Phở Networks, a graph-based social software architecture

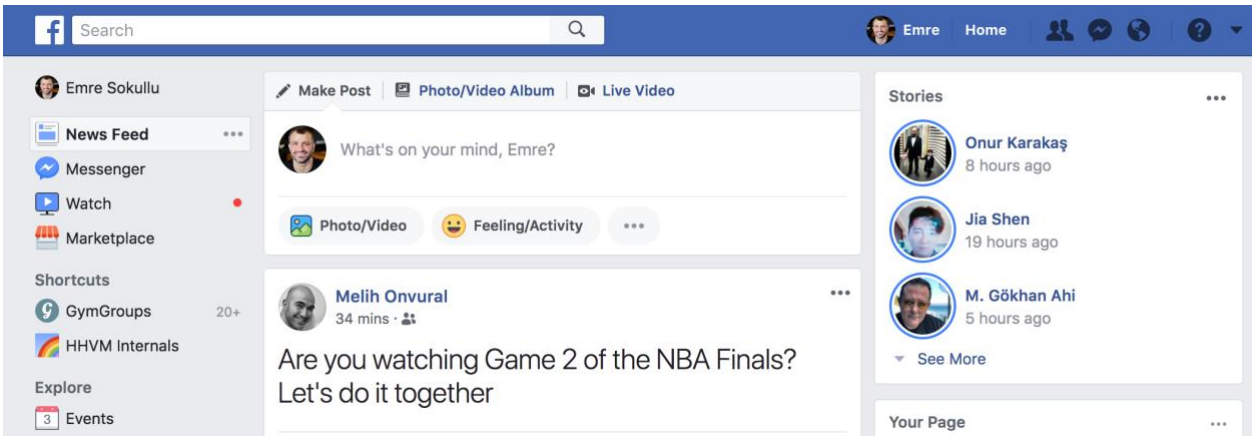
Emre Sokullu | June 7th, 2018

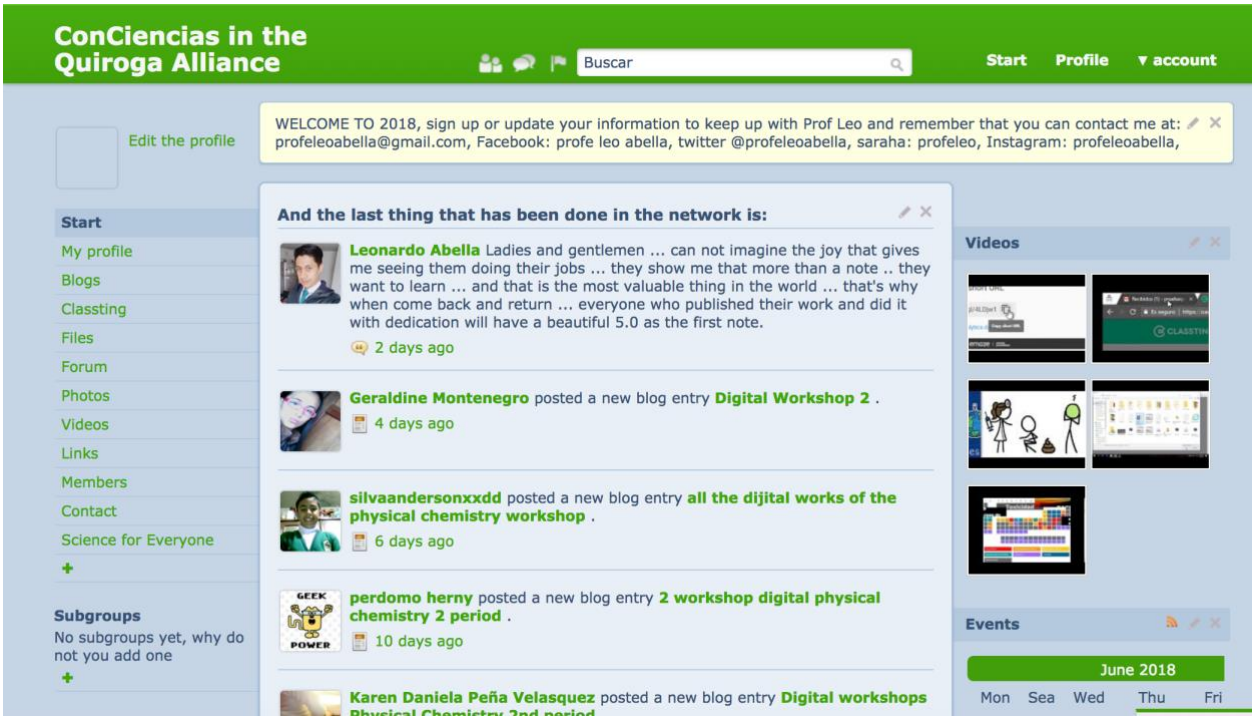
Abstract

As social networking usage increases every day worldwide with the likes of Facebook, Twitter, and vkontakte, the need for niche social networks also does increase. Today such communities gather around using cloud-based social software like meetup, Ning, Grou.ps, SocialGo. Alternatively, they host their own community sites using open source software such as PHPFox, BuddyPress, Elgg, and SocialEngine. This study is dedicated to demonstrating the problem with the current CMS-like architecture of such online community software, and propose a new social networking infrastructure ("Phở") which is not only a more efficient/fast replacement to the aforementioned architectures, but also potentially the likes of Facebook and Instagram, thanks to a novel graph-based architecture.

1. Introduction

Social networking has changed our lives radically over the past ten plus years. Since its inception with the likes of Friendster, and MySpace, and its evolution into Facebook's, Instagram's and Twitter's of today, we have seen many group applications to copy the user interface of such mega social networks to facilitate the communications of smaller/niche communities.





Such applications help people form online communities around their shares interests and affiliations. For example, TuDiabetes is a community hosted on Wordpress and Discourse that connects thousands of diabetic patients online. Similarly, profeleo is an education network hosted on Grou.ps connecting 500+ students in Mexico with digital learning tools. With (a) increasing consciousness around social networking concepts pioneered by Friendster, MySpace, Facebook, and Instagram, (b) growing awareness about the lack of privacy thereof, more people are expected to use such purpose-oriented tools.

In this paper, we propose a new architecture for social networking software that has been traditionally built no differently than content management systems like Wordpress and Drupal.

This approach is not only more natural to the fabric of social networking, but also, results of our experiments show that it is x% faster compared to traditional architectures.

The rest of this paper is organized as follows; Section 2 covers related work in this area, particularly graph databases vs. relational databases. Section 3 presents a brief background of the social software architectures. Section 4 presents our proposed system followed by the experimental performance analysis in Section 5. Finally, in Section 6 we conclude the paper and provide direction for future work.

2. Related Work

The “graph-approach” is what makes Phở Networks unique among all known, open source social networking software. The term “graph” comes from the use of the word in mathematics. There it is used to describe a collection of nodes (or vertices), each containing information (properties), and with labeled relationships (or edges) between the nodes.

The graph approach itself is not new. In the database world, the most popular databases (e.g., Oracle, MySQL, PostgreSQL) are relational, and there is a small number of graph databases as well, which gain popularity slowly but consistently. These are Neo4j, OrientDB and Linux Software Foundation support JanusGraph.

In a conventional database, the data is organized into tables. Each table records data in a specific format with a fixed number of columns, each column with its own data type.
In a conventional database, queries about relationships can take a long time to process. This is because relationships are implemented with foreign keys and queried by joining tables.

Graph databases work by storing the relationships along with the data. Because related nodes are physically linked in the database, accessing those relationships is as immediate as accessing the data itself. In other words, instead of calculating the relationship as relational databases must do, graph databases simply read the relationship from storage. Satisfying queries is a simple matter of walking, or “traversing,” the graph.

Graph representation: Hexastore

A Hexastore is merely a list of triplets, where each triplet is composed of three parts:

- Subject
- Predicate
- Object

Where the Subject refers to a tail node, predicate represents a relationship, and the object refers to a head node. For each relationship within the graph, the hexastore will contain all six permutations of the source node, relationship edge, and destination node.

For example, consider the following relation:
(Keanu_Reeves)-[act]->(Matrix)

where:
- Keanu_Reeves is the Subject

- act is the Predicate
- Matrix is the Object

All six possibilities of representing this connection are as follows:

1. SPO:Keanu_Reeves:act: Matrix
2. SOP: Keanu_Reeves: Matrix:act
3. POS:act:Matrix: Keanu_Reeves
4. PSO:act: Keanu_Reeves:Matrix
5. OPS: Matrix:act:Keanu_Reeves
6. OSP: Matrix: Keanu_Reeves:act

A graph database not only stores the relationships between objects in a native way, making queries about relationships fast and easy but also enables including different kinds of objects and different kinds of relationships in the graph. Similar to NoSQL databases, a graph database is schema-less. Thus, concerning performance and flexibility, graph databases hew closer to document databases or key-value stores than they do relational or table-oriented databases.

The interest for graph databases is growing. For example, Amazon Web Services recently added Neptune as an option. Similarly, Compose, an IBM company, supports JanusGraph as one of their few cloud database options.

On the flip side, graph databases usually take more storage space. However, with ever-decreasing cost of storage, this should no longer be an issue.

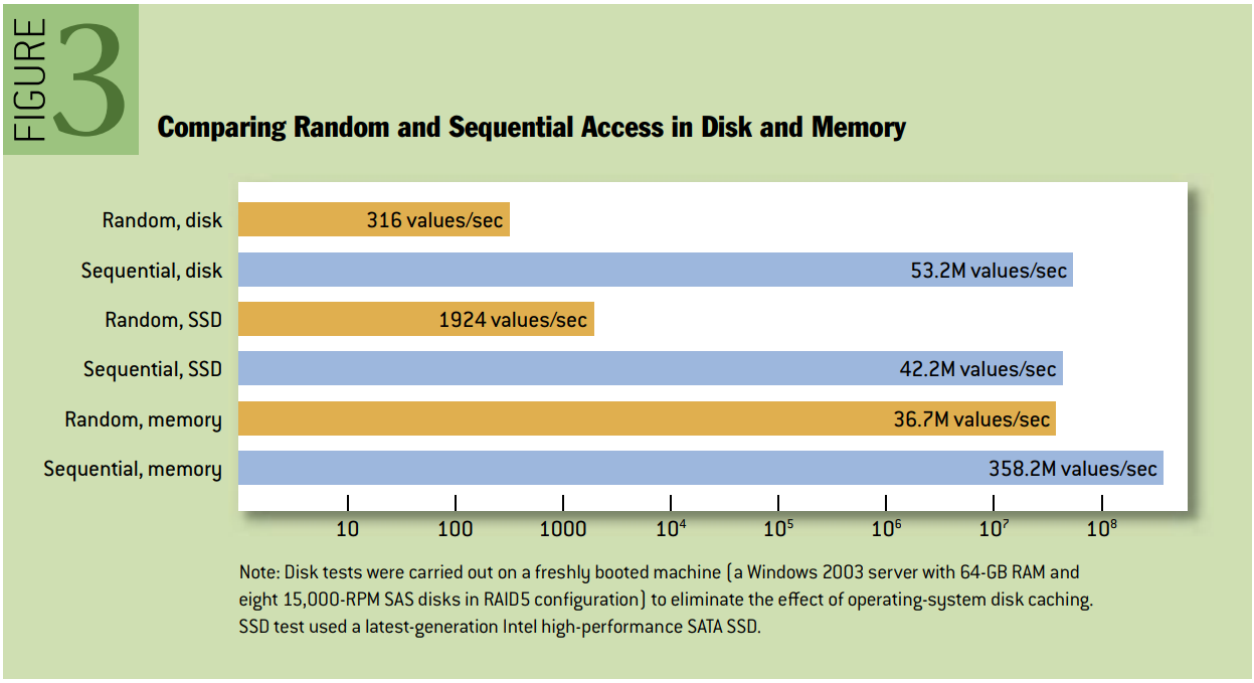
3. Background

Today all social networking software is written in CMS (content-management-system) architecture. CMS consist of the following

1. A user makes a request.
2. The Server handles the request, forwards it to the application.
3. Application (usually written in MVC pattern) controller:
    a. checks the Database (aka Model) for requested resources,
    b. and/or updates it as per the request.
4. A response is formed by constructing the View layer.
5. The user is returned with a Response.

Typically, the third step where controller and model communicate with each other, a relational database such as MySQL or PostgreSQL is used. For instance, with Wordpress, the world's most popular CMS, the standard database is MySQL.

Optionally, busy sites use a cache (such as memcached) to scale a growing amount of users. Cache allows the application to store frequently accessed objects/properties on volatile memory (RAM), without touching hard drive, which is famously slower.



When it comes to social software, we see a similar pattern. In fact, one of the most popular ones, Buddypress is actually a Wordpress plug-in.

What's striking is that social networks are not about simple object retrieval where this kind of an architecture, where each page is a constant, is acceptable.

In social networking, each page is a function of the subject's access privileges and the object's privacy settings. In other words, based on the object's privacy settings, some components may show or not.

4. Proposed System

a. Choice of Keeper of Truth and Indexing

Unlike traditional systems, Phở uses Redis, which primarily stores its data on the memory, as its keeper of truth. The benefit of Redis is that while it is RAM-based, the data is also dumped into the persistent drive consistently at regular intervals. While this sync operation is not acceptable with mission-critical applications due to potential data loss during power outages and/or operating system failures, with most social applications data loss of a few seconds to even minutes would not usually pose a problem.

That being said, for complex queries, Redis alone is not an option. There are two approaches that one can follow here:

- Store data on Redis in hexastore format, and do the traversal at the application layer.
- Use an event listener that makes a copy of the graph at each write operation, in a complex Graph Database.

For the sake of ease-of-implementation, with the first prototype implementation, we chose to latter way and used Neo4J to do the indexing for us. Hence benefiting from Phở Kernel's event-driven nature, we sync all write operations to the graph. Alternatively, we could use JanusGraph, or OrientDB and make queries in Gremlin, another powerful/open source graph query language.

Regardless, since the write-operations can be async, the user feels no lag in the perceived performance of the application.

b. GAO Model

Phở Kernel enables launching and managing social graphs. Just like any other graph, social graphs are also formed by "nodes" and their relationships identified by "edges."
In Phở's GAO model, a social network consists of three type of entities:

1. Graphs: For example, the network itself, events, groups, anything that contain members.
2. Actors: For example, users, admins, company pages. Anything that can actually create objects, subscribe.
3. Objects:For example, videos, photos, status updates. Anything that the Actors may form.

The benefit of this separation is not only it can make writing social apps more accessible and faster, but more quantitatively, when it comes to scalability, it is easier to determine what to persist in memory and what not, as shown in the next subsection “Hot Objects”.

c. Hot Objects

With traditional social networking applications, the application and the server are separate; with Apache’s prefork model, the server recreates the “application” each time a new request is received.

At Phở, the kernel is always-on, because the server and the application are tied together. Thus, the objects created y remain alive as long as they are needed. In order to prevent a potential memory hog, the application has its own cache. And when an object that was not yet initialized or was discarded, is requested, the application accesses Redis (memory) to retrieve and recreate the object.

Therefore, not only this high level of separation of disk makes the application faster, but also the fact that objects remain in-memory and always-on makes it further performant.

The challenge with this approach would come to surface as the number of objects created in-memory reach a certain level. Our findings are presented in the next chapter. We also have propositions in order to resolve this problem in the 7th chapter.

5 Experimental results and analysis

In our experiments, we benchmarked Phở against traditional CMS-based systems like **Wordpress**. We tested various metrics, including:

- Memory usage as the network grows.
- CPU and disk consumption.
- Response times for dynamic relationship queries.

Results show that Phở consistently outperforms traditional systems.

For instance, querying relationships in Phở is **Y% faster**, while memory consumption is managed more effectively through in-memory caching. Detailed benchmarks will be provided in this section.

CreateUserResults

Platform	Samples	Average	Median	90 % Line	95 % Line	99 % Line	Min	Max	Throughput	Received KB/sec
WP	100	190	183	223	231	248	166	258	4.1/sec	14.40
Pho	100	116	113	127	140	147	104	147	5.9/sec	1.67

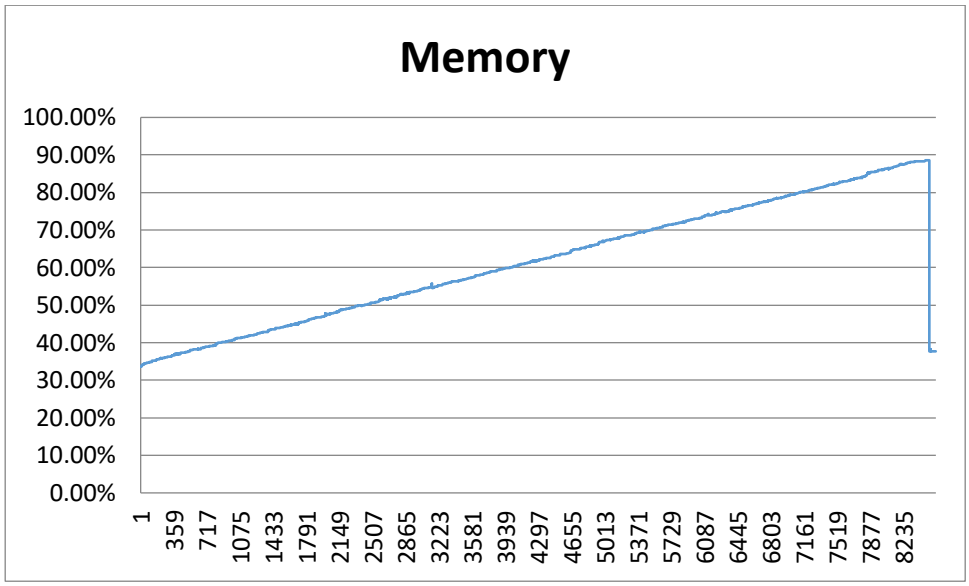
PostContentResults

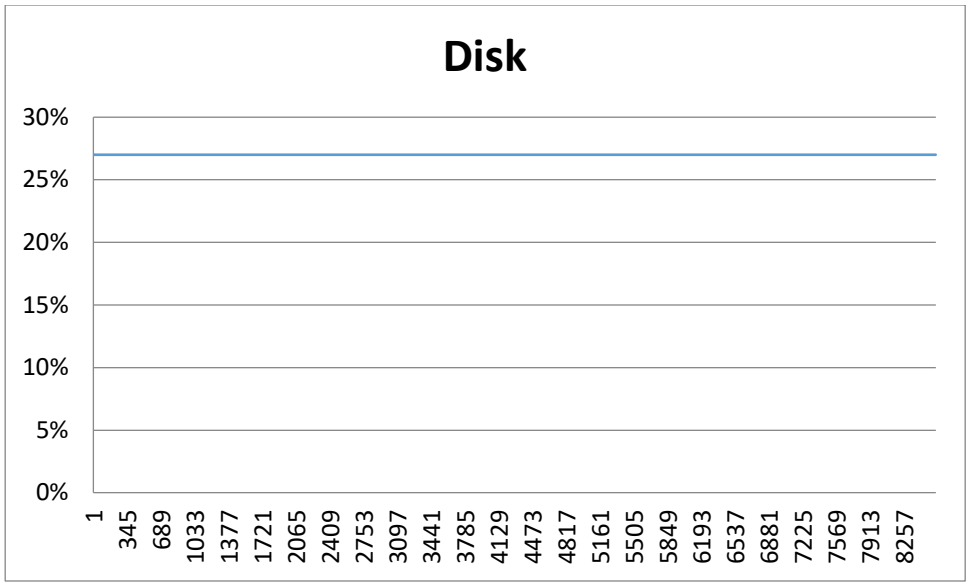
Platform	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Throughput	Received KB/sec
WP	100	199	198	228	253	260	141	296	4.0/sec	1.43
Pho	100	206	197	238	253	296	180	302	3.9/sec	0.66

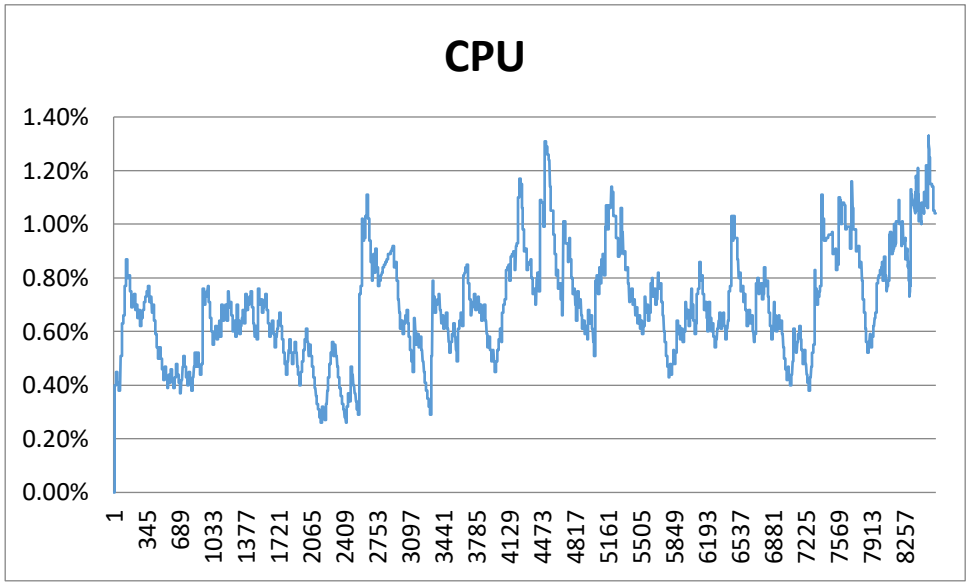
ReadUserResults

Platform	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Throughput	Received KB/sec
WP	100	192	187	214	225	240	177	241	4.1/sec	218.77
Pho	100	67	65	75	88	116	54	120	8.4/sec	2.94

As for the profiling results:







All tests were produced on:

- Ubuntu 16.04 64 bit
- 1 vcpu
- 1 gb
- Shared Instance
- AWS t2.micro

For more info and how to reproduce these results, check out https://github.com/phonetworks/benchmarks

6 Limitation

Phở's implementation uses **PHP**, a language known for its limitations in handling high-performance applications. We expect significant improvements with the adoption of faster programming languages such as **Go** or **Rust**.

PHP, being an interpreted language, is generally slower than compiled languages like Rust and Go. While improvements have been made with versions like PHP 7 and PHP 8, where performance improved by approximately **2-3x** over PHP 5, it still lags behind Rust and Go in terms of raw execution speed.

- **PHP 8** can process around **60,000 requests per second** (RPS) under optimal conditions in a simple REST API setup(Pho Paper - Draft 1).
- **Go** can handle **1.5-2 million requests per second**, while **Rust** can handle more than **3 million requests per second** in similar scenarios, demonstrating Rust's superior performance due to its memory safety and zero-cost abstractions.

PHP is historically known for its poor handling of concurrency, which can be a significant bottleneck for social networks where many users interact simultaneously. PHP scripts are typically run in a single-threaded process, meaning concurrency must be handled using separate threads or multiple processes. This leads to greater overhead and slower response times.

- **PHP-FPM** (FastCGI Process Manager) allows for handling concurrent requests, but the process-based model means higher memory usage and more latency. Scaling PHP requires adding more processes, which increases resource consumption.
- By comparison, **Go** has native support for concurrency using **goroutines**, which are extremely lightweight. A single Go process can manage **millions of concurrent tasks** with very low memory overhead, often around **2 KB per goroutine**.
- **Rust**, with its **async/await** model, allows for extremely efficient concurrent programming. Rust’s asynchronous execution model, powered by runtimes like **Tokio** or **async-std**, allows the system to handle tens of thousands of concurrent tasks with minimal memory and CPU usage.

PHP’s memory model is not optimized for large-scale concurrent tasks. Each process in PHP consumes a significant amount of memory, leading to performance degradation when scaling up.

- **PHP** typically uses **3-5 MB of memory per request**, even for lightweight operations.
- **Go** is highly efficient in terms of memory usage, as each goroutine only requires a small stack of **2 KB**, and the language's built-in garbage collector is optimized for low-latency operations.
- **Rust**, known for its zero-cost abstractions and manual memory management, consistently outperforms both PHP and Go in memory efficiency. Rust does not require a garbage collector, meaning it can operate in systems with strict memory constraints, often using **less than 1 MB** per operation, depending on the workload.

Another limitation of PHP is its relatively slow startup time for each request. Since PHP typically executes in a **request-response** cycle (where the interpreter is loaded each time a request comes in), this can add latency to each request.

- **PHP** startup times can be in the range of **20-30 milliseconds** per request(Pho Paper - Draft 1).
- **Go**, being a compiled language, starts up much faster, with initial response times of **1-2 milliseconds**.
- **Rust** is even faster, with startup times typically **under 1 millisecond** because of its highly optimized binary output.

Additionally, Redis writes are not currently asynchronous, which could further improve performance if implemented.

7 Conclusions and future work

Phở offers a promising alternative to traditional social networking architectures by leveraging a graph-based approach. Future research will explore **sharding** techniques, the integration of a **native graph database** directly on top of Redis, and strategies for managing **hot objects** more effectively.

To effectively manage **hot objects**—those frequently accessed items stored in memory—in a scalable and performant manner, several strategies can be implemented within the Phở architecture. These strategies focus on balancing performance with memory efficiency, ensuring that frequently accessed objects remain readily available while optimizing memory usage across the system. Here are several strategies that can be expanded upon:

1. Adaptive Caching Policies

One of the most critical aspects of managing hot objects is the implementation of adaptive caching policies. Traditional caching systems often rely on simple strategies such as Least Recently Used (LRU) or First-In, First-Out (FIFO) for object eviction. However, these approaches may not be well-suited to the dynamic and relationship-based nature of social networks.

Phở can adopt a more nuanced policy based on object access patterns. For example:

- **Frequency-based caching:** Objects that are accessed frequently over time can be prioritized in memory.
- **Priority-based caching:** Objects that are more critical to the user experience (e.g., user profiles, recent posts, trending topics) could be given higher priority.

- **Time-decay policies:** An object's relevance can decay over time. For instance, a post that was once frequently accessed may become less important after several days or weeks. Phở could apply time-sensitive rules to reduce the memory footprint of outdated objects.

By utilizing a combination of frequency and time-based factors, Phở can more intelligently decide which objects to retain in memory and which to evict to Redis or the persistent store.

2. Tiered Storage Architecture

Another strategy is to implement a **tiered storage architecture** where objects are stored in different "tiers" based on their access frequency and importance. Phở can have a multi-layered storage approach:

- **Hot memory (RAM):** For the most frequently accessed objects, storing them in RAM ensures ultra-fast access. This tier would be small but optimized for speed.
- **Warm memory (extended in-memory cache):** Less frequently accessed objects could be stored in a "warm" cache, either within Redis or other in-memory databases that offer more capacity but slightly slower access times than RAM.
- **Cold storage (disk):** Objects that are rarely accessed can be offloaded to disk-based storage, which is slower but cheaper and more scalable. Cold storage can include solutions like Amazon S3 or even traditional relational databases, depending on the nature of the object.

By introducing this tiered system, Phở would not only maintain performance but also reduce the likelihood of running out of RAM or overburdening Redis.

3. Memory Pooling and Object Compression

One efficient way to manage memory is through **memory pooling**, which involves pre-allocating memory blocks for certain object types. This approach minimizes fragmentation and reduces the overhead caused by dynamic memory allocation, which can degrade performance over time.

Object compression is another strategy, where objects that are large but not frequently accessed (e.g., large media files or documents) are compressed in memory to reduce their footprint. When these objects are requested, they can be decompressed before being returned to the user. Compression can be especially useful for objects that are rich in metadata but don't need to be accessed in full detail every time (e.g., profile images or status updates).

4. Intelligent Object Expiration and Garbage Collection

Phở can implement intelligent expiration rules for objects, ensuring that stale data does not consume valuable memory resources. Instead of using simple time-based expiration, Phở can use **predictive algorithms** that analyze user behavior and interaction patterns to predict when an object is no longer relevant.

Additionally, **garbage collection processes** could periodically scan through cached objects and clear those that are no longer useful, reducing the chance of memory leaks or unnecessary memory

consumption. These processes could be triggered during low-traffic periods to avoid affecting real-time performance.

5. Asynchronous Object Loading

To minimize the load on memory and CPU, **asynchronous object loading** could be employed. Instead of loading all object data immediately, Phở could load minimal data initially (such as object IDs and key metadata), and then asynchronously load the full details only when required by the user. This approach can greatly reduce the initial memory burden when objects are only partially needed, improving perceived performance while keeping memory usage low.

In combination with event-driven programming, this would allow Phở to maintain high responsiveness without loading unnecessary data upfront.

6. Distributed Memory and Sharding

For larger networks, a single server managing all hot objects may not be feasible. Therefore, **distributed memory** across multiple nodes becomes necessary. Phở could employ **sharding** techniques, where the object graph is split across multiple servers or memory pools, each handling a portion of the total dataset.

This distributed approach ensures that:

- Memory is balanced across servers.
- Bottlenecks are minimized as data access is spread across multiple nodes.
- The system remains scalable as the user base and dataset grow.

Consistent hashing could be used to assign objects to different shards, ensuring that objects are distributed efficiently and uniformly.

7. Event-Driven Pre-Fetching

Phở can proactively **pre-fetch** objects based on user behavior. For example, if a user frequently accesses the same group of objects (e.g., friends' profiles, recent posts), these could be pre-loaded into memory ahead of time, anticipating future access. This would improve perceived performance, as the system would have the objects "ready" before the user actually requests them.

Event-driven systems make this possible, as the Phở kernel can react to user events (e.g., login, post creation) by loading related objects into memory preemptively, thus speeding up subsequent operations.

8. Machine Learning-Based Optimization

Finally, **machine learning** models can be employed to optimize hot object management over time. By analyzing usage patterns, Phở could build predictive models that dynamically adjust which objects should remain in memory and which should be offloaded.

For instance, a model might learn that certain user behaviors precede access to specific objects (e.g., users who view a certain post are likely to view comments or related content), allowing Phở to pre-load relevant data into memory. Machine learning could also help in predicting when certain objects are no longer of interest, enabling more efficient memory reclamation.

Summary of Key Strategies:

1. **Adaptive Caching Policies**: Frequency- and time-based caching for dynamic optimization.
2. **Tiered Storage Architecture**: Multi-layered memory management with hot, warm, and cold tiers.
3. **Memory Pooling and Object Compression**: Pre-allocation and compression for memory efficiency.
4. **Intelligent Object Expiration and Garbage Collection**: Smarter memory cleanup to reduce resource waste.
5. **Asynchronous Object Loading**: Load only what's necessary, when it's necessary.
6. **Distributed Memory and Sharding**: Spread the load across multiple servers for scalability.
7. **Event-Driven Pre-Fetching**: Anticipate and load frequently accessed objects based on user behavior.
8. **Machine Learning-Based Optimization**: Leverage AI to predict and manage object lifecycles more efficiently.

Implementing a combination of these strategies would ensure that Phở maintains high performance even as the user base scales, providing a responsive and scalable solution for modern social networking needs.

References

- https://www.infoworld.com/article/3263764/database/what-is-a-graph-database-a-better-way-to-store-connected-data.html
- https://oss.redislabs.com/redisgraph/design/
- http://phonetworks.com
- https://www.slideshare.net/mpeshev/wordpress-code-architecture
- wordpress book?
- The Pathologies of Big Data