GIGASPACES

WRITE ONCE.
SCALE ANYWHERE.

Designing a scalable twitter

Nati Shalom,

CTO & Founder GigaSpaces

John D. Mitchell

Mad Scientist of Friendster.

About GigaSpaces Technologies

Enabling applications to run a distributed cluster as if it was a single machine…

75+ Cloud Customers

Among Top 50 Cloud Vendors

300+ Direct Customers

Dias nummer 2

a2

1. Replaced the number of deployments (+2000) with the number of Cloud customers.
- Details about the specifics of the numbers are in the notes below.

2. added logo's of Cloud Customers and partners (included Spring in there) at the bottom part of the slide

alit; 23-08-2009

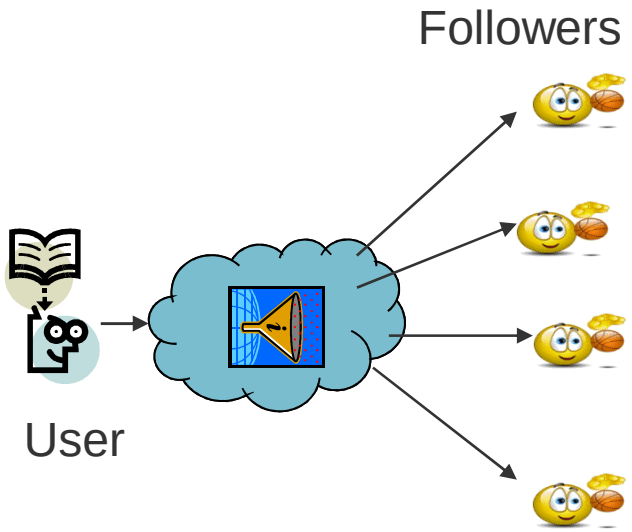
Introduction – Why twitter as an example

- Everybody knows twitter
- Twitter exposes really difficult scaling challenges
- Twitter can serve as a generic example for any real-time/social web application
- If you can built a scalable twitter your already half way through.

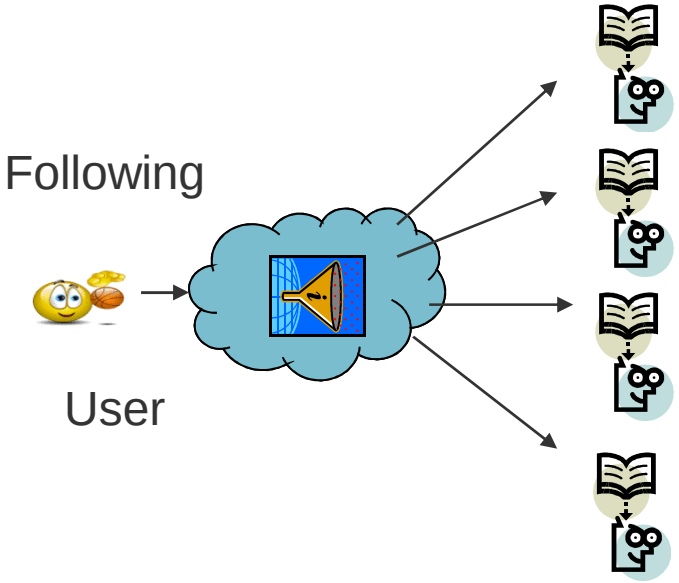
Why twitter is different then other web apps?

- Twitter makes many to many relationship first class citizen
- Twitter behaves like a living organism

Every post gets to number of followers

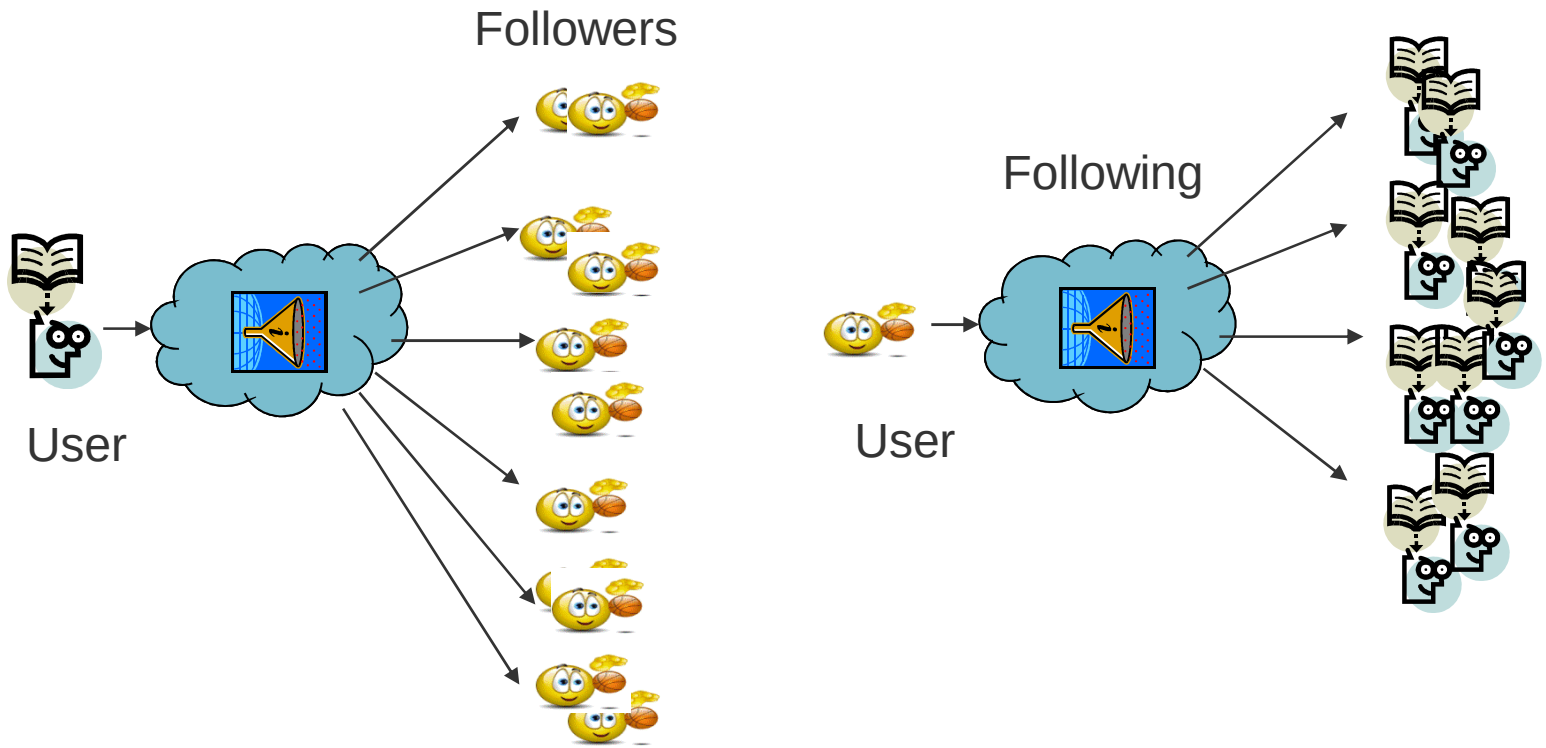


Every user follows number of users



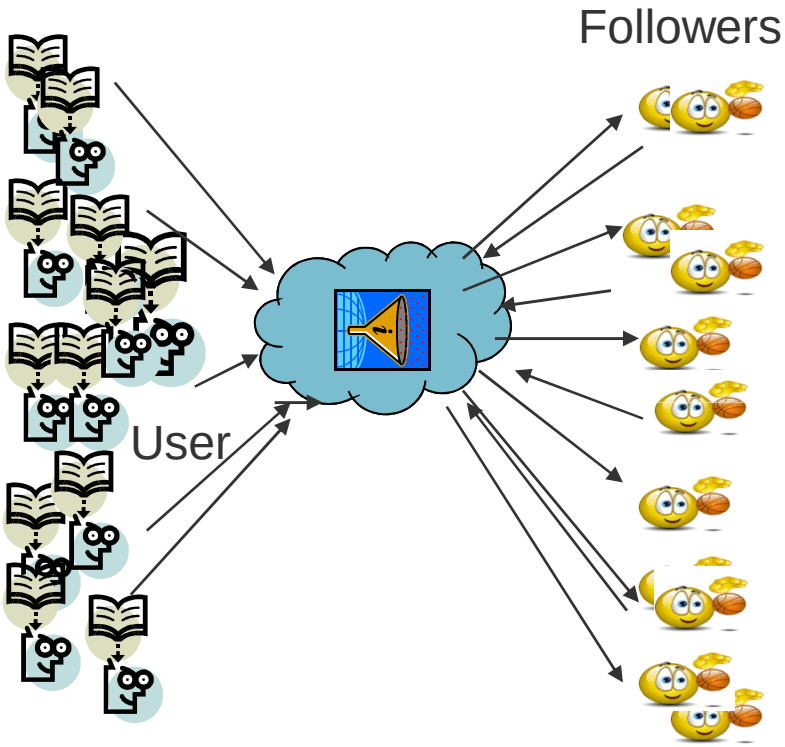
Scaling challenges 1 - Fluctuating growth

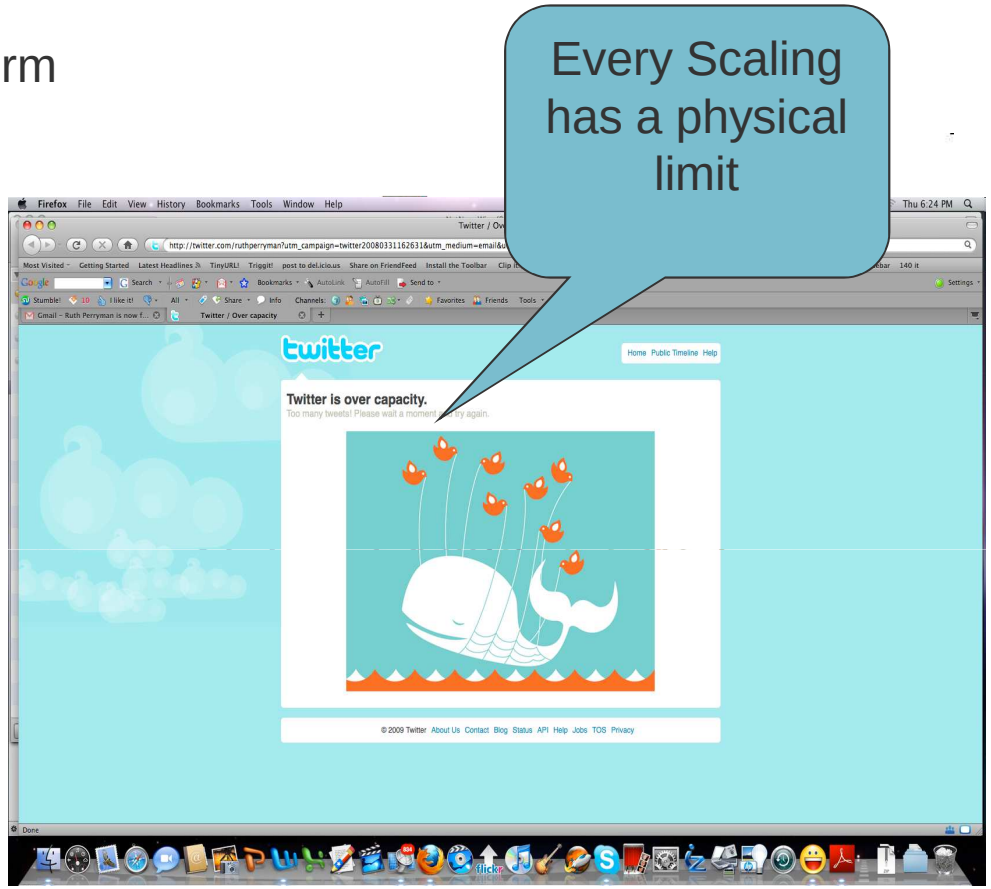
- Every user network (Following/Followers) grows constantly
- Traffic tends to fluctuate randomly



Scaling challenges 2 – The crowd effect

- Re-tweet can lead to a perfect storm



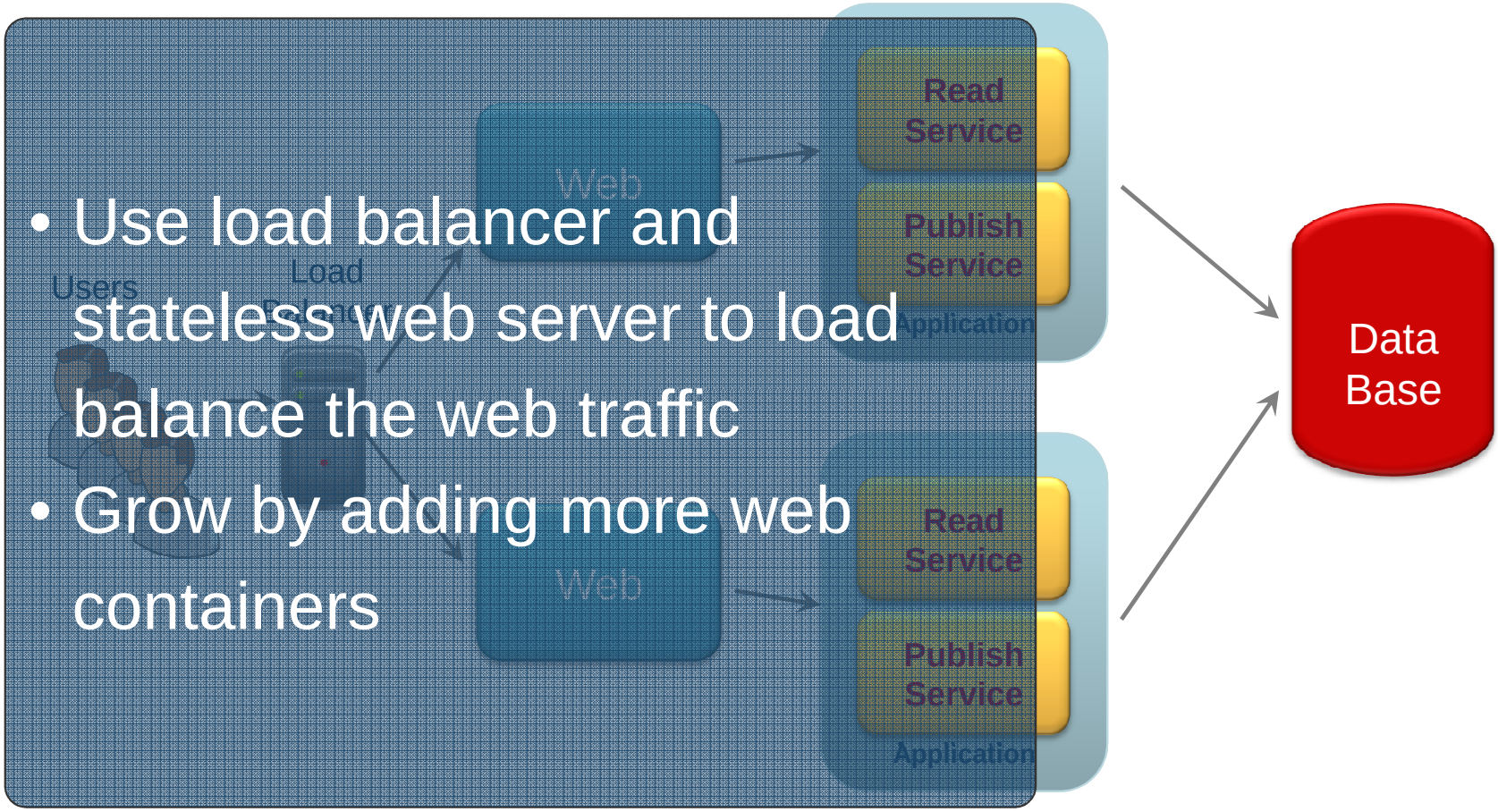


Source: Twitter.com

Designing a scalable twitter

Approach 1 – Use share nothing web scaling approach

- Use load balancer and stateless web server to load balance the web traffic
- Grow by adding more web containers



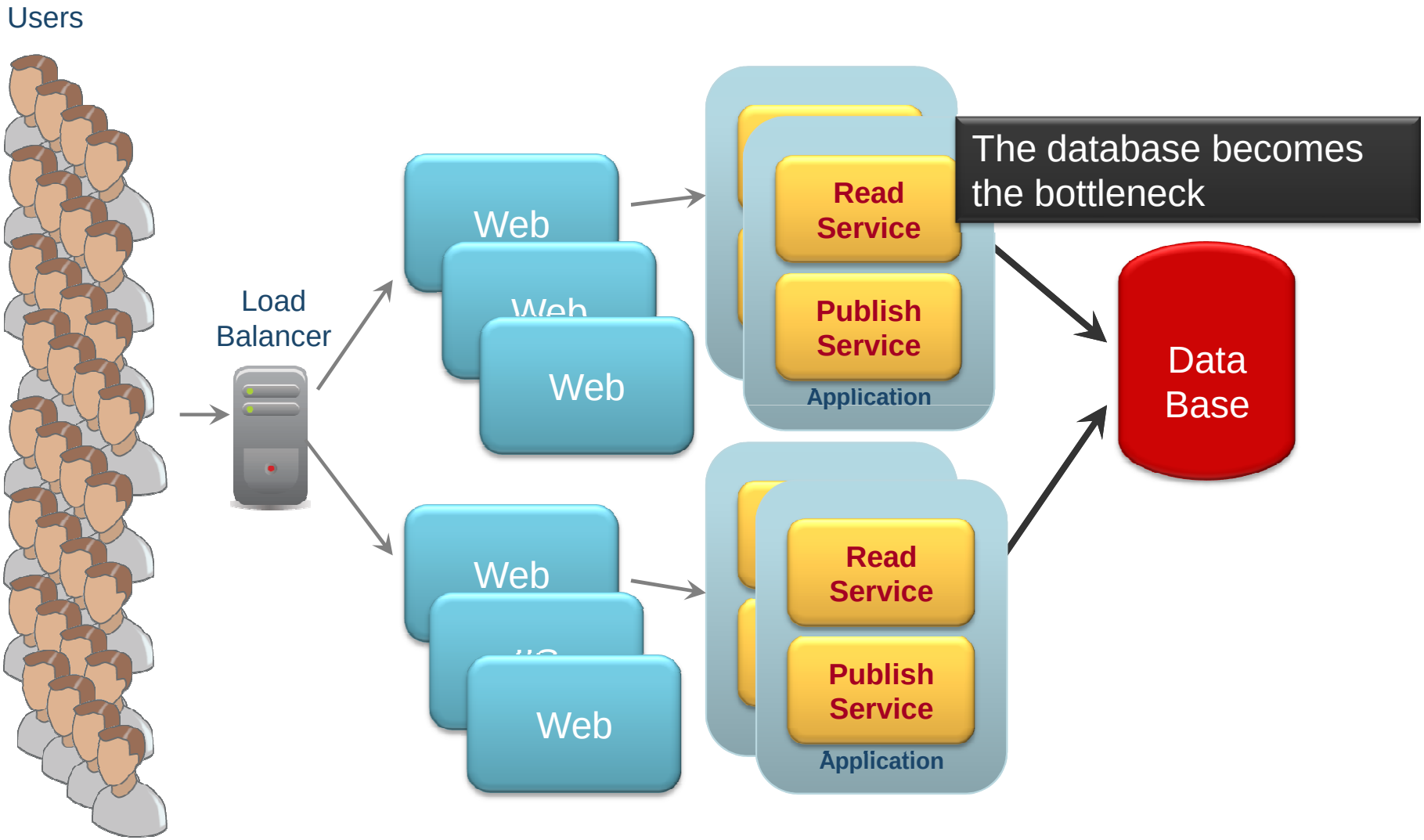
Reader/Publisher Service

```
namespace MicroBlog.Services
{
   public interface IReaderService
   {
            ICollection<Post> GetUserPosts(String userID);

            ICollection<Post> GetUserPosts(String userID, DateTime fromDate);
   }
}
```

```
namespace MicroBlog.Services
{
   public interface IPublisherService
   {
            void PublishPost(Post post);
   }
}
```

Approach 1 scalability challenge



Reader – Database Implementation

```
public ICollection<Post> GetUserPosts(string userID, DateTime
fromDate)
   {
          // Create command:
          IDbCommand dbCommand =
                    _dbConnection.CreateCommand();
                    dbCommand.CommandText = String.Format(
                              "SELECT * FROM Post WHERE
                              UserID='{0}' AND PostedOn > {1}",
                              userID, fromDate);

                         // Execute command:
          IDataReader dataReader = dbCommand.ExecuteReader();

          // Translate results from db records to .NET objects:
          List<Post> result = ReadPostsFromDataReader(dataReader);

          // Return results:
          return result;
}
```

Eliminate database bottlenecks – a questions of alternatives

- Solving the database bottleneck requires change
- Difference approaches offers various degree of change
  - Database clusters- read-only databases
    - Requires significant data access design changes
    - Does not improve performance In Memory DataGrid – Change at the application data not the database
  - Memcache – Small change/Small value
    - fairly limited not transactional, doesn't support SQL query, non clustered
  - Google App Engine Persistency (On top of Big Table)
    - Provides standard JPA access (Smaller change)
    - Many model limitations
    - Does not provide database integration into existing model
  - Application Partitioning (several different databases for the same application)
    - Requires significant data access design changes
    - Need to synchronize between the databases for transactional integrity

The NOSQL movement

- Distributed key/value store (#NOSQL) –
  - New hot trend, influenced by Google, Amazon..)
  - Use Distributed commodity HW then expensive HW
  - Designed for massive scale
  - Examples
    - FIlesystem based implementation
      - Amazon Dynamo/SimpleDB
      - Google Big Table
      - Casandra
    - Memory based implementation
      - GigaSpaces
      - Gemstone
      - IBM extremeScale
      - JBoss infinispan
      - Oracle Coherence

Did you know?

- *Disk failure/year - 3% vs. 0.5 - 0.9% estimated*
- *No correlation between failure rate and disk type - SCSI, SATA, or fiber channel*
- *Lower temperatures are associated with higher failure rates*

NOSQL common principles

- Assume that failure is inevitable
  - Disk, machine, network will fail
  - Don’t avoid it (through costly HW) - cope with it
- Reduce the impact of failure by:
  - Keeping multiple replicas
  - Distributing the data (partitioning)

- CAP Theorem
  - Relax some of the consistency constrains – eventual consistency

- Use Map/Reduce to handle aggregation
  - Parallel query
  - Execute the query close to the data
- Available as Filesystem or In-Memory implementation

In-Memory vs Filesystem based approaches

- In-memory
  - Pros:
    - Complementary – the database can be kept unchanged
    - Designed for real time performance (Not limited to disk I/O)
    - Enable execution of the code with the data (stored procedure)
  - Cons
    - Memory is more expensive then file-system (Cost of per GB storage)
- File-System
  - Pros:
    - Can manage terra bytes of data
    - Low cost per GB storage
  - Cons
    - Fairly radical change -> requires complete re-write

- Best of both worlds
  - Memory based storage for real time access
  - Filesystem for long-term data

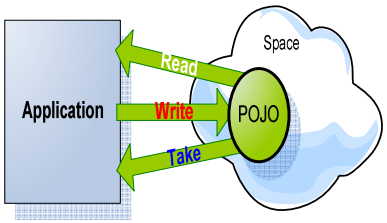
Choosing the right solution for our twitter app

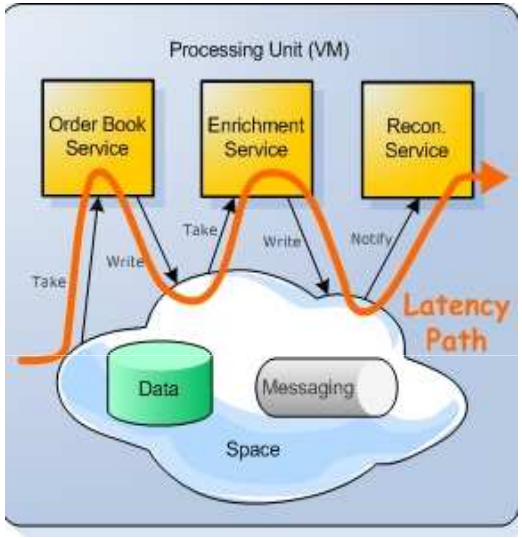
- Performance analysis:
  - Twitter allows 150 req/hour per user
  - For 1M users that means 40,000 req/sec
- Capacity
  - The actual data that needs to be accessed in real-time is only the window of time between interactions (Last 10/30 min should be more then sufficient)
  - The rest of the data can be stored in file-system storage (For users who just logged in)
  - Assuming a ratio of 20% writes and the size per post is 128 bytes:
    - Data produced = 40k*20% * 128 = 1Mb/Sec
    - We can keep a buffer of an hour in less then 5G
- Solution:
  - Use In Memory Data Grid for the real-time access and files-ystem storage for long term, search and other analytics requirements

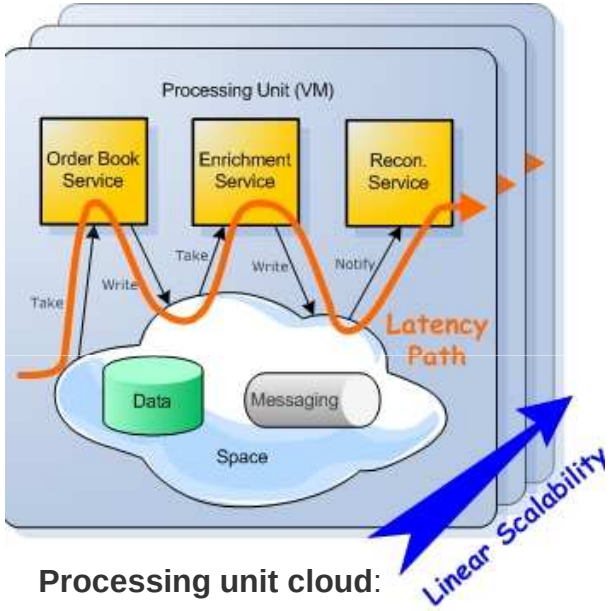
Scaling twitter using Space Based Architecture (SBA)

What is Space Based Architecture (SBA)

Space-Based Architecture (**SBA**) is a software architecture pattern for achieving linear scalability of stateful, high-performance applications, based on the Yale's Tuple-Space Model (Source Wikipedia)







What is a space:

- Elegant – 4 API
- Solves:
  - Data sharing
  - Messaging
  - Workflow
  - Parallel proc.

What is Processing unit :

- Bundle of services, data, messaging
- Collocation into single VM
- Unified Messaging & Data
- In-Memory

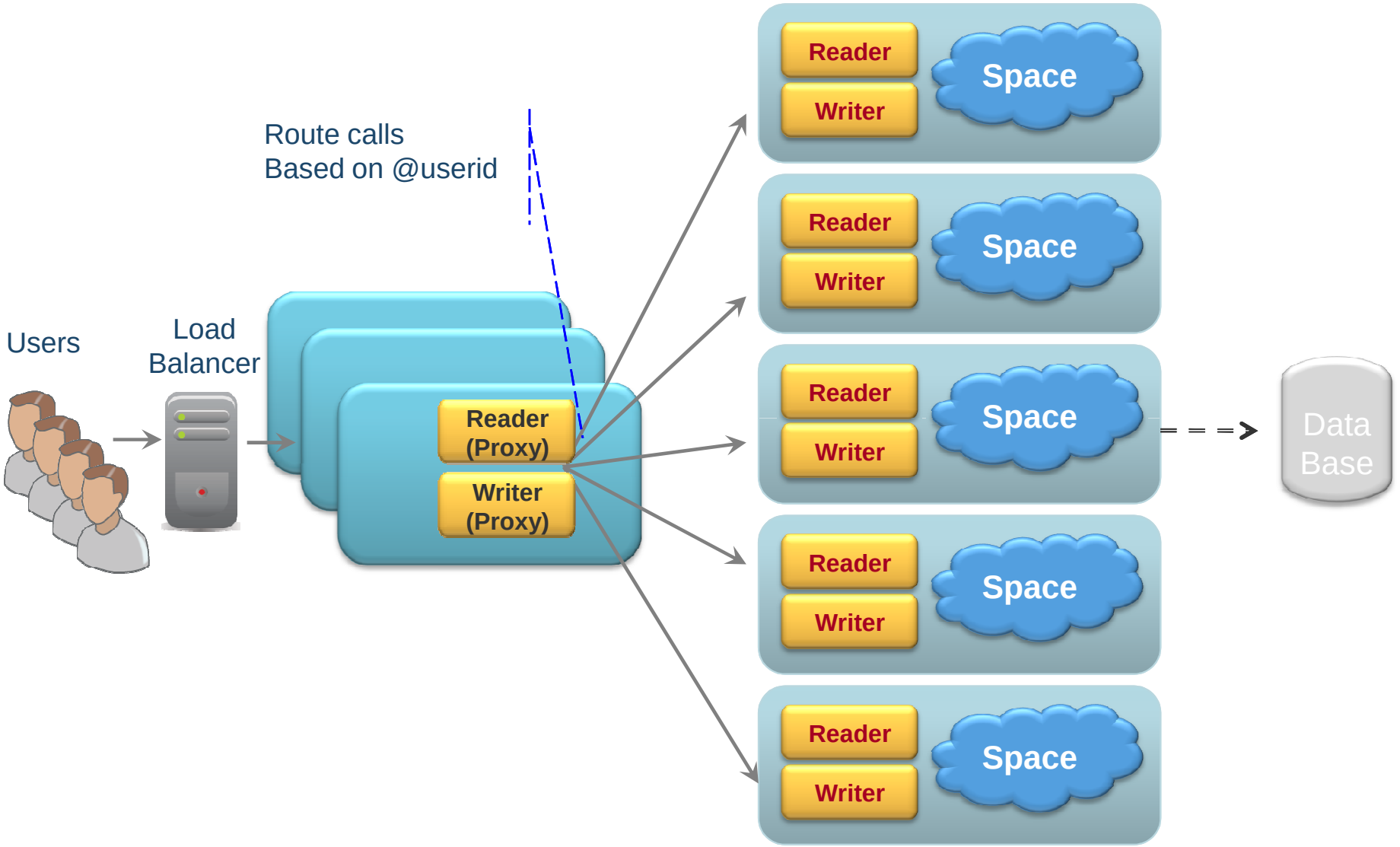
Processing unit cloud:

- Scale through Partitioning
- Virtualized middleware

SBA (Step I) – Remove DB Bottlenecks DataGrid & NoSQL

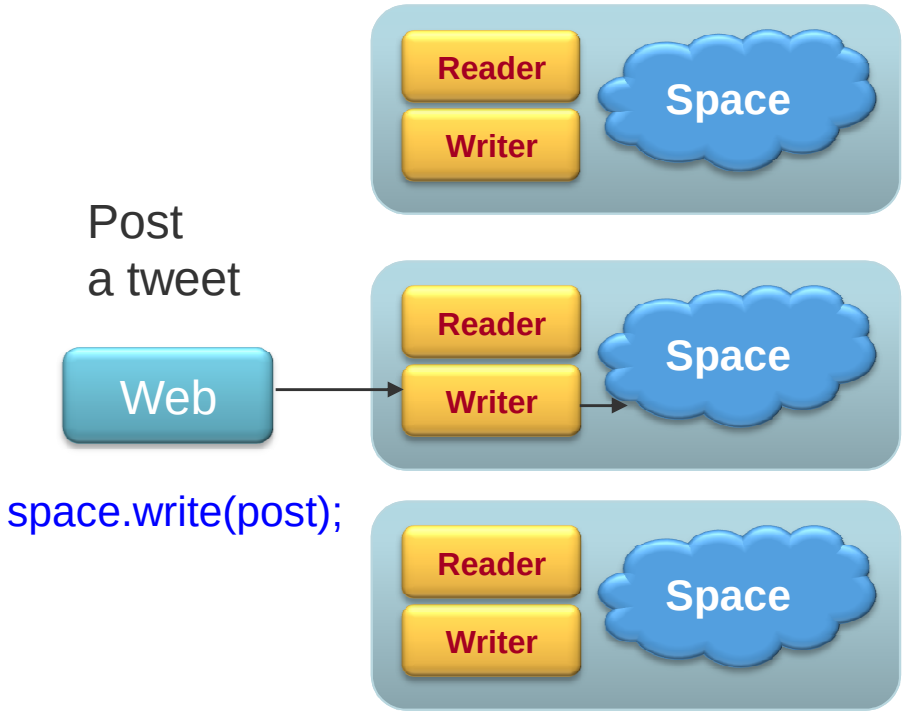
- Reduce I/O bottlenecks – less DB access
- In-Memory caching
- Reduced Network hops (if in-process)

SBA (Step II) – Linear Scalability: Partitioning and Collocation

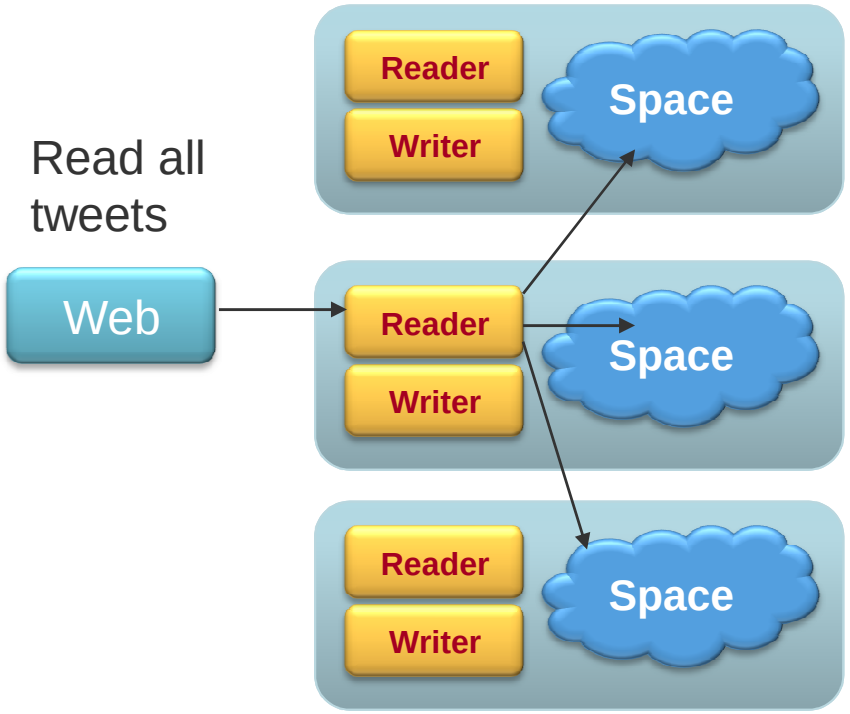


Scaling read/write

- Partition the data based on @user-id
- Read uses Map/Reduce pattern to read the data from all relevant partitions

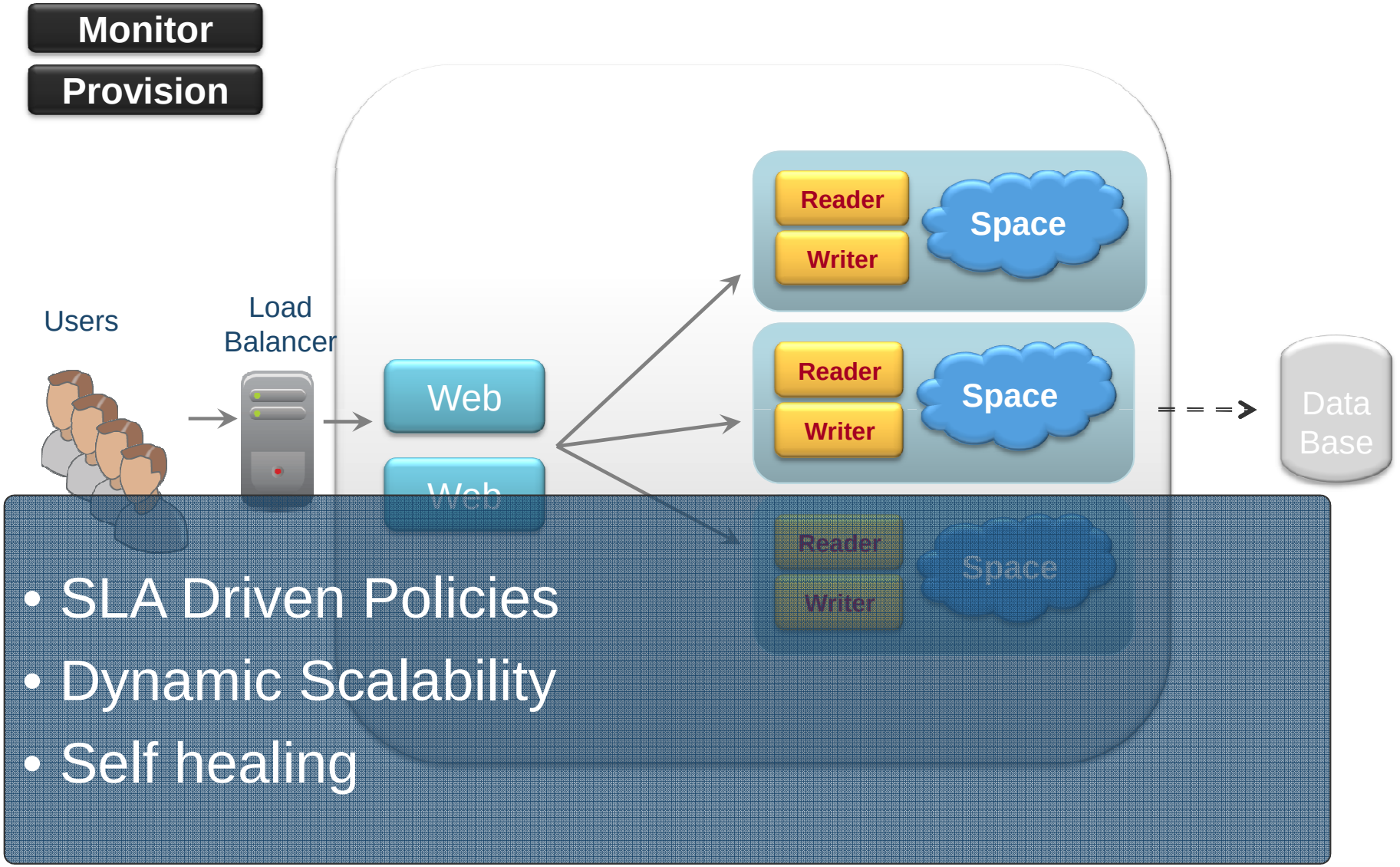


Write



Read- Map/Reduce

Adding Dynamic Scalability



Summary - Space Based Architecture

- Linear Scalability
  - Predictable cost model – pay per value
  - Predictable growth model
- Dynamic
  - On demand – grow only when needed
  - Scale back when resources are not needed anymore
- SLA Driven
  - Automatic
  - Self healing
  - Application aware
- Simple
  - Non intrusive programming model
  - Single clustering Model

Questions?