Chapter

6

Scalability

CHAPTER AUTHORS

Amulya Khare

Yipeng Huang

Hung Doan

Mohit Singh Kanwal

PAST CONTRIBUTORS: Hoang Duc, Juliana Ung Bee Chin, Nguyen Van Quang Huy

1 Table of Contents

1 Table of Contents 2
1 Introduction 3
1.1 Scalability 3
1.2 Vertical Scaling 4
1.3 Horizontal Scaling 5
2 Load Balancing 5
2.1 DNS Load Balancing 6
2.2 Hardware Load Balancing 6
2.3 Software Load Balancing 7
3 Scaling Databases 7
3.1 Need For Parallelism 8
3.2 Key Metrics of measuring parallelism: Scaleup and Speedup 8
3.3 Shared Nothing Architecture 8
3.4 Replication 9
3.4.1 Database replication using the master-slave model 9
3.4.2 Multi-master Replication Model 10
3.4.3 Replication Delay and Consistency 11
3.5 Partitioning 11
3.5.1 Partitioning Strategies 11
3.5.2 Vertical Partitioning 12
3.5.3 Horizontal Partitioning 13
4 Cache as a Scalability Solution 14
4.1 Object caches 15
4.2 Memcached 16
4.3 Reverse Proxy Cache 17
4.4 Content Delivery Network (CDN) 18
5 Performance and Scalability 19
5.1 The macro perspective: Performance monitoring 19
5.2 The micro perspective: Performance measurement 23
6 Conclusion 24

1 Introduction

Within ten years of its introduction, the Internet has transformed from plain hypertext medium into a platform supporting interactive information systems and used by a growing world population. Today, a vast array of platforms, languages and tools are available for building high-end web applications that support large-scale business needs and appeal to a large number of people. While building a web application is a relatively straightforward task for an application developer, building web applications that can scale and scale well is often quite difficult.

As a web application grows and gains popularity, it is used by more and more users. Often the application cannot scale at the same speed that the users pour into the application. The problem is not only a problem because of the increased number of users, but having users that interact more heavily with the site. For example you might have 100,000 active users but never experience any problems. Then you release a feature that allows your users to share pictures easily and suddenly you have an application that is responding very slowly or even crashing. Why did this happen? The feature that handles picture sharing was designed and tested with only a few users and it fell apart under heavy load. So the techniques and technologies that work at the small scale can fail as the application starts to grow in terms of traffic and data volumes.

Once you have such problems, the next step is to find where the bottleneck is and why is it happening. This might appear to be easy but it’s comparable to looking for a needle in a haystack. Sizable web applications are quite complex. Parts coded in different programming languages and by different people. Not only that, but sometimes problems arise in different parts of the application such as the database or the server. Even if the bottleneck is found, the solution might range from adding a code patch to re-developing a particular section of the application to fix the issue. To avoid wasting a lot of time and effort in the future, thinking about scale up-front can help one build applications that work well on a small scale and can be extended easily to handle increasing load without requiring major architectural redesigns.

In this book chapter we will look at making scalable web applications by employing tried and tested practices at the various layers and also gain insights into how the various parameters of performance stack up with and against each other.

1.1 Scalability

Scalability is sometimes defined as "the ease with which a system or component can be modified to fit the problem area." A scalable system has three simple characteristics:

- The system can accommodate increased usage,
- The system can accommodate an increased data set.
- The system is maintainable and works with reasonable performance.

Scalability is not just about speed. Performance and scalability for a system differ from and correlate to each other. Performance measures how fast and efficiently a system can complete certain computing tasks, while scalability measures the trend of performance

with increasing load. If the performance of a software system deteriorates rapidly with increasing load (number of users or volume of transactions) prior to reaching the intended load level, then it is not scalable and will eventually under perform. In other words, we hope that the performance of a software system would sustain as a flat curve with increasing load prior to reaching the intended load level, which is the ideal scalability one can expect.

In other words, scalability refers to the ability of a system to give reasonable performance under growing demands (rising traffic or increased data volume). Not only is a scalable system assured to perform well under increasing load, it would also reduce the need of having to redesign the system under such challenges, and this translates to business gains such as the mitigation of possible financial loss or decreased customer confidence. Scalability is one of the most valuable quality attributes of a system.

The next question that arises now is “How do we scale?” In its simplest form, adding more resources in order to handle the increased load can scale an application. In the next section, we discuss two broad methods: vertical scaling and horizontal scaling, of adding more resources for a particular application. The understanding of these two methods is essential before we delve deep into the various technical aspects of scaling specific components.

1.2 Vertical Scaling

Scaling up or vertical scaling refers to resource maximization of a single unit to expand its ability to handle increasing load. In hardware terms, this includes adding processing power and memory to the physical machine running the server. In software terms, scaling up may include optimizing algorithms and application code. Optimization of hardware resources, such as parallelizing or having optimized number of running processes is also considered techniques of scaling up.

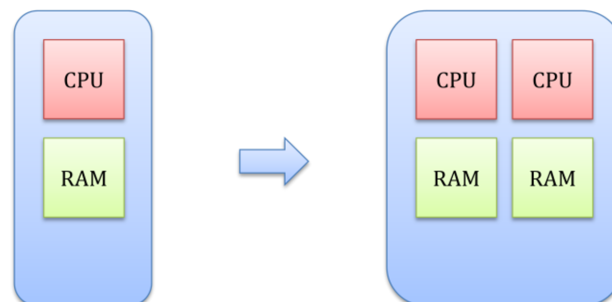


Figure 1 Example of vertical scaling by upgrading the physical machine.

Although scaling up may be relatively straightforward, this method suffers from several disadvantages. Firstly the addition of hardware resources results in diminishing returns instead of super-linear scale. The cost for expansion also increases exponentially. The curve of cost to computational processing is a power law in which the cost begins to increase disproportionally to the increase in processing power provided by larger servers.

In addition, there is the inevitable downtime requirement for scaling up. If all of the web application services and data reside on a single unit, vertical scale on this unit does not guarantee the application’s availability.

1.3 Horizontal Scaling

Scaling out or horizontal scaling refers to resource increment by the addition of units to the system. This means adding more units of smaller capacity instead of adding a single unit of larger capacity. The requests for resources are then spread across multiple units thus reducing the excess load on a single machine.

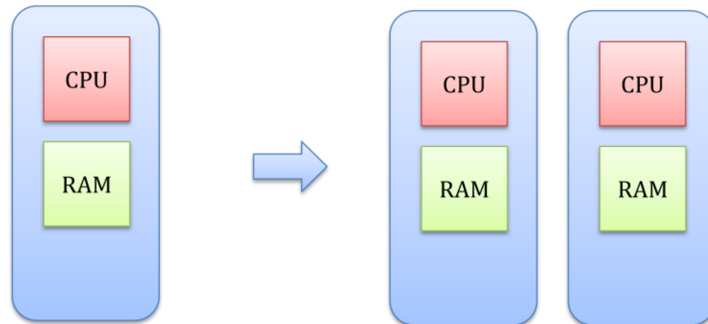


Figure 2 Example of horizontal scaling by adding more machines.

Having multiple units allows us the possibility of keeping the system up even if some units go down, thus, avoiding the "single point of failure" problem and increasing the availability of the system. Also generally, the total cost incurred by multiple smaller machines is less than the cost of a single larger unit. Thus horizontal scaling can be more cost effective compared to vertical scaling.

However, there are disadvantages advantages of horizontal scaling as well. Increasing the number of units means that more resources need to be invested in their maintenance. Also the code of the application itself needs to be modified to allow parallelism and distribution of work among various units. In some cases this task is not trivial and scaling horizontally may be a tough task.

In the next section, we look at the technique of load balancing that is often used to support horizontally scaled web application architecture.

2 LOAD BALANCING

When we start to scale horizontally, a new problem appears. We have multiple processors residing on different physical machines, but we have no management system to spread requests among them. We have multiple requests coming in to the same IP, which we want to service with multiple machines. The problem is to decide which application machine or processing unit would respond to which request. The solution can come from a number of methods, which could be grouped under the technique of load balancing.

There is a limit to what vertical scale can achieve for a particular application. This limit can be determined by the company's budget to buy upgraded hardware or the technical limitation when the company is already using the best server available in the market. These systems need to scale out and be load balanced. Thus, load balancing has now become a must in almost any web service architecture, playing an important part in ensuring availability and scalability of a system.

A load balancer accepts requests from users and then directs them to the right web server. The "right" server here is decided by certain criteria, mainly regarded as the load balancing strategy. There are many strategies, common ones are:

- Round Robin: each server takes turn to receive requests. This is the simplest strategy, similar in spirit to First In First Out applied in caching.
- Least number of connections: the server with the lowest number of connections will be directed the request. This is an attempt to prevent high load.
- Fastest response time: the server that has the fastest response time (either recently or frequently) will be directed the request. This is an attempt to handle requests as fast as possible.
- Weighted: this strategy can be highly customized since the weightage can be configured. This strategy takes into account the scenario where servers in the cluster may not have the same capacity (processing power, storage, etc.). Thus, the more powerful servers will receive more requests than the weaker ones under weighted strategy.

Load balancing can be done in many ways. A simple load balancing can be done at the Domain Name System (DNS) level by managing the DNS responses to address requests from client. Other ways may include installing some hardware or software tools at the application's backend. In the next section, we look at the different ways in which we can achieve load balancing.

2.1 DNS Load Balancing

The easiest way to load balance between web servers is to create more than one record in the DNS zone for your application's domain. When a user then enters your address into their browser, your browser asks the DNS server to return a list of records for that domain. DNS servers shuffle these records and send them back in a random order to each requesting host. The client, on the other hand, always starts by trying the first in the list thus getting directed to a web server on a random basis. DNS-based load balancing is by far the easiest way to balance requests between multiple geographical locations, as no additional configuration is required at the application's end.

This approach has certain disadvantages. Adding or removing any machines from the pool is a slow process. Depending on the cache time of DNS servers, it could take up to a couple of days to make a change to the DNS zone that appears for all users. During that time, some users will see the old zone while some will see the new one. Also because of DNS caching, a user will get stuck on a single machine for an hour or more. If many users share a DNS cache, as in the case with large ISPs, a large portion of your users will get stuck to a single server thus; DNS load balancing is not a very practical solution.

2.2 Hardware Load Balancing

The most straightforward way to balance requests between multiple machines in a pool is to use a hardware appliance. You plug it in, turn it on, set some settings, and start serving traffic. The basic principle is that network traffic is sent to a shared IP in many cases called a virtual IP (VIP), or listening IP. This VIP is an address that it attached to the load balancer. Once the load balancer receives a request on this VIP it will need to make a decision on where to send it, based on its load balancing strategy.

There are several core advantages we gain by using a hardware appliance. Adding and removing real servers from the VIP happens instantly. As soon as we add a new server, traffic starts flowing to it and, when we remove it, traffic immediately stops. Also, we can balance load however we see fit. If we have one web server that has extra capacity for some reason (maybe more RAM or a bigger processor), then we can give it an unfair share of the traffic; making use of all the resources we have instead of allowing extra capacity to go unused.

However, there are a couple of downsides to using a hardware appliance. First time configuration of the load-balancing device may not be an easy job. However, depending on your scale, this might be not be much of an issue, especially if the device is to be set up once and be left there for use for a long time. The main disadvantage with hardware load balancers is that hardware load-balancing devices tend to be very expensive, starting in the tens of thousands of dollars up to the hundreds of thousands. Remembering that we need at least two for disaster tolerance, this can get pretty expensive.

2.3 Software Load Balancing

Software load balancing provides a cheap alternative to hardware load balancers. A number of load balancing software are available in the market, such as Perlbal (http://www.danga.com/perlbal/), Pound (http://www.apsis.ch/pound/) which are free and open-source. They run simple to super-complex software operating systems, on much cheaper hardware (sometimes even ordinary server machines) and provide a much more convenient way of balancing load.

Software load balancers are generally classified into two categories: layer 4 and layer 7, based on the network layer information they use for load balancing. Layer 4 load balancers make use of the information provided by TCP (transmission control protocol) at the network layer. The load balancer captures the request at this layer and utilizes the information contained in the TCP stream: the source and destination IP address and port, which is sufficient to route the request. Given this information, we can direct the connection to the correct port at the backend. Since connection must be established between client and server in connection-oriented transport before sending the request content, the load balancer usually selects a server without looking at the content of the request.

Layer 7 load balancers, on the other hand, inspect the message right up to the application layer, examining the HTTP request itself. They are able to look at the request and its headers and use those as part of the balancing strategy. The requests thus can be balanced based on information in the query string, in cookies or any header we choose, as well as the regular layer 4 information, including source and destination addresses. For example, an often used element for layer 7 balancing is the HTTP request URL itself. By balancing based on the URL, one can ensure that all requests for a specific resource go to a specific server. Layer 7 load-balancers, can provide quality of service requirements for different types of contents and improve overall cluster performance.

3 SCALING DATABASES

A database is the organization of collection of data entities and their relationships. For example, a university database might contain *entities* such as students, faculty, courses, and classrooms, and *relationships* such as students' enrollment in courses, faculty

teaching courses, etc. Formally, software designed to store data are called database management systems (DBMS) but in this section DBMS and database will be used interchangeably.

The database is the most common bottleneck in web applications, since a lot of reads and writes occur at the database level, and hence the primary focus in this book section is on scaling them. A scalable database is one that performs well under increasing traffic and dataset (Henderson, 2006).

In this book section we will attempt to provide a useful understanding of horizontal scaling methods with respect to the database layer.

3.1 Need For Parallelism

Historically, Database Systems were centralized sequential systems where data was located in a single site. However, many data operations can work in parallel. For example, reads can be parallelized easily because no changes are made to the database. Parallelism is the primary method of achieving scalability.

3.2 Key Metrics of measuring parallelism: Scaleup and Speedup

The ideal parallel system demonstrates two key properties: (1) linear speedup: Twice as much hardware performing the task in half the elapsed time, and (2) linear scaleup: Twice as much hardware performing twice as large a task in the same elapsed time.

Speed-up and **scale-up** are illustrated in the Figure below. The speed-up curves show how, for a fixed database size, more transactions can be executed per second by adding CPUs. The scale-up curves show how adding more resources (in the form of CPUs) enables us to process larger problems. The first scale-up graph measures the number of transactions executed per second as the database size is increased and the number of CPUs is correspondingly increased. An alternative way to measure scale-up is to consider the time taken per transaction as more CPUs are added to process an increasing number of transactions per second; the goal here is to sustain the response time per transaction.

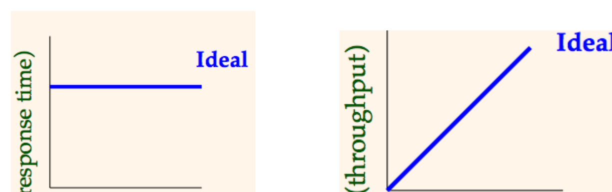


Figure 3 Speedup and Scaleup Curves

3.3 Shared Nothing Architecture

Core to the horizontal scaling of the database is the architecture principle employed called the "**Shared-Nothing Architecture**". This is done in order to take advantage of the parallelism in the database operations.

Using shared nothing architecture for the database layer is the primary method exploited to provide horizontal scaling.

It's becoming increasing popular to use a distributed architecture for web applications in recent times. This architecture is based on a shared-nothing hardware design in

which processors communicate with one another only by sending messages via an interconnection network. In such systems, tuples of each relation in the database are separately stored across disk storage units attached directly to each processor. Partitioning allows multiple processors to scan large relations in parallel without needing any exotic I/O devices (J. & Jim, 1992). Figure above displays the shared nothing architecture, alongside shared-disk and shared-memory architectures.

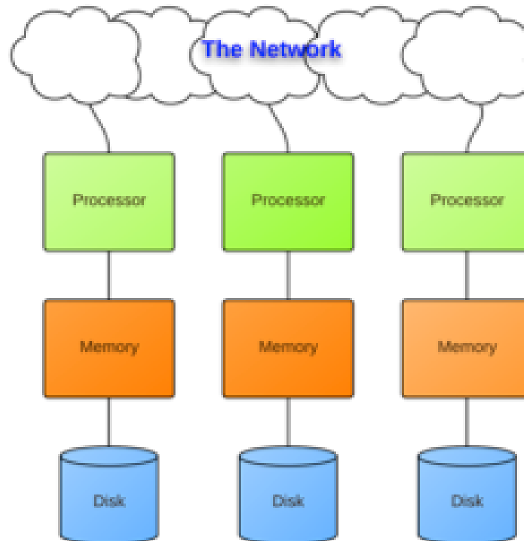


Figure 4 A shared nothing architecture diagram, note how the individual processor and memories do not share resources

The shared-nothing facilitates scalability by providing linear speed-up and linear-scaleup. Consequently, ever-more-powerful parallel database systems can be built by taking advantage of rapidly improving performance for single-CPU systems and connecting as many CPUs as desired. (Ramakrishnan & Gehrke, 2003)

Shared-nothing architectures minimize interference by minimizing resource sharing. As such horizontal scaling of the database via replication or partitioning schemes becomes very easy, as we shall see in the subsequent sections.

3.4 Replication

In database replication, copies (replicas) of data are stored on multiple machines. Clients may read from any of these replicas. Whenever a write occurs, the updated data is replicated to all machines to ensure **consistency**. The database server that does this replication is called the master, while those at the receiving end of the replication are called slaves.

3.4.1 Database replication using the master-slave model

From a single server, the simplest replication model to scale out to is the dedicate master and single slave servers (Figure 2). Client web server(s) may now read from any of these two machines. However, all writes must only be directed to the master, who will subsequently replicate data updates to the slave.

This simple two-machine configuration may be extended to more slaves.

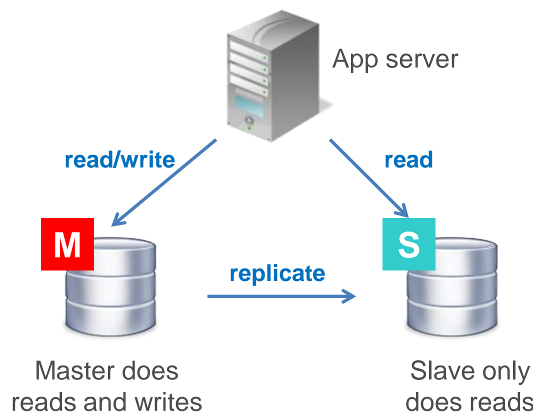


Figure 5 Master Slave Replication Model

Database replication enables read demands to be distributed among multiple machines. However, having one master introduces a single-point-of-failure in the setup. If the master is unavailable, the application will lose the ability to update the data.

3.4.2 Multi-master Replication Model

Instead of having a one-way replication between a master and slave, the multi-master model enables each master to replicate data to the other.

Since there is now more than one machine responsible for updates, the multi-master model eliminates the single-point-of-failure that the master-slave model has. However, this model introduces the danger of data collision. This may happen if we use the primary ID auto-incrementing feature, and were to insert two different records into each of the masters simultaneously.

There are several ways to handle data collision. We may author the application code such that writes to certain tables are only issued to one master and writes to other tables are issued to the other master. We may also avoid the auto-incrementing feature and use alternatives such as other unique keys or other services to generate unique IDs. These are not necessarily convenient workarounds, depending on our needs.

3.4.3 Replication Delay and Consistency

Replication delay is the time it takes for data to be replicated to a machine. Replication delays across all machines will vary. A faster slave machine, or one serving fewer threads for example, would be able to update its data faster than a slower machine. Hence, there will be a time during replication when data is not the same (in other words, not consistent) across all the database replicas. Data inconsistency may also be attributed to a replication model: in the master-slave model, the master machine always has the most updated copy of the data at any time. This is however untrue for any machine in the multi-master model (assuming there is traffic), since different writes would happen at the same time to multiple masters. If we need consistent reading, then we could enforce a synchronous replication (where writes are replicated to all machines) before performing a read.

3.5 Partitioning

Partitioning a relation involves distributing its tuples over several disks. Data partitioning has its origins in centralized systems that had to partition files, either because the file was too big for one disk, or because the file access rate could not be supported by a single disk.

Distributed databases use data partitioning when they place relation fragments at different network sites. Data partitioning allows parallel database systems to exploit the I/O bandwidth of multiple disks by reading and writing them in parallel. This approach allows enabling distributed access to the database thereby aiding in scaling out the database.

3.5.1 Partitioning Strategies

The simplest partitioning strategy distributes tuples among the fragments in a ***round-robin*** fashion. This is the partitioned version of the classic entry-sequence file. Round robin partitioning is excellent if all applications want to access the relation by sequentially scanning all of it on each query. The problem with round robin partitioning is that applications frequently want to associatively access tuples, meaning that the application wants to find all the tuples having a particular attribute value. For example, the SQL query looking for the Smith's in the phone book is an example of an associative search.

Hash partitioning is ideally suited for applications that want only sequential and associative access to the data. Tuples are placed by applying a *hashing* algorithm to an attribute of each tuple. The function specifies the placement of the tuple on a particular disk. Associative access to the tuples with a specific attribute value can be directed to a single disk, avoiding the overhead of starting queries on multiple disks.

Range partitioning maps contiguous attribute ranges of a relation to various disks. Round-robin partitioning maps the i'th tuple to disk i **mod** n. Hashed partitioning, maps each tuple to a disk location based on a hash function. Each of these schemes spreads data among a collection of disks, allowing parallel disk access and parallel processing.

Database systems pay considerable attention to clustering related data together in physical storage. If a set of tuples is routinely accessed together, the database system attempts to store them on the same physical page.

Hashing tends to randomize data rather than cluster it. *Range partitioning* clusters tuples with similar attributes together in the same partition. It is good for sequential and associative access, and is also good for clustering data.

The problem with range partitioning is that it risks *data skew,* where all the data is place in one partition, and *execution skew* in which all the execution occurs in one partition. Hashing and round-robin are less susceptible to these skew problems. Range partitioning can minimize skew by picking non-uniformly-distributed partitioning criteria.

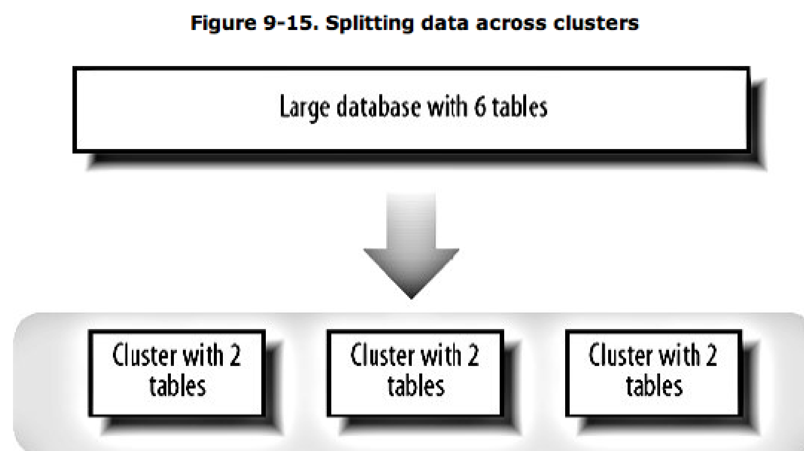
While partitioning is a simple concept that is easy to implement, it raises several new physical database design issues. Each relation must now have a partitioning strategy and a set of disk fragments. Increasing the degree of partitioning usually reduces the response time for an individual query and increases the overall throughput of the system. For sequential scans, the response time decreases because more processors and disks are used to execute the query. For associative scans, the response time improves because fewer tuples are stored at each node and hence the size of the index that must be searched decreases.

There is a point beyond which further partitioning actually increases the response time of a query.

3.5.2 Vertical Partitioning

Vertical partitioning, also known as clustering, is called so because of its limited scope for growth. As with horizontal scaling, clustering involves splitting your database into multiple chunks or clusters, each of which contains a subset of all your tables.

Figure 9-15. Splitting data across clusters



By identifying which queries operate on which tables, we can modify our database dispatching layer to pick the right cluster of machines depending on the tables being queried. Each cluster can then be structured as per your wish: single machine, a master with slaves, or a master-master pair. Different clusters can use different models to suit the needs of the various tables they support.

However, there is a limit to vertical partitioning. This limitation means that if we have a single table or set of joined tables with many writes, we're always bound by the power of a single server, in other words, partitioning will have no effect.

There are other downsides to this approach. Management of the clusters is more difficult than a single database, as different machines now carry different datasets and have different requirements. As we add any components that are not identical, the management costs increases.

3.5.3 Horizontal Partitioning

Horizontal partitioning, also known as **sharding** is a well-known method for horizontal scaling. At its heart is the concept of splitting data between multiple servers so that each server manages only a portion of the overall data. The individual self-contained data partitions are called **shards**.

The benefit of using horizontal partitioning is that the database servers enforce "share nothing" architecture and hence bring with it the capacity to scale without any limits.

MySQL 5's NDB storage engine tries to do something like this internally without you having to change any of your application logic.

Performing horizontal partitioning manually is difficult. Selecting a range of data from a table that has been split across multiple servers becomes multiple fetches with a merge and sort operation. Joins between horizontally partitioned tables become impossibly complicated. This is certainly true for the general case, but if we design our application carefully and avoid the need for cross-shard selects and joins, we can avoid these pitfalls.

The key to avoiding cross-shard queries is to partition your data in such as way that all the records you need to fetch together reside on the same shard. For example, data may be sharded according to customer locations; the resulting shards may now be physically placed to decrease network delays.

When a request reaches the application, a form of look up mechanism is needed to determine the shard from which data should be retrieved.

ID	FirstName	LastName	Faculty
1	John	Carter	FASS
2	Allen	Tan	SoC
3	Bob	Markzinski	SoC
4	Susan	Roberts	Biz
5	Julia	Lyline	FASS
6	Mark	Yankze	SoC
7	Twain	Shernie	Biz
8	Shania	Klow	Biz
9	Susan	Boyle	SoC
10	Bob	Goranski	FASS

ID	FirstName	LastName	Faculty
1	John	Carter	FASS
5	Julia	Lyline	FASS
10	Bob	Goranski	FASS

ID	FirstName	LastName	Faculty
2	Allen	Tan	SoC
3	Bob	Markzinski	SoC
6	Mark	Yankze	SoC
9	Susan	Boyle	SoC

ID	FirstName	LastName	Faculty
4	Susan	Roberts	Biz
7	Twain	Shernie	Biz
8	Shania	Klow	Biz

Figure 6 Splitting the records horizontally into the three different blocks, which can be placed on different locations depending on the faculty.

As seen from the sections above both partitioning and replication provide schemes via which we can horizontally scale our application without having the need to install high performance and expensive hardware. However, with the addition of partitioning logic

and replication software, data sometimes can become inconsistent and as such requires additional logic to handle it. For a detailed treatment of the methods and problems associated with replication and partitioning refer to *Building Scalable Websites* by Henderson.

4 Cache as a Scalability Solution

In previous sections, we have discussed techniques to improve the scalability and performance of web applications by adding more servers to our web servers. There is one shared theme in these techniques, that is, increasing the capacity of our servers so that they can handle more operations. However, there is another way to increase the performance of our server: store the result of common operations temporarily to handle these operations faster. In this chapter we will look into a technique that will help us achieve that mean: Caching.

Caching is a technique that enables applications or devices store data that is likely to be reused in memory, so that the data may be served faster in subsequent requests. The data can be data read from database, web pages, etc. By adding caches to your servers, you can avoid reading or creating the same data record or web pages, thus reduce both the response time and the load on your server. Consequently, your application becomes more scalable.

Caching can be applied at many layers of a server, such as the database layer, web server layer, and network layer. In this chapter we will discuss three types of caches that are regularly applied to scaling web servers, namely object caches, reverse proxy caches and content delivery networks (CDNs).

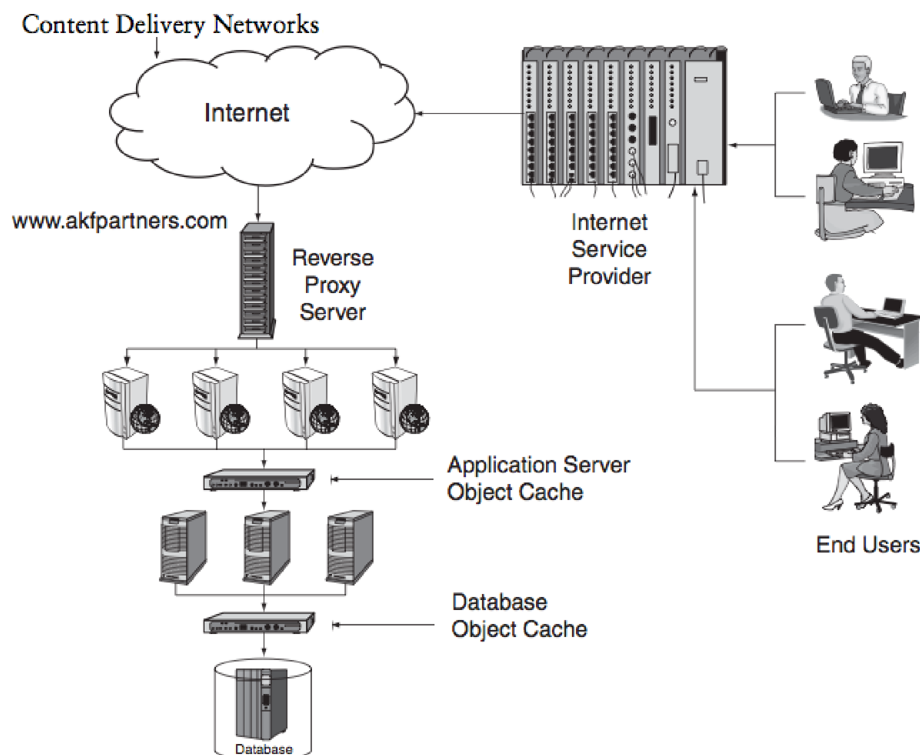


Figure 7: The different layers of web structure at which caching can be applied

(credit: The Art of Scalability, Martin L. Abbot & Michael Fisher 2009)

4.1 Object caches

Object caches are used to store objects for the application to reuse. These objects are either come from a database directly or are generated through data computation.

Object caches are typically placed in front of the database tier because it involves the most heavy computation (Fig 7). Consequently, it is almost always the slowest performing tier and also the most expensive tier to scale when because data has to be written and read from disk. This is especially costly when it is necessary to maintain consistency.

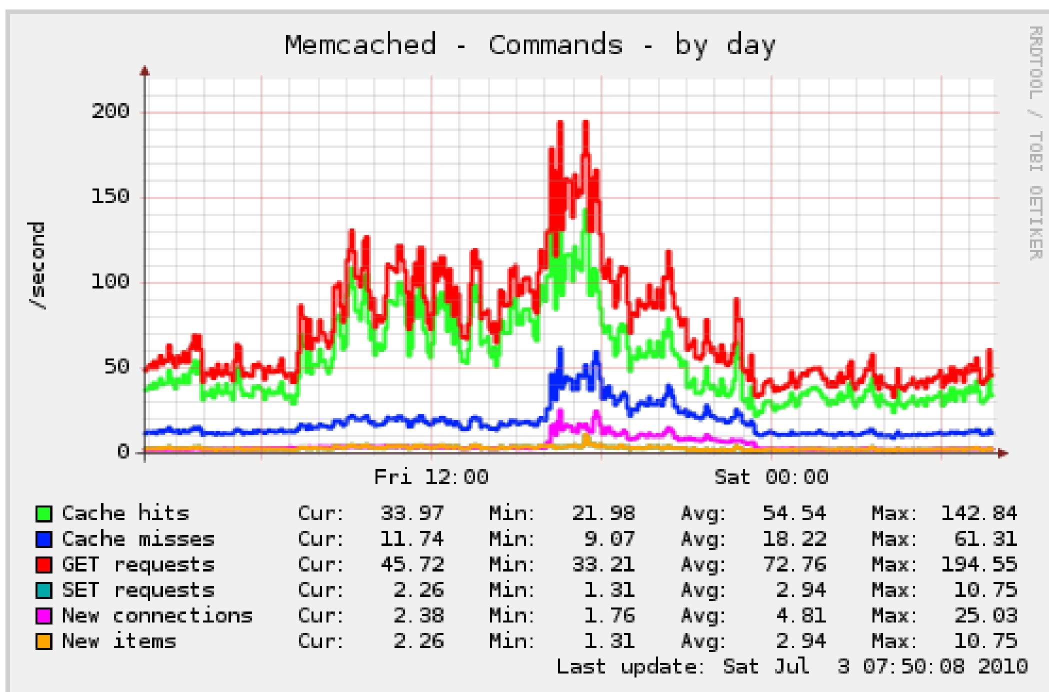


Figure 8: Object cache may handle most of your requests.

(credit: http://2bits.com/)

You may put another layer of object cache "between" the web servers and application servers. This makes sense if the application server is performing a great deal of calculations or manipulations that are cacheable. Just like the database cache, the application server cache saves the server from having to recalculate the same data again and again.

However, objects should not be cached indiscriminately. E.g. it make sense to cache user permission objects but not for transaction objects, because user permissions are rarely changed and are accessed frequently; whereas a transaction object is likely to be different since they are seldom accessed after the transaction is completed.

There are a number of different software solutions to implement object caching, including Memcached, Redis, and Membase. We will look deeper into Memcached, which is currently the most popular choice in industry.

> **INFO** Memcached is widely adopted on popular websites, such as Facebook, Twitter, Wikipedia, Flickr and so on. Facebook has 800 servers to run Memcached.

4.2 Memcached

Memcached is a "high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load." (From http://memcached.org) Memcached caches data, objects and database query results in RAM (hence the name "memory caching") for fast retrieval of data by utilizing key value stores. It is a distributed memory cache solution and can leverages memory of multiple servers by virtualizing them together. Each individual server node is responsible for its own memory allocated for caching and does not need to be aware of memory allocations on other servers. The servers' memory spaces form a single logical memory, accessible by application servers. Using this attribute, memcached can easily be scaled horizontally by adding more servers to the server pool.

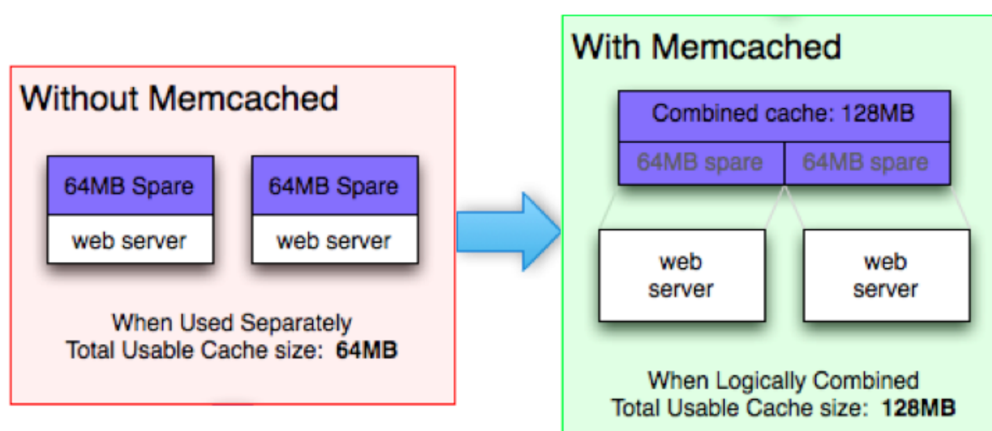


Figure 9: Memcached works by virtualizing the memory of different servers into one big logical memory space. (Credit: http://memcached.org/)

This distributed solution counteracts problems such as memory space constraints. Without distributed caching, each server will operate separate cache spaces which may result in redundancy if they cache the same data. A typical size for the memory is 4GB for 32-bit systems. However, by combining memory space of multiple servers, Memcached can create a memory cache space of the 100GB or more.

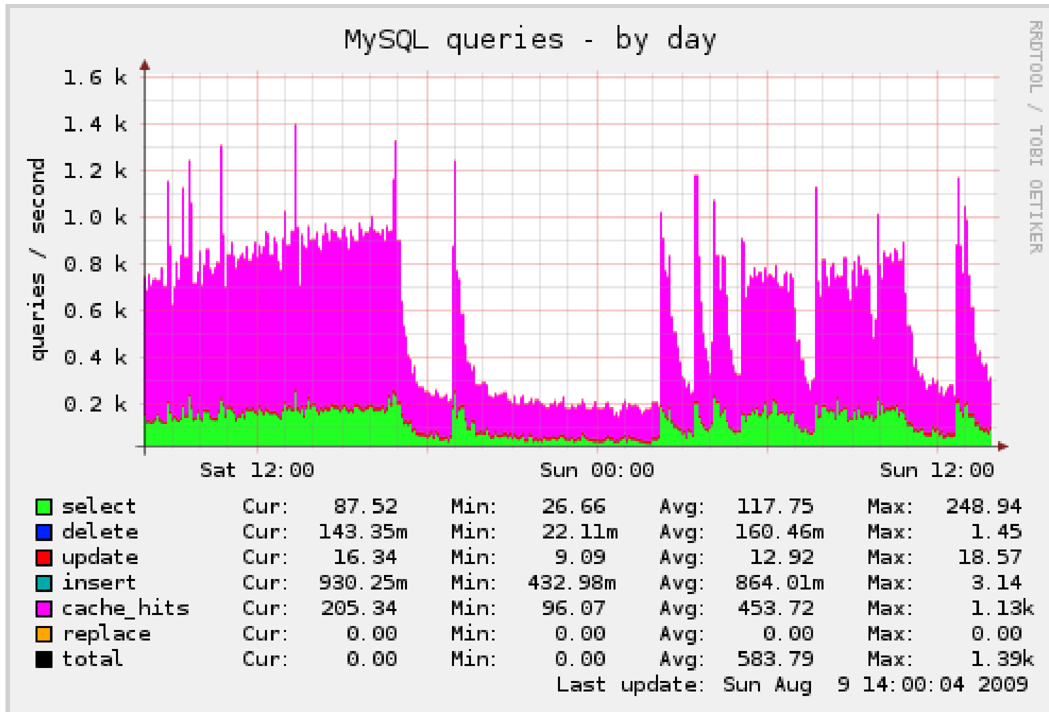


Figure 10: Using Object Caches (Memcached) to significantly reduce database load. (Credit: http://2bits.com/)

4.3 Reverse Proxy Cache

Reverse proxy caches is a strategy for web application caching. While object cache are usually used to cache database objects, reverse proxy cache are used to cache the result of web server e.g. web, DNS and other network lookups. Reverse proxy caches reduce loads on web servers and improve response time to user requests, facilitating scalability.

In implementation, reverse proxy servers are put in front of web servers to redirect cached requests (Fig 5). Requests that are cached are immediately returned to users without processing on the web server. The cached item can be static content such as images but also dynamic pages. The configuration of the specific application will determine what can be cached. Just because your service or application is dynamic does not mean that you cannot take advantage of reverse proxy caching.

The reverse proxy server handles all the requests of a cached page until the pages or data on them is out of date. Additionally when the server receives a request for a page which it does not have, the request is passed to the web server which fulfils the request and refreshes the cache.

There are many different servers that can be used to implement a reverse proxy server, such as Apache HTTP Server with mod_proxy and mod_cache, Lighttpd, and Nginx. Squid and Varnish are open source reverse proxy softwares. Each web server implementation has its own advantages, e.g. Apache is commonly used and easy to setup, but Varnish is explicitly designed for this purpose and will probably perform better than Apache as a proxy server. You should first study your system and requirements to choose software that is suitable for your needs.

4.4 Content Delivery Network (CDN)

A Content Delivery Network is a collection of (third party e.g. Akamai, Cloudfare) servers deployed in different geographical locations containing cached copies of your data or content (such as images, web pages and so on). Using CDN you can improve page load time for user, since the data is retrieved at a location closest to it's users. As an aside, you may also be able to increase the availability of content, since it is stored in multiple locations. You will be able to reduce bandwidth, since the data is served by the CDN provider. Consequently, you can reduce server setup, maintainance costs and manpower.

Periodically the CDN gateway servers makes requests to your application server to validate the content being cached, and update them if necessary. Content being cached is usually static content such as images, Javascript files, css files and so on.

CDN sounds great and simple! However, it usually comes at a premium price. But the decision of whether or not to use CDN should take into account the benefits that CDN provides, especially the reduction in response time is significant for end users, which could lead to increased in user activity (faster response time often elicits more paid transactions). According to the book "Scalability Rules," many websites with greater than 10M of annual revenues are better off using CDNs than serving the traffic themselves.

INFO

CLOUDFLARE (www.cloudflare.com)

CDN services like Amazon S3 or Akamai may require changes in the application code e.g. changing the URL of the image in responding HTML, or writing code to transfer uploaded file to Akamai services. Cloudflare offers an alternative CDN service that is suitable for small-scale web applications. To use it, the user only needs to point his domain to Cloudflare's nameserver.

HOW CLOUDFLARE WORKS

Suppose our web site, mysite.com, is hosted on a webserver at the IP address 1.1.1.1.

Originally, when someone types in "mysite.com" in the web browser, it will make a DNS lookup (to our ISP provider) to discover the IP address of "mysite.com". It then directs subsequent requests to that IP address (in this case, 1.1.1.1).

When we point our domain to Cloudflare's nameserver, Cloudflare's DNS server will handles DNS lookup on behave of our server. Cloudflare returns the IP address of a data center that is geographically nearest to the visitor (say 9.9.9.9). From now on, subsequent requests from visitors will go to 9.9.9.9.

When a request arrives at 9.9.9.9, the Cloudflare's frontline servers check if the resource is in the local cache. Cloudflare caches parts of websites that are static e.g. images, CSS, and Javascript. Utilizing CDN technology, these static resources are delivered from the data center nearer to the visitor.

If the request is for a type of resource that Cloudflare does not cache, or if a current copy is not available, a request is made from the data center (9.9.9.9) back to the original server (1.1.1.1). Because it uses dedicated lines, data passage via Cloudflare is expectantly quicker.

In this section, we discuss how to handle large amounts of traffic by avoid handling them by utilizing caching. In this manner, caching can be one of the best tools that you should leverage to scale your web application. We have also discussed three levels of caching that are most under your control, which are caching at the object, application, and content delivery network levels.

A word of caution, though, is that although Caches may improve the performance and scalability of your system, they will also increase complexity of your system. Multiple levels of caching can make it more difficult to troubleshoot problems in your product. As such, you should monitor and measure your cache system closely (please refer to our part of measuring web application). Besides, while caching is a mechanism that often increases the scalability of your web application, it also needs to be engineered to scale well. Developing a caching solution that doesn't scale well will create a scalability chokepoint within your system and lead to lower availability down the road. You should ensure that you've designed the caches to be highly available and easily maintained.

5 Performance and Scalability

If your scaled up or scaled out solution is performing well, performance evaluation is unnecessary. However when scalability performance grows intolerable or the cost of upgrading becomes exorbitant, performance evaluation becomes increasingly important to squeeze out every ounce of possible performance from your computer systems.

Performance evaluation enables you to identify and deal with bottlenecks as they arise, quantify the impact of a design change or optimization and "capacity plan" for expected levels of workload traffic. In the following section, we will use identifying bottlenecks as a running example.

How do bottlenecks affect scalability? Firstly, bottlenecks often result in a delay or denial of service to users of the system. Secondly, bottlenecks often affect how you should scale. For example if programs on your computer system are inherently I/O intensive, it makes good sense to upgrade the swap out a hard-disk to one with a higher RPM or a SSD which is significantly more efficient at handling I/O. Hence even if your computer systems are scaling well without evaluation, evaluation is important concern for future scalability in light of growing resource consumption.

Our goal is to identify possible bottlenecks in computer systems. But how this be done systematically? A seasoned practitioner understands that he or she must tackle the problem with one eye on the macro and the other on the micro perspective.

5.1 The macro perspective: Performance monitoring

Part of the solution is to observe the system in its "natural" or day-to-day state. This is called **performance monitoring**. Although it may be counter intuitive to rely on monitoring generics (such as CPU utilization, I/O activity, etc) instead of drilling right down to the performance problem, monitoring is often an important part of observing the problem. For example, if you were monitoring a system and observed a slowdown in system response time, you could check if there had been a sudden increase in resource consumption. However if you were not already monitoring your system, you would not be able to tell whether resource consumption was abnormal.

Performance monitoring also has other benefits that help us to take on performance bottlenecks that cannot be achieved through performance measurement or other techniques.

1. Performance monitoring allows us to preempt bottlenecks before they occur. This is possible because systems often have characterizable workloads (e.g. CPU intensive, I/O intensive) when work is repeated (e.g. each customer signing up for a user account, requires a similar SQL INSERT statement). By observing workloads, trends, and resources, it is possible to forecast when a bottleneck could occur.
2. Performance monitoring helps us to respond quickly to bottlenecks after they occur. For example, in the Nagios monitoring system, a bottleneck created by a failed hardware generates an SMS alert so that administrators can tend to the crisis quickly. This could be important to maintain a service level agreement e.g. 99.9% uptime.
3. Performance monitoring provides high level of accountability. The monitoring software stores records of performance-monitored logs which can be used as evidence of the system performance, so that every incident is reviewed and dealt with.

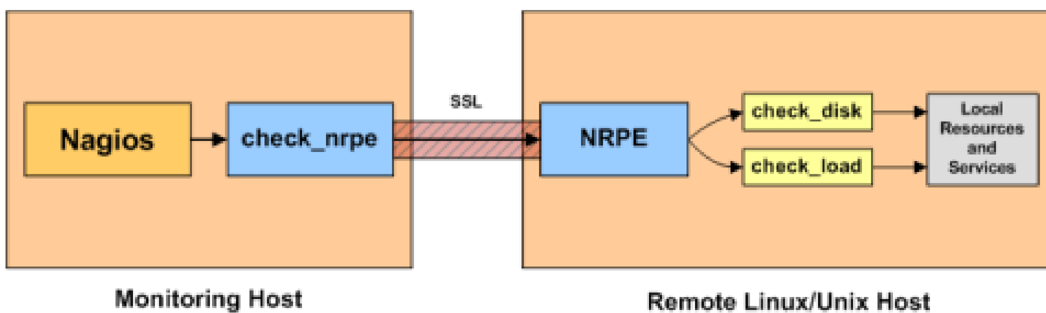
Here are a couple of tools to get you started:

Nmon, short for Nigel's Monitor after its author, is a useful systems administrator, tuner, and benchmark for UNIX-like systems (If you are on Windows, try perfmon) It should really be described as “the 1 tool to rule them all” because it monitors CPU, memory, network, and disk utilization, as well as filesystem and NFS statistics, process and kernel activity. These are the main components that characterize a computer system's workload. Nmon provides an interactive “live” monitoring mode as well as a logging facilitate for longer term data capture and post event analysis.

```
nmon-13g─────────────────────Hostname=yipeng-ThinkPRefresh= 2secs ───08:26.55─
 CPU Utilisation ─────────────────────────────────────────────────────────────
                           +-------------------------------------------------+
CPU  User%  Sys% Wait% Idle|0          |25         |50          |75       100|
 1  33.7   4.5   0.5   61.3|UUUUUUUUUUUUUUUUss                  >            |
 2  33.8   3.0  15.9   47.3|UUUUUUUUUUUUUUUUsWWWWWWW             >           |
                           +-------------------------------------------------+
Avg 33.8   3.5   8.0   54.6|UUUUUUUUUUUUUUUUsWWWW                >           |
                           +-------------------------------------------------+
 Memory Stats ────────────────────────────────────────────────────────────────
                 RAM     High      Low     Swap
 Total MB       3860.2    -0.0     -0.0    4513.0
 Free  MB       1595.8    -0.0     -0.0    4476.9
 Free Percent     41.3%  100.0%   100.0%     99.2%
             MB                  MB                 MB
                     Cached=   960.2     Active=  1066.8
 Buffers=   377.5 Swapcached=    36.1  Inactive =   826.4
 Dirty  =     0.1 Writeback =     0.0  Mapped   =   158.0
 Slab   =   263.6 Commit_AS =  2324.9 PageTables=    23.0
 Disk I/O ──/proc/diskstats────mostly in KB/s─────Warning:contains duplicates─
DiskName Busy  Read WriteKB|0          |25         |50          |75       100|
sda        26%    0.0 2953.0|WWWWWWWWWWWW>                                   |
sda1        0%    0.0    0.0|>                                               |
sda2        0%    0.0    0.0|>                                               |
sda3        0%    0.0    0.0|>                                               |
sda4        0%    0.0    0.0|>                                               |
sda5        0%    0.0    0.0|>                                               |
sda6       25%    0.0 2953.0|WWWWWWWWWWWW>                                   |
sda7        0%    0.0    0.0|>                                               |
Totals Read-MB/s=0.0      Writes-MB/s=5.8      Transfers/sec=77.9
```

Nmon is more relevant for resource monitoring on a single machine. It is possible to run nmon on every machine in your network, captured data would be piecemeal at best. A network monitoring tool is more appropriate to monitor a cluster of computers.

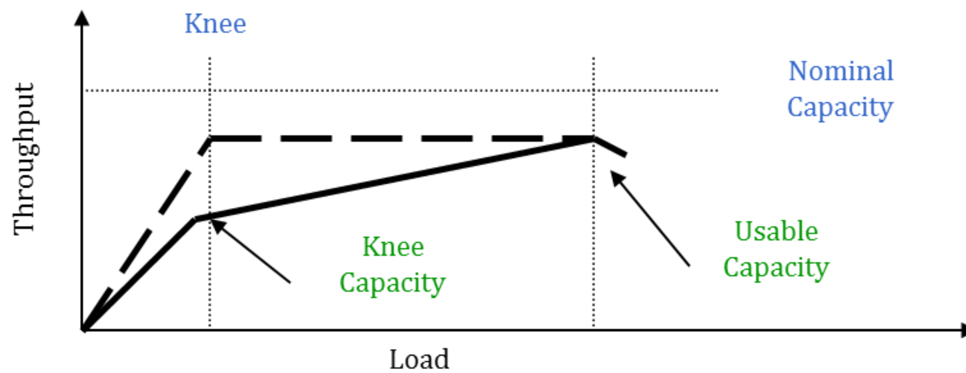
A list of free network monitoring tools can be found on at http://en.wikipedia.org/wiki/Comparison_of_network_monitoring_systems. A quick search on the web suggests that popular choices include Nagios, Cacti, Ganglia, Munin. Nagios is by far the most popular choice. But Windows users should experiment with Cacti or Ganglia. Nagios can monitor Windows machines but requires a Linux monitoring server. It has a large list of plugins that enables it to perform different functions. For example, the Nagios Remote Plugin Executor (NRPE) allows a monitoring server to retrieve resource statistics of remote machines at minimal overhead.



In addition to resource monitoring on each machine, Nagios can monitors network resources (e.g. routers and switches), network services (e.g. HTTP, FTP, SSH, POP3), and remote events (e.g. exceeding temperature thresholds). The strength and weakness of

Nagios is that it doesn't monitor anything unless you tell it to. This makes it more difficult to pick up but thankfully, many problems are fairly obvious e.g. the unavailability of (no response from) HTTP, FTP, SSH, POP3.

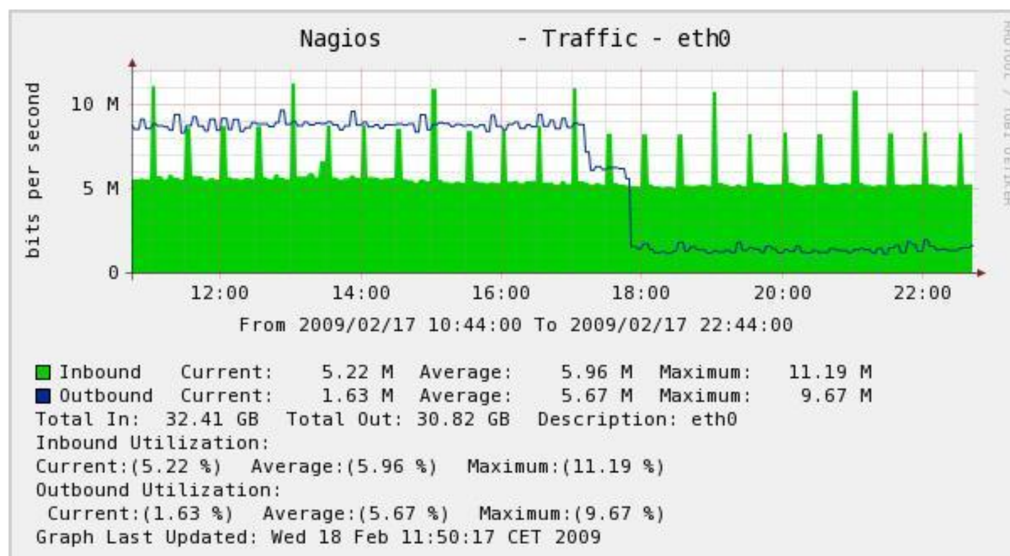
Monitoring network resources is particularly interesting in our context because it is a potential bottleneck that we want to preempt.



Picture credit: http://web.cs.wpi.edu/~claypool/courses/533-S04/slides/

With reference to the above graph, the nominal capacity is the theoretical maximum or bandwidth of the network. The usable capacity is the maximum achievable throughput within some prespecified response time, whist the knee capacity is sweet spot of operations beyond which things "go bad". Increasing the load further results in significant increase in response time. Hence both the usable and knee capacities can be considered bottlenecks.

With a Nagios and a load test, you can establish the knee and usable thresholds for your computer system and assign Nagios to alert you if it exceeds those thresholds. Nagios can monitor the traffic rate as well as packet loss and round time average using SNMP (Simple Network Management Protocol), enabling you to quantify the system performance for your users. See http://nagios.sourceforge.net/docs/3_0/monitoring-routers.html for more details.



(Picture credit: http://www.drachen-server.de/archives/25-IO-tuning-for-Nagios.html)

5.2 The micro perspective: Performance measurement

Now let's consider the micro perspective where we focus on the program in question directly. By profiling a program, we can assess whether and why it is performing poorly and make changes to improve it. This is called **performance measurement and optimization**. Although it may be the most direct way of drilling right down to the performance problem, it may not always reveal a representative picture if the problem is tightly coupled with the environment (e.g. memory contention between programs) or if a representative environment cannot be simulated.

The tools used in performance measurement are aptly called profilers because they produce a profile of the program. Like the profiles produced by a police sketch artist, they describe key features but may not be entirely accurate especially since there is overhead from executing the profiling itself. There are two types of methods of profiling by statistical sampling and code instrumentation. From the user perspective, the difference is that statistical sampling provides less specificity but at lower overhead and vice versa.

The output of the profiler can be just flat (as shown below), depicting the approximate call time of each function or represented in a call-graph form (where chained calls are depicted).

Flat profile example from **gprof**#:

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.34	0.02	0.02	7208	0.00	0.00	open
16.67	0.03	0.01	244	0.04	0.12	offtime
16.67	0.04	0.01	8	1.25	1.25	memccpy
16.67	0.05	0.01	7	1.43	1.43	write
16.67	0.06	0.01	236	0.00	0.00	mcount
0.00	0.06	0.00	192	0.00	0.00	tzset
0.00	0.06	0.00	47	0.00	0.00	tolower
0.00	0.06	0.00	45	0.00	0.00	strlen

Profilers vary in the level of detail that they provide. Here's an example of a line by line annotation. Ir, Dr, Dw refer to instruction cache reads, data cache reads and data cache writes respectively.

```
Ir        I1mr ILmr Dr      D1mr  DLmr  Dw      D1mw   DLmw

      .    .    .       .     .     .       .      .      .  void init_hash_table(char *file_name, Word_Node *table[])
      3    1    1       .     .     .       1      0      0  {
      .    .    .       .     .     .       .      .      .      FILE *file_ptr;
      .    .    .       .     .     .       .      .      .      Word_Info *data;
      1    0    0       .     .     .       1      1      1      int line = 1, i;
      .    .    .       .     .     .       .      .      .
      5    0    0       .     .     .       3      0      0      data = (Word_Info *) create(sizeof(Word_Info));
      .    .    .       .     .     .       .      .      .
  4,991    0    0   1,995     0     0     998      0      0      for (i = 0; i < TABLE_SIZE; i++)
  3,988    1    1   1,994     0     0     997     53     52          table[i] = NULL;
      .    .    .       .     .     .       .      .      .
      .    .    .       .     .     .       .      .      .      /* Open file, check it. */
      6    0    0       1     0     0       4      0      0      file_ptr = fopen(file_name, "r");
      2    0    0       1     0     0       .      .      .      if (!(file_ptr)) {
      .    .    .       .     .     .       .      .      .          fprintf(stderr, "Couldn't open '%s'.\n", file_name);
      1    1    1       .     .     .       .      .      .          exit(EXIT_FAILURE);
      .    .    .       .     .     .       .      .      .      }
      .    .    .       .     .     .       .      .      .
165,062    1    1  73,360     0     0  91,700      0      0      while ((line = get_word(data, line, file_ptr)) != EOF)
146,712    0    0  73,356     0     0  73,356      0      0          insert(data->;word, data->line, table);
      .    .    .       .     .     .       .      .      .
      4    0    0       1     0     0       2      0      0      free(data);
      4    0    0       1     0     0       2      0      0      fclose(file_ptr);
      3    0    0       2     0     0       .      .      .  }
```

(Photo credit: http://valgrind.org/docs/manual/cg-manual.html)

With a line by line profiling we can find costly segments of our code and focus our efforts on tuning those partitions, rather than working blind on optimizations that may not make a significant difference. In the above example, we may expect the majority of the cost to come from the insertion statement. Interestingly the while loop is just as expensive as the insertion. This may lead us to try some form of loop unrolling or blocked data strategy (retrieving more data with each iteration).

Some profilers recommendations include: Valgrind, VTune, CodeAnalyst, AQtime, and SAP Memory Analyzer. Hopefully from this short introduction, you will have understood and been inspired by the power of performance measurement to achieve scalability.

6 CONCLUSION

In this book chapter, we've taken you on a tour of scalability from theory to practice. We've explored what scalability is, how it is implemented, and looked at parallel databases, caching, load balancing, and performance optimization - key techniques that are used to scale web applications. We've also showcased software tools to bring these techniques to life, MySQL Cluster, Memcached, Nagios, Valgrind, and Cloudfare. Scalability isn't easy, but you have learnt enough to get started. So start implementing some of these techniques, keep iterating and learning, and watch your scaling headaches disappear!