GIGASPACES | WRITE ONCE. SCALE ANYWHERE.

# Designing a Scalable Twitter with Space-Based Architecture

By Nati Shalom, CTO

IN THIS PAPER



Everybody knows Twitter – the popular online service that allows people to send short messages to their friends and colleagues. If you think about it, Twitter exposes a few very difficult scalability problems that are common to most real-time or social web applications. If you can design a scalable Twitter, you're already half-way to solving the scalability issues of most modern web applications. This paper shows you how to do just that.

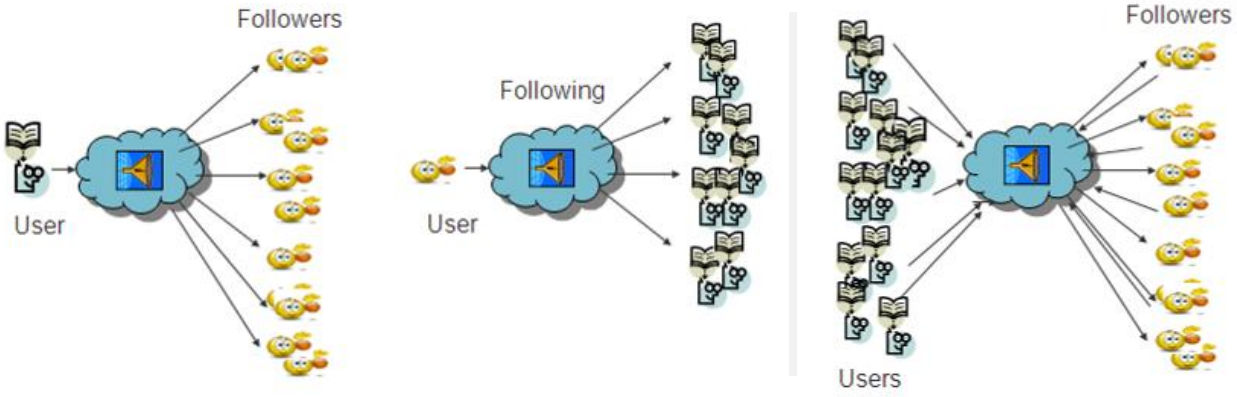
**Disclaimer:** This paper is a hypothetical case study. It is not intended to accurately portray the actual Twitter web application, rather it uses publicly-available details about Twitter's features and behavior, infers Twitter's scalability challenges from these details, and proposes solutions. The analysis may not be correct for the actual Twitter website and may not portray the full complexity of Twitter's problem. The intention is to show generic challenges and solutions that are applicable to most modern web applications.

## TWITTER'S SCALABILITY CHALLENGE

In Twitter, the primary relationship between entities is many-to-many. Every post is sent to numerous followers of the user who sent the post; at the same time, each user can follow many other users. This causes Twitter to behave like a living organizm, growing unexpectedly in many different directions.

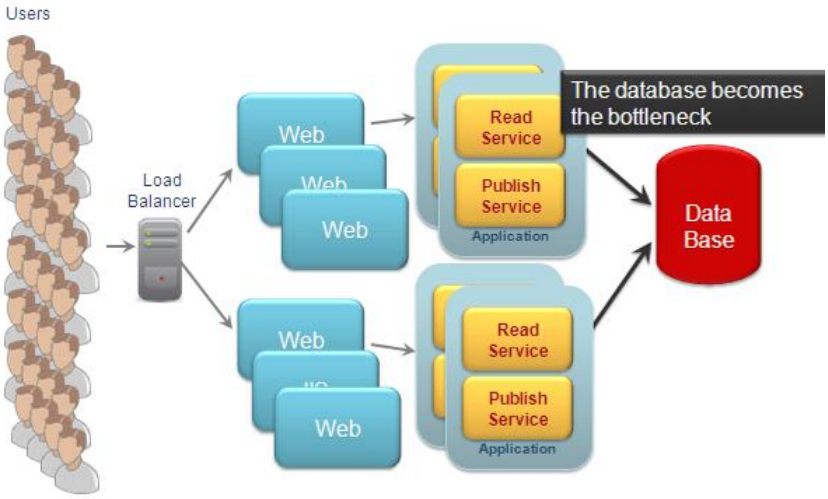
I'll focus on what I perceive to be Twitter's biggest scalability problems:

- **Fluctuating growth** – networks of users and followers constantly grow, traffic tends to fluctuate randomly.
- **The crowd effect** – the retweet feature, which allows users to re-send interesting messages to their own followers, can lead to a "perfect storm". A particularly interesting tweet can spark a sudden, massive and unpredictable increase in traffic.



## TRADITIONAL SOLUTION: SCALING BY ADDING WEB CONTAINERS

The most common and obvious way to scale Twitter is to front the web application with a load balancer, which diverts traffic to one of several web servers/containers. However, as many of us have come to know, this approach has its limitations.




As the application grows, more and more web containers are added, but there is still one central database. At some point, the database cannot handle the load, and the application cannot scale further without a change.

## SOLVING THE DATABASE BOTTLENECK: NOSQL AND MEMORY-BASED SOLUTIONS

It's clear that a change is necessary to solve the scalability problem, but what sort of change? Keeping the existing database, and simply clustering it to make it more scalable, or getting rid of the database alltogether?

### NoSQL and the Fallacy of Disk Reliability

Two of the biggest assumptions behind the use of databases in applications are being strongly challenged today. The first is the use of a traditional SQL database. The new and growing NoSQL movement, influenced by Internet industry leaders like Google and Amazon, claims that **simple key/value stores running on commodity hardware** and designed for massive scale, are the modern replacement for the database. One of the important principles of the NoSQL movement is that **failure is inevitable**, and we should build systems that reduce the impact of failure.[1]

The second assumption is that disk is a reliable form of storage. Many organizations take it for granted that their data is "safe" if it is stored on a disk. But according to empirical research, physical disks are not as reliable as many of us think. Actual disk failure/year as measured in the field is up to **600% higher than failure estimates** provided by vendors (3% actual failure vs. 0.5-0.9% estimated). Amazingly, there is no correlation between type of disk (SCSI, SATA, fiber channel) and failure rates – high-end hardware is not more reliable. And lower disk temperatures are actually associated with ***higher*** failure rates.[2]

---

[1] For more on the common principles behind NoSQL solutions, and their relevance for mainstream applications, see Nati Shalom's Blog (natishalom.typepad.com), The Common Principles Behind NOSQL Alternatives.

[2] For a review of this literature and its implications, see Nati Shalom's Blog (natishalom.typepad.com), Why Existing Databases (RAC) are So Breakable.

### Pros and Cons of SQL and NoSQL Solutions[3]

| | Sample Solutions | Change Required | Scalability | Performance & Latency | Cost |
|---|---|---|---|---|---|
| **SQL** | **MySQL database clustering** | Moderate, across the board (*) | Only affects read scalability | No change | Moderate storage cost (**) |
| **NoSQL (disk-based)** | **BigTable, Dynamo, Cassandra, etc.** | Complete re-write | Read/write scalability | Moderately improves read/write | Low storage cost |
| **NoSQL (in-memory)** | **Memcached** | Moderate, application only | Only affects read scalability | Improves read performance only | High storage cost (***) |
| | **GigaSpaces XAP In-Memory Data Grid, Oracle Coherence, etc.** | Moderate, application only | Read/write scalability | Highest read/write performance | High storage cost (***) |

(*) In most database clustering schemes, there is a master and several read-only replicas. Contrary to conventional thinking, this is not transparent – applications need to be modified to write to the master and read from one of the replicas. This change needs to happen across the board because the database is shared by many applications, including back-end, analytics, etc.

(**) Disk has a low cost of storage compared to memory, but database clustering solutions are based on expensive high-end hardware, which drives up costs.

(***) While memory is much more expensive than disk, according to a recent Standford University study, in-memory data grids "become much more attractive for applications with high throughput requirements. When measured in terms of cost per operation or energy per operation, [data grids] are 100-1000x ***more*** efficient than disk-based systems".[4]

### What Should We Use to Scale Twitter? The Best of Both Worlds

Twitter sets an upper limit of 150 req/hour per user. For 1 million active users, that means 40,000 req/second. Assuming 128 bytes per post, the data feed is over 5MB per second, or 432GB for just one day of data. There are two conflicting requirements: On the one hand, it simply **isn't possible to deliver 40K req/second on a physical disk**, pushing us in the direction of memory-based solutions which provide lower latency and higher performance. On the other hand, the **total volume of data is huge**, so the cost of memory storage will become prohibitive.

But these requirements **aren't really conflicting**. If you think about it, only tweets from the last 10-30 minutes really need to be accessed in real time (the rest can be considered "history" and pulled in the background from the disk). Assuming 20% of requests are writes, the real-time data is, at most: 40K * 20% * 128 = only 1MB/sec. To be on the safe side and get a one-hour buffer of real-time data, **5GB of memory storage** is plenty. This is a reasonable size which permits the use of memory-based solutions.

The proposed solution, therefore, is to take the best of both worlds – to use **an in-memory data grid for real-time access**, and **a traditional database for long term storage**, search and other analytics requirements (making the change transparent to all back-end applications).

## HOW TO BUILD A SCALABLE TWITTER

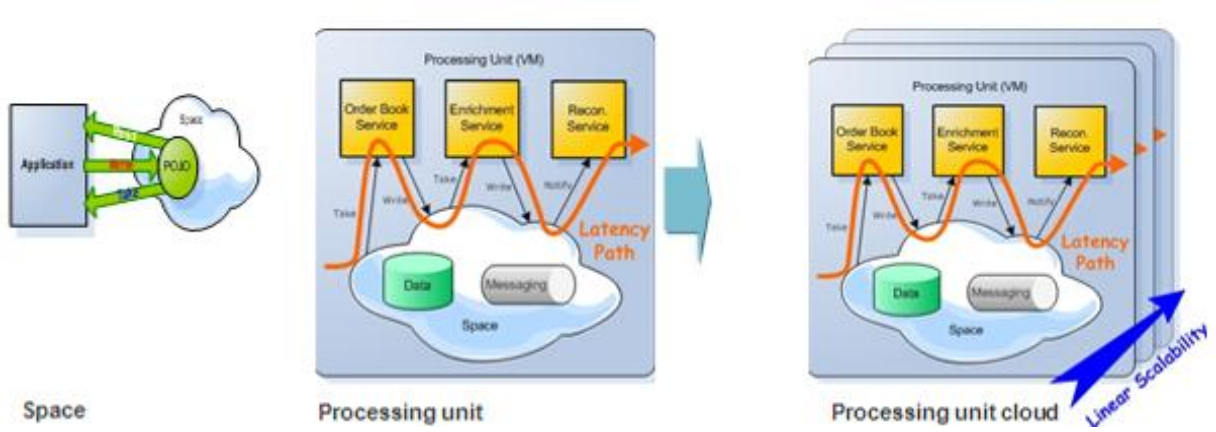
To build a scalable Twitter, I propose to use **Space-Based Architecture**, a software architecture pattern for achieving linear scalability of stateful, high-performance applications, which has its roots in Yale's Tuple-

[3] To learn more about these different storage options, see Nati Shalom's Blog (natishalom.typepad.com), The Common Principles Behind NOSQL Alternatives.

[4] Ousterhout et. al., The Case for RAMClouds: Scalable High Performance Storage Entirely in DRAM, p. 12, emphasis in original.

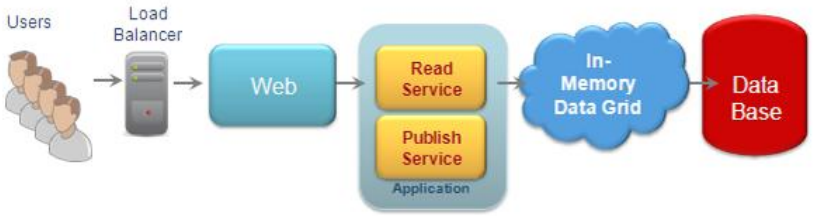
Space Model. The primary components in this architecture are:

- **The space** – a shared memory unit with 4 simple APIs, which permits data sharing, messaging, workflow management and parallel processing.
- **The processing unit** – a bundle of services, data and messaging components, collocated into a single virtual machine and running in-memory. Duplicating this processing unit achieves linear scalability.




## Removing Database Bottlenecks with In-Memory Data Grid and NoSQL

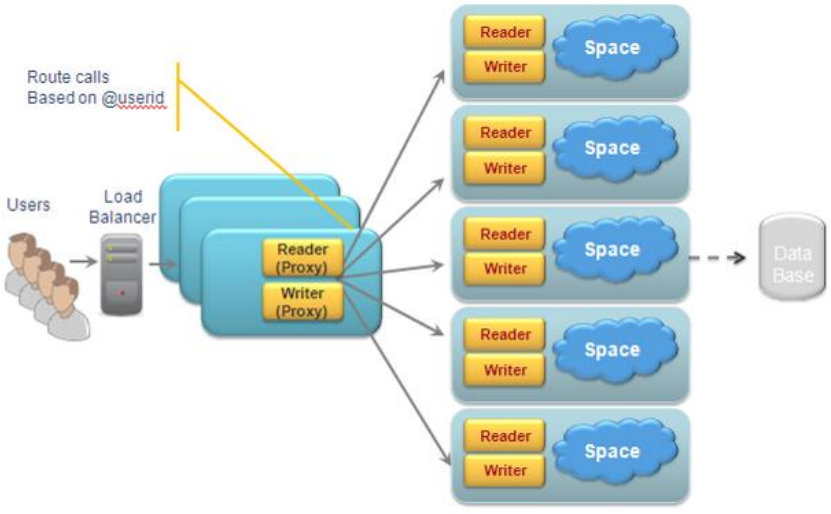
As a first step, putting an in-memory data grid between the web server and the database immediately releases the database bottleneck. As the application grows, the in-memory data grid scales up, and the database is used in the background, for long-term storage only. This is a NoSQL solution because the primary storage medium is not a database.




## Achieving Linear Scalability with Partitioning and Collocation

The next step is to bundle the read and publish services into a **processing unit**, together with a **space** which provides in-memory data grid and other capabilies.

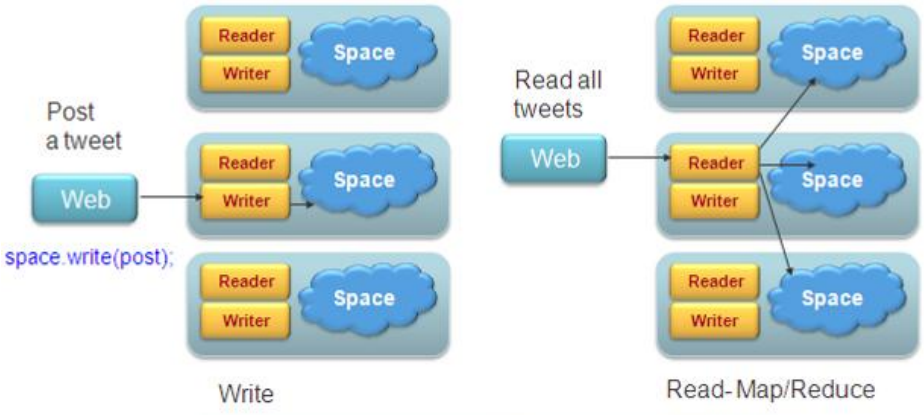
Each processing unit is a partition which holds a subset of the tweets users send. An important question is how to divide the tweets between partitions: there are a few possible approaches, the best one seems to be **partitioning by Twitter user ID**. This way each partition holds all the tweets for a certain number of users. When the number of users grow, the processing units can be duplicated as many times as necessary, without limitation.



Using GigaSpaces XAP, it is even possible to add **dynamic scalability**, meaning that processing units are launched or terminated on-demand, based on monitoring of the number of active users currently on Twitter.[5]

### Scaling Reads and Writes

To post a tweet, the Writer service simply **"writes" it to the space**, and it is instantly routed to the appropriate partition. To read all tweets, the Reader service uses **Map/Reduce** – tweets are collected from all partitions and returned to the service in aggregated form.




**>> To see this read/write strategy in more detail**, with code samples, see Nati Shalom's Blog (natishalom.typepad.com), Designing a Scalable Twitter.

## SUMMARY: BENEFITS OF SPACE-BASED ARCHITECTURE

Twitter is a great example of how difficult it can be to scale web applications that have a real-time or social networking aspect. This paper argued that applications like Twitter should take heed of the NoSQL trend and the rise of in-memory storage, but not do away with the database altogether. Using Space Based Architecture, the application can run real-time queries against a fast, in-memory, NoSQL component like an in-memory data grid, while using the database for long-term storage.

This solution would provide Twitter with unlimited, linear scalability, allowing it to grow on-demand without the need for re-architecture, and easily scale back when needed. It is relatively non-intrusive – based on a simple programming model and a single clustering model, and backward-compatible with the SQL world. Most importantly, unlike many NoSQL solutions, it does not compromise on consistency and reliability, ensuring a high level of resilience which the SQL world has promised – and often failed – to deliver.




**U.S. Headquarters**
GigaSpaces Technologies Inc.
317 Madison Ave, Suite 823
New York, NY 10017
Tel: 646-421-2830
Fax: 646-421-2859

**International Office**
GigaSpaces Technologies Ltd.
4 Maskit St., P.O. Box 4063
Herzliya 46140, Israel
Tel: +972-9-952-6751
Fax: +972-9-956-4410

**Europe Office**
GigaSpaces Technologies Ltd.
2 Sheraton St.
London, W1F 8BH, United Kingdom
Tel: +44-207-117-0213
Fax: +44-870-383-5135

**Asia Pacific Office**
GigaSpaces Technologies Ltd.
Level 21, Centennial Tower
3 Temasek Avenue
Singapore 039190
Tel : +65-6549-7220
Fax: +65-6549-7001

Corporate Site: www.gigaspaces.com
Team's Blog: blog.gigaspaces.com
Community Site: www.openspaces.org


[5] For a brief tutorial showing how this works in GigaSpaces XAP, see the product documentation (www.gigaspaces.com/wiki), Scaling Your Web Application.