Scalable System Design

by Ricky Ho MVB · Apr. 08, 11 · Integration Zone · News

Building scalable system is becoming a hotter and hotter topic. Mainly because more and more people are using computer these days, both the transaction volume and their performance expectation has grown tremendously.

This one covers general considerations. I have another blogs with more specific coverage on DB scalability as well as Web site scalability.

General Principles

"Scalability" is not equivalent to "Raw Performance"

- Scalability is about reducing the adverse impact due to growth on performance, cost, maintainability and many other aspects
- e.g. Running every components in one box will have higher performance when the load is small. But it is not scalable because performance drops drastically when the load is increased beyond the machine's capacity

Understand environmental workload conditions that the system is design for

- Dimension of growth and growth rate: e.g. Number of users, Transaction volume, Data volume
- Measurement and their target: e.g. Response time, Throughput

Understand who is your priority customers

- Rank the importance of traffic so you know what to sacrifice in case you cannot handle all of them

Scale out and Not scale up

- Scale the system horizontally (adding more cheap machine), but not vertically (upgrade to a more powerful machine)

Keep your code modular and simple

- The ability to swap out old code and replace with new code without worries of breaking other parts of the system allows you to experiment different ways of optimization quickly
- N le modularity for any (including performance-related) reasons

Don't eck, Measure it

- ow code which are frequently executed. Don't optimize slow code if they are rarely executed

- Write performance unit test so you can collect fine grain performance data at the component level
- Setup a performance lab so you can conduct end-to-end performance improvement measurement easily

Plan for growth

- Do regular capacity planning. Collect usage statistics, predict the growth rate

Common Techniques

Server Farm (real time access)

- If there is a large number of independent (potentially concurrent) request, then you can use a server farm which is basically a set of identically configured machine, frontend by a load balancer.
- The application itself need to be stateless so the request can be dispatched purely based on load conditions and not other factors.
- Incoming requests will be dispatched by the load balancer to different machines and hence the workload is spread and shared across the servers in the farm.
- The architecture allows horizontal growth so when the workload increases, you can just add more server instances into the farm.
- This strategy is even more effective when combining with Cloud computing as adding more VM instances into the farm is just an API call.

Data Partitioning

- Spread your data into multiple DB so that data access workload can be distributed across multiple servers
- By nature, data is stateful. So there must be a deterministic mechanism to dispatch data request to the server that host the data
- Data partitioning mechanism also need to take into considerations the data access pattern. Data that need to be accessed together should be staying in the same server. A more sophisticated approach can migrate data continuously according to data access pattern shift.
- Most distributed key/value store do this

Map / Reduce (Batch Parallel Processing)

- The algorithm itself need to be parallelizable. This usually mean the steps of execution should be relatively independent of each other.
- Google's Map/Reduce is a good framework for this model. There is also an open source Java framework Hadoop as well.

Content Delivery Network (Static Cache)

- This is common for static media content. The idea is to create many copies of contents that are distributed geographically across servers.
- User request will be routed to the server replica with close proxmity

Cache Engine (Dynamic Cache)

- This is a time vs space tradeoff. Some executions may use the same set of input parameters over and over again. Therefore, instead of redo the same execution for same input parameters, we can remember the previous execution's
- plemented as a lookup cache.
- HCache are some of the popular caching packages.

Resources Pool

- DBSession and TCP connection are expensive to create, so reuse them across multiple requests

Calculate an approximate result

- Instead of calculate an accurate answer, see if you can tradeoff some accuracy for speed.
- If real life, usually some degree of inaccuracy is tolerable

Filtering at the source

- Try to do more processing upstream (where data get generated) than downstream because it reduce the amount of data being propagated

Asynchronous Processing

- You make a call which returns a result. But you don't need to use the result until at a much later stage of your process. Therefore, you don't need to wait immediately after making the call., instead you can proceed to do other things until you reach the point where you need to use the result.
- In additional, the waiting thread is idle but consume system resources. For high transaction volume, the number of idle threads is (arrival_rate * processing_time) which can be a very big number if the arrival_rate is high. The system is running under a very ineffective mode
- The service call in this example is better handled using an asynchronous processing model. This is typically done in 2 ways: Callback and Polling
- In callback mode, the caller need to provide a response handler when making the call. The call itself will return immediately before the actually work is done at the server side. When the work is done later, response will be coming back as a separate thread which will execute the previous registered response handler. Some kind of co-ordination may be required between the calling thread and the callback thread.
- In polling mode, the call itself will return a "future" handle immediately. The caller can go off doing other things and later poll the "future" handle to see if the response if ready. In this model, there is no extra thread being created so no extra thread co-ordination is needed.

Implementation design considerations

- Use efficient algorithms and data structure. Analyze the time (CPU) and space (memory) complexity for logic that are execute frequently (ie: hot spots). For example, carefully decide if hash table or binary tree should be use for lookup.
- Analyze your concurrent access scenarios when multiple threads accessing shared data. Carefully analyze the synchronization scenario and make sure the locking is fine-grain enough. Also watch for any possibility of deadlock situation and how you detect or prevent them. A wrong concurrent access model can have huge impact in your system's scalability. Also consider using Lock-Free data structure (e.g. Java's Concurrent Package have a couple of them)
- Analyze the memory usage patterns in your logic. Determine where new objects are created and where they are eligible for garbage collection. Be aware of the creation of a lot of short-lived temporary objects as they will put a high load on the Garbage Collector.
- However, never trade off code readability for performance. (e.g. Don't try to bundle too much logic into a single method). Let the VM handle this execution for you.

Coupling in Distributed Systems

How Otto Puts Docker's Development and Deployment Capabilities to Shame

Evolving the Gradle API to Reduce Configuration Time

Free DZone Refcard
RESTful API Lifecycle Management

Topics: ARCHITECTS , ARCHITECTURE , CACHING , INFRASTRUCTURE , SYSTEM DESIGN

Published at DZone with permission of Ricky Ho , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.