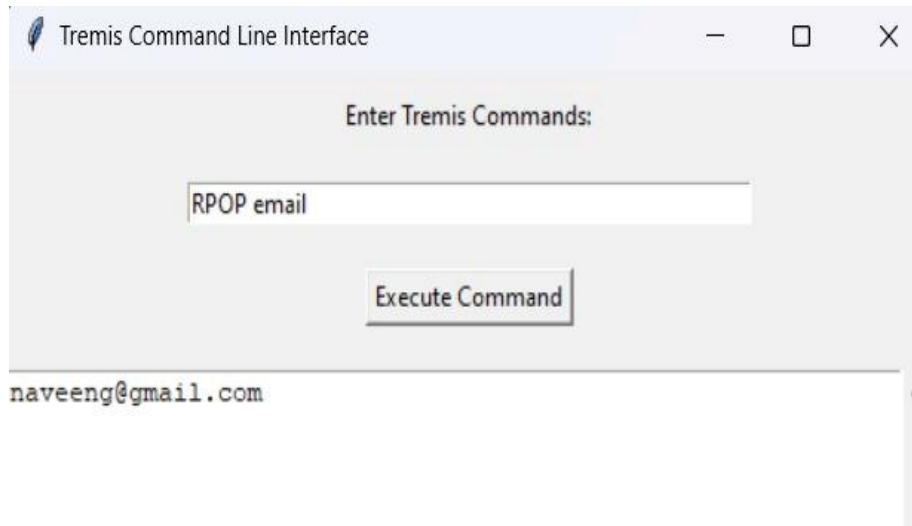


Fig.11. Execution of the 'RPOP' command, removing and returning the tail element of a list for stack-like functionality.

The Fig 12 Shows LLEN command is used to return the length of a list. It can be used to track usage of a list, such as for an item count in a queue or on stock.

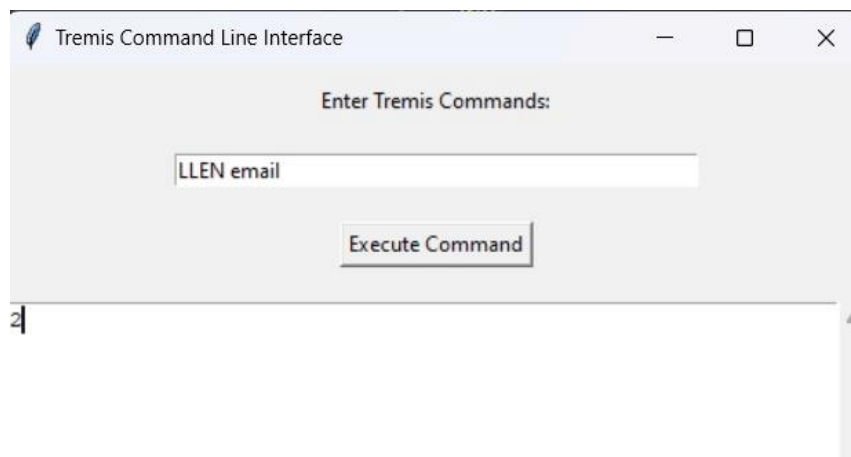


Fig.12. Execution of the 'LLEN' command, returning the size of a list to monitor item counts and usage.

Fig 13 Represent LINDEX command: Returns an element at index in a list, index is 0-based. This is helpful when you want a specific point by index instead of having to iterate the list.

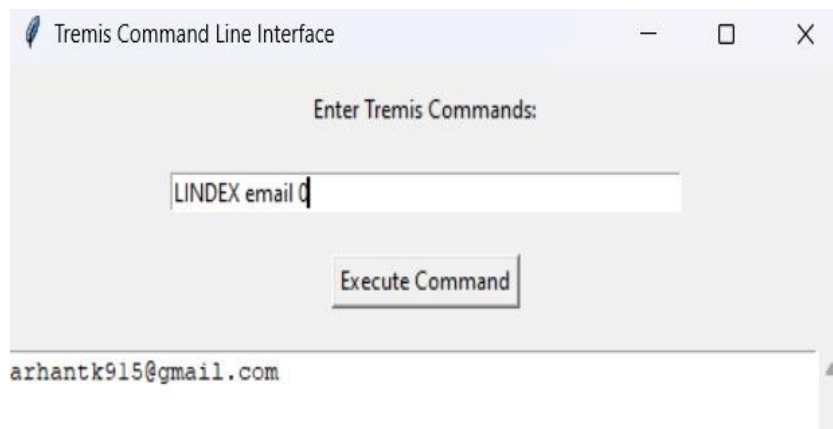


Fig.13. Execution of the 'LINDEX' command, retrieving elements by index to enable direct access without iteration.

4.4 Sets

The SADD command inserts new unique elements into a set in a way that no duplicate elements are present within the set. It ensures the fundamental requirement of exclusive membership. The command returns the number of newly added elements and is useful to count updates applied to the set. SADD Shown in Fig 14.



Fig.14. Execution of the 'SADD' command, inserting unique elements into a set to ensure exclusive membership.

SREM command removes specified elements from a set to optimize set membership handling. It can be safely run even if the specified elements don't exist as it will simply ignore them and not raise an error. Fig 15 Shows the SREM.

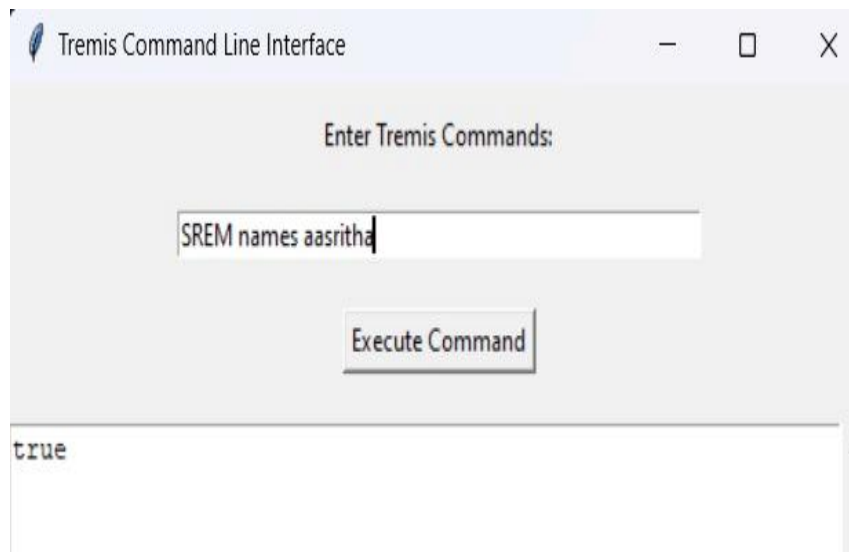


Fig.15. Execution of the 'SREM' command, removing specified elements from a set while maintaining efficiency.

The SMEMBERS command returns the entire set of members, providing a snapshot of its members at the time it is executed. This command is required to iterate over and view the unique members held within the set. Fig 16 Shows the SMEMBERS.

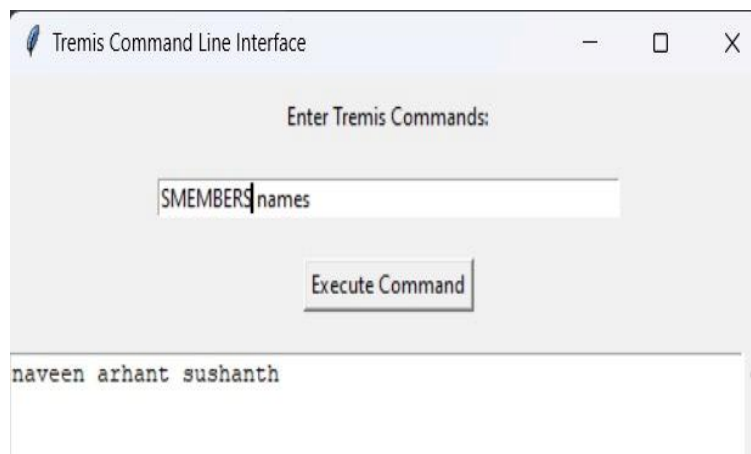


Fig.16. Execution of the 'SMEMBERS' command, returning all members of a set for complete membership inspection.

The SISMEMBER command checks if an element is a member of a set or not. It returns a binary value, yes or no, to represent if the element is a member of the set or not, and hence is a handy utility for membership testing. SISMEMBER Shown in Fig 17.

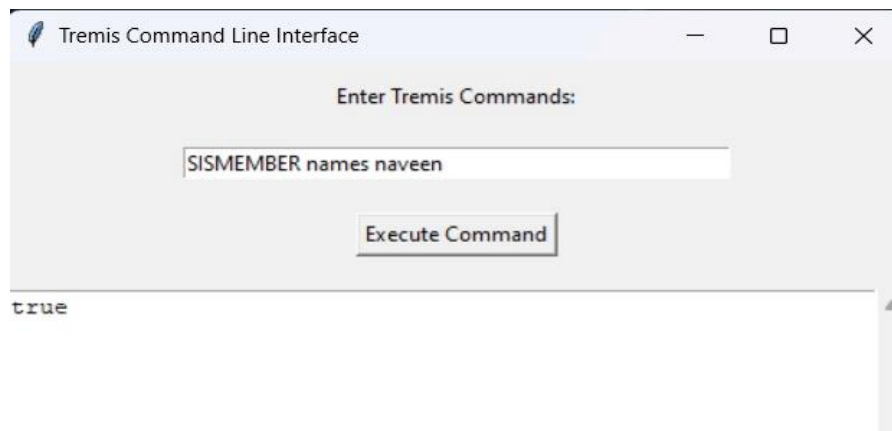


Fig.17. Execution of the 'SISMEMBER' command, verifying whether a specific element exists within a set.

Here subscribing makes a client subscribe to a specific channel. When a client subscribes, it will be receiving all messages published to the given channel. Several channels can be subscribed to by a client, and each subscription ensures that they will receive the updates concerning their interests. It is very effective because clients only receive notifications whenever messages are published in their subscribed channels, therefore reducing unnecessary communication overhead. This optimizes the use of resources by providing relevant information.

User engagement tools, such as subscription prompts, are crucial for improving channel exposure and increasing retention of your views. Fig 18 and Fig 19 show that the images of "SUBSCRIBE CHANNEL01" and "SUBSCRIBE CHANNEL02" are employed to catch attention and motivate a person to make decision, supporting the use of effective call-to-action intentions in digital media content.

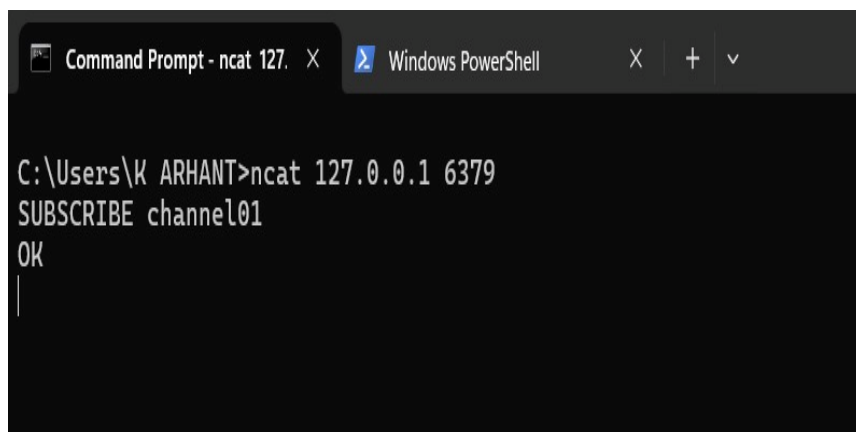
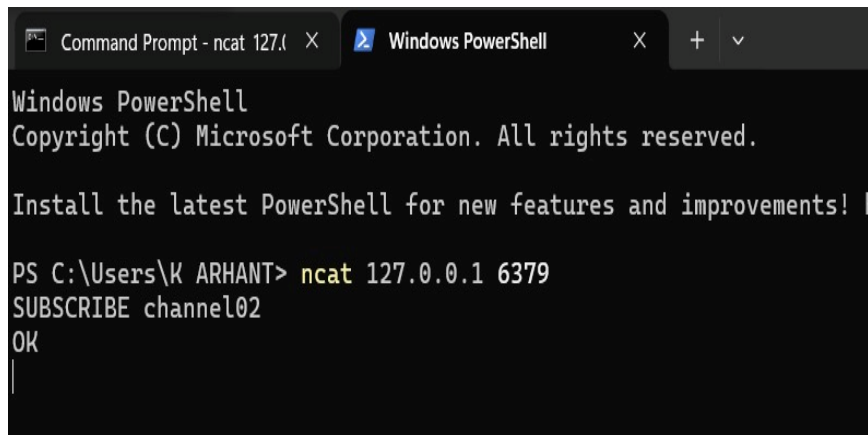


Fig.18. Example of the 'SUBSCRIBE' command, showing subscription to a communication channel ('CHANNEL01').

A screenshot of a Windows PowerShell terminal window. The title bar shows 'Command Prompt - ncat 127.0.0.1' and 'Windows PowerShell'. The terminal text reads: 'Windows PowerShell', 'Copyright (C) Microsoft Corporation. All rights reserved.', 'Install the latest PowerShell for new features and improvements! https://aka.ms/psinstall-latest', 'PS C:\Users\K ARHANT> ncat 127.0.0.1 6379', 'SUBSCRIBE channel02', 'OK', and a cursor on the next line.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

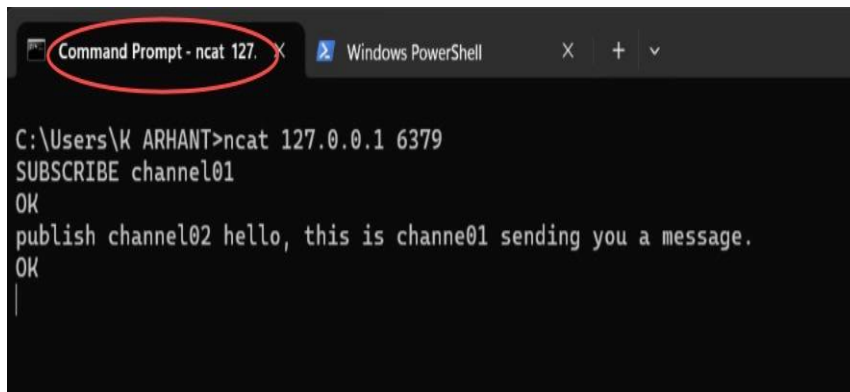
Install the latest PowerShell for new features and improvements! https://aka.ms/psinstall-latest

PS C:\Users\K ARHANT> ncat 127.0.0.1 6379
SUBSCRIBE channel02
OK
|
```

Fig.19. Example of the 'SUBSCRIBE' command, showing subscription to a second communication channel ('CHANNEL02').

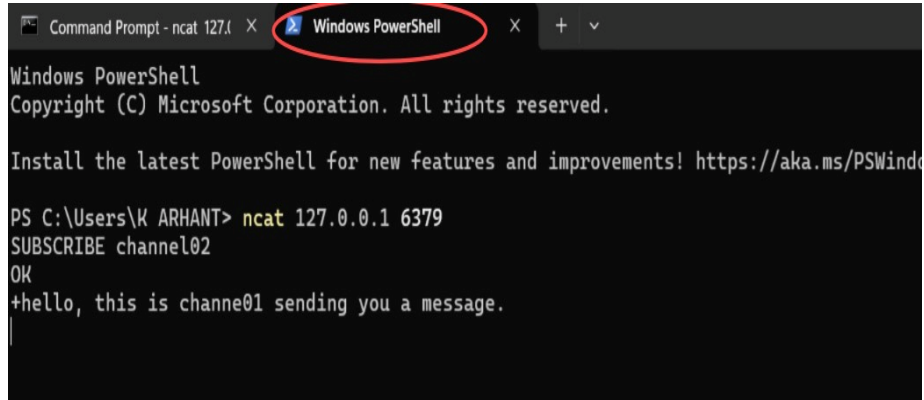
Here publishing is done by a publisher, publishing messages to an individual channel. Any channel subscriber will get the message immediately. It ensures messages sent in real-time, something essential for use like live notification, updates, and notifications. The publish operation is designed to be efficient and reliable so that the system can handle a high volume of messages and send them out quickly to all subscribers.

The message exchange of communication framework is visually explicated. Note that the contents are first broadcast to Channel 01 as illustrated by Fig. 20. Then, in Fig 21 we can observe that the Channel 01 effectively sends the message, showing that the the publish-subscribe communication mechanism works correctly in the system.

A screenshot of a Windows PowerShell terminal window. The title bar shows 'Command Prompt - ncat 127.0.0.1' (circled in red) and 'Windows PowerShell'. The terminal text reads: 'C:\Users\K ARHANT> ncat 127.0.0.1 6379', 'SUBSCRIBE channel01', 'OK', 'publish channel02 hello, this is channe01 sending you a message.', 'OK', and a cursor on the next line.

```
C:\Users\K ARHANT> ncat 127.0.0.1 6379
SUBSCRIBE channel01
OK
publish channel02 hello, this is channe01 sending you a message.
OK
|
```

Fig.20. Execution of the 'PUBLISH' command, broadcasting a message to subscribers of 'CHANNEL01'.

A screenshot of a Windows PowerShell terminal window. The title bar shows two tabs: 'Command Prompt - ncat 127.0.0.1' and 'Windows PowerShell', with the latter being the active tab and circled in red. The terminal content shows the PowerShell prompt 'PS C:\Users\K ARHANT>' followed by the command 'ncat 127.0.0.1 6379'. The output of the command is 'SUBSCRIBE channel02', 'OK', and a message: '+hello, this is channe01 sending you a message.'.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\K ARHANT> ncat 127.0.0.1 6379
SUBSCRIBE channel02
OK
+hello, this is channe01 sending you a message.
```

Fig.21. Successful delivery of a published message, demonstrating message flow in the publish–subscribe framework.

5 Discussion

In-memory databases are already commonly utilized due to their performance and efficiency with speed over the traditional disk-based system. Research into hybrid memory models, as indicated in comparable works, promises to lead to breakthroughs in the future. Blending DRAM and NAND flash has already been successful in terms of maintaining balance between performance, low cost, and low power consumption, serving as good reference points for future development.

While Tremis presently employs RAM with the facility of optional disk storage, integrating advanced memory management techniques like data migration, prefetching, and hybrid memory hierarchies could enhance its performance. These improvements would be capable of addressing issues like scalability, cost-effectiveness, and power efficiency, aligning Tremis with the demands of large-scale or resource-constrained applications. Besides, applying such approaches could render Tremis a competitive product that bridges the gap between high-speed operation and sustainable resource utilization.

This discussion highlights the necessity of continuously enhancing in-memory database systems to serve various user requirements through innovation from industrial practice and academic research.

6 Conclusions

This paper presented Tremis, an in-memory data store built in Go with support for persistence and a simple API. Our evaluation shows that Tremis delivers strong performance for fundamental operations and scales reliably under concurrent access. Future work includes adding replication, clustering, authentication, and durability enhancements, as well as expanding experimental comparisons with established systems such as Redis. Tremis thus provides a foundation for further exploration of lightweight, high-performance data management systems.

References

- [1] Makris, Antonios, et al." Database system comparison based on spa tiotemporal functionality." Proceedings of the 23rd international database applications engineering symposium. 2019
- [2] Fang, Jian, et al." In-memory database acceleration on FPGAs: a survey." The VLDB Journal 29 (2020): 33-59.
- [3] Lee, Donghun, et al." Improving in-memory database operations with acceleration DIMM (AxDIMM)." Proceedings of the 18th International Workshop on Data Management on New Hardware. 2022.
- [4] E. I. Chong, M. Perry and S. Das," Improving RDF Query Performance Using In-memory Virtual Columns in Oracle Database," 2019 IEEE 35th International Conference on Data Engineering (ICDE), Macao, China, 2019, pp. 1814-1819, doi: 10.1109/ICDE.2019.00197.
- [5] I. B. Peng, M. B. Gokhale, K. Youssef, K. Iwabuchi and R. Pearce," Enabling Scalable and Extensible Memory-Mapped Data stores in Userspace," in IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 4, pp. 866-877, 1 April 2022, doi: 10.1109/TPDS.2021.3086302.
- [6] O. Panchenko," In-Memory Database Support for Source Code Search and Analytics," 2011 18th Working Conference on Reverse Engineering, Limerick, Ireland, 2011, pp. 421-424, doi: 10.1109/WCRE.2011.60.
- [7] T. Lahiri et al.," Oracle Database In-Memory: A dual format in memory database," 2015 IEEE 31st International Conference on Data Engineering, Seoul, Korea (South), 2015.
- [8] Y. Wang et al.," The Performance Survey of in Memory Database," 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), Melbourne, VIC, Australia, 2015, pp. 815-820, doi: 10.1109/ICPADS.2015.109.
- [9] S. P. Dembele, L. Bellatreche, A. Lorusso, F. Marongiu and D. San taniello," In-Memory Database Query Energy Estimation: Modeling Green Strategy Support," 2023 IEEE World Conference on Applied Intelligence and Computing (AIC), Sonbhadra, India, 2023, pp. 278 285, doi: 10.1109/AIC57670.2023.10263900.
- [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] Yun, JT., Yoon, SK., Kim, JG. et al. Effective data prediction method for in-memory database applications. J Supercomput 76, 580–601 (2020). <https://doi.org/10.1007/s11227-019-03050>
- [12] Sai PC, Karthik K, Prasad KB, Pranav VS, Ramasamy G. Web-based real time chat application using MERN stack. InChallenges in Information, Communication and Computing Technology 2025 (pp. 195-199). CRC Press.
- [13] M. S. R., & Ramasamy, G. (2024). Heartbeat and scream detection system. Retrieved September 1, 2025, from <https://dx.doi.org/10.2139/ssrn.5091638>
- [14] Ojas O, Vashishtha A, Kumar Jha S, Kumar Shah V, Kumar R, Ramasamy G. James: Enhancing Judicial Efficiency with Smart Administration. Available at SSRN 5091509. 2024 Nov 15.
- [15] Lahiri, T., Neimat, M.A. and Folkman, S., 2013. Oracle TimesTen: An In-Memory Database for Enterprise Applications. IEEE Data Eng. Bull., 36(2), pp.6-13.