At eBay, one of the primary architectural forces we contend with every day is scalability. It colors and drives every architectural and design decision we make. With hundreds of millions of users worldwide, over two billion page views a day, and petabytes of data in our systems, this is not a choice - it is a necessity.

In a scalable architecture, resource usage should increase linearly (or better) with load, where load may be measured in user traffic, data volume, etc. Where performance is about the resource usage associated with a single unit of work, scalability is about how resource usage changes as units of work grow in number or size. Said another way, scalability is the shape of the price-performance curve, as opposed to its value at one point in that curve.

There are many facets to scalability - transactional, operational, development effort. In this article, I will outline several of the key best practices we have learned over time to scale the transactional throughput of a web-based system. Most of these best practices will be familiar to you. Some may not. All come from the collective experience of the people who develop and operate the eBay site.

Best Practice #1: Partition by Function

Whether you call it SOA, functional decomposition, or simply good engineering, related pieces of functionality belong together, while unrelated pieces of functionality belong apart. Further, the more decoupled that unrelated functionality can be, the more flexibility you will have to scale them independently of one another.

At the code level, we all do this all the time. JAR files, packages, bundles, etc., are all mechanisms we use to isolate and abstract one set of functionality from another.

At the application tier, eBay segments different functions into separate application pools. Selling functionality is served by one set of application servers, bidding by another, search by yet another. In total, we organize our roughly 16,000 application servers into 220 different pools. This allows us to scale each pool independently of one another, according to the demands and resource consumption of its function. It further allows us to isolate and rationalize resource dependencies - the selling pool only needs to talk to a relatively small subset of backend resources, for example.

At the database tier, we follow much the same approach. There is no single monolithic database at eBay. Instead there is a set of database hosts for user data, a set for item data, a set for purchase data, etc. - 1000 logical databases in all, on 400 physical hosts. Again, this approach allows us to scale the database infrastructure for each type of data independently of the others.

Best Practice #2: Split Horizontally

While functional partitioning gets us part of the way, by itself it is not sufficient for a fully scalable architecture. As decoupled as one function may be from another, the demands of a single functional area can and will outgrow any single system over time. Or, as we like to remind ourselves, "if you can't split it, you can't scale it." Within a particular functional area, then, we need to be able to break the workload down into manageable units, where each individual unit retains good price-performance. Here is where the horizontal split comes in.

At the application tier, where eBay's interactions are by design stateless, splitting horizontally is trivial. Use a standard load-balancer to route incoming traffic. Because all application servers are created equal and none retains any transactional state, any of them will do. If we need more processing power, we simply add more application servers.

The more challenging problem arises at the database tier, since data is stateful by definition. Here we split (or "shard") the data horizontally along its primary access path. User data, for example, is currently divided over 20 hosts, with each host containing 1/20 of the users. As our numbers of users grow, and as the data we store for each user grows, we add more hosts, and subdivide the users further. Again, we use the same approach for items, for purchases, for accounts, etc. Different use cases use different schemes for partitioning the data: some are based on a simple modulo of a key (item ids ending in 1 go to one host, those ending in 2 go to the next, etc.), some on a range of ids (0-1M, 1-2M, etc.), some on a lookup table, some on a combination of these strategies. Regardless of the details of the partitioning scheme, though, the general idea is that an infrastructure which supports partitioning and repartitioning of data will be far more scalable than one which does not.

Best Practice #3: Avoid Distributed Transactions

At this point, you may well be wondering how the practices of partitioning data functionally and horizontally jibe with transactional guarantees. After all, almost any interesting operation updates more than one type of entity - users and items come to mind immediately. The orthodox answer is well-known and well-understood - create a distributed transaction across the various resources, using two-phase commit to guarantee that all updates across all resources either occur or do not. Unfortunately, this pessimistic approach comes with substantial costs. Scaling, performance, and latency are adversely affected by the costs of coordination, which worsens geometrically as you increase the number of dependent resources and incoming clients. Availability is similarly limited by the requirement that all dependent resources are available. The pragmatic answer is to relax your transactional guarantees across unrelated systems.

It turns out that you can't have everything. In particular, guaranteeing immediate consistency across multiple systems or partitions is typically neither required nor possible. The CAP theorem, postulated almost 10 years ago by Inktomi's Eric Brewer, states that of three highly desirable properties of distributed systems - consistency (C), availability (A), and partition-tolerance (P) - you can only choose two at any one time. For a high-traffic web site, we have to choose partition-tolerance, since it is fundamental to scaling. For a 24x7 web site, we typically choose availability. So immediate consistency has to give way.

At eBay, we allow absolutely no client-side or distributed transactions of any kind - no two-phase commit. In certain well-defined situations, we will combine multiple statements on a single database into a single transactional operation. For the most part, however, individual statements are auto-committed. While this intentional relaxation of orthodox ACID properties does not guarantee immediate consistency everywhere, the reality is that most systems are available the vast majority of the time. Of course, we do employ various techniques to help the system reach eventual consistency: careful ordering of database operations, asynchronous recovery events, and reconciliation or settlement batches. We choose the technique according to the consistency demands of the particular use case.

The key takeaway here for architects and system designers is that consistency should not be viewed as an all or nothing proposition. Most real-world use cases simply do not require immediate consistency. Just as availability is not all or nothing, and we regularly trade it off against cost and other forces, similarly our job becomes tailoring the appropriate level of consistency guarantees to the requirements of a particular operation.

Best Practice #4: Decouple Functions Asynchronously

The next key element to scaling is the aggressive use of asynchrony. If component A calls component B synchronously, A and B are tightly coupled, and that coupled system has a single scalability characteristic -- to scale A, you must also scale B. Equally problematic is its effect on availability. Going back to Logic 101, if A implies B, then not-B implies not-A. In other words, if B is down then A is down. By contrast, if A and B integrate asynchronously, whether through a queue, multicast messaging, a batch process, or some other means, each can be scaled independently of the other. Moreover, A and B now have independent availability characteristics - A can continue to move forward even if B is down or distressed.

This principle can and should be applied up and down an infrastructure. Techniques like SEDA (Staged Event-Driven Architecture) can be used for asynchrony inside an individual component while retaining an easy-to-understand programming model. Between components, the principle is the same -- avoid synchronous coupling as much as possible. More often than not, the two components have no business talking directly to one another in any event. At every level, decomposing the processing into stages or phases, and connecting them up asynchronously, is critical to scaling.

Best Practice #5: Move Processing To Asynchronous Flows

Now that you have decoupled asynchronously, move as much processing as possible to the asynchronous side. In a system where replying rapidly to a request is critical, this can substantially reduce the latency experienced by the requestor. In a web site or trading system, it is worth it to trade off data or execution latency (how quickly we get everything done) for user latency (how quickly the user gets a response). Activity tracking, billing, settlement, and reporting are obvious examples of processing that belongs in the background. But often significant steps in processing of the primary use case can themselves be broken out to run asynchronously. Anything that can wait should wait.

Equally as important, but less often appreciated, is the fact that asynchrony can substantially reduce infrastructure cost. Performing operations synchronously forces you to scale your infrastructure for the peak load - it needs to handle the worst second of the worst day at that exact second. Moving expensive processing to asynchronous flows, though, allows you to scale your infrastructure for the average load instead of the peak. Instead of needing to process all requests immediately, the queue spreads the processing over time, and thereby dampens the peaks. The more spiky or variable the load on your system, the greater this advantage becomes.

Best Practice #6: Virtualize At All Levels

Virtualization and abstraction are everywhere, following the old computer science aphorism that the solution to every problem is another level of indirection. The operating system abstracts the hardware. The virtual machine in many modern languages abstracts the operating system. Object-relational mapping layers abstract the database. Load-balancers and virtual IPs abstract network endpoints. As we scale our infrastructure through partitioning by function and data, an additional level of virtualization of those partitions becomes critical.

At eBay, for example, we virtualize the database. Applications interact with a logical representation of a database, which is then mapped onto a particular physical machine and instance through configuration. Applications are similarly abstracted from the split routing logic, which assigns a particular record (say, that of user XYZ) to a particular partition. Both of these abstractions are implemented in our home-grown O/R layer. This allows the operations team to rebalance logical hosts between physical hosts, by separating them, consolidating them, or moving them -- all without touching application code.

We similarly virtualize the search engine. To retrieve search results, an aggregator component parallelizes queries over multiple partitions, and makes a highly partitioned search grid appear to clients as one logical index.

The motivation here is not only programmer convenience, but also operational flexibility. Hardware and software systems fail, and requests need to be re-routed. Components, machines, and partitions are added, moved, and removed. With judicious use of virtualization, higher levels of your infrastructure are blissfully unaware of these changes, and you are therefore free to make them. Virtualization makes scaling the infrastructure possible because it makes scaling manageable.

Best Practice #7: Cache Appropriately

The last component of scaling is the judicious use of caching. The specific recommendations here are less universal, because they tend to be highly dependent on the details of the use case. At the end of the day, the goal of an efficient caching system to maximize your cache hit ratio within your storage constraints, your requirements for availability, and your tolerance for staleness. It turns out that this balance can be surprisingly difficult to strike. Once struck, our experience has shown that it is also quite likely to change over time.

The most obvious opportunities for caching come with slow-changing, read-mostly data - metadata, configuration, and static data, for example. At eBay, we cache this type of data aggressively, and use a combination of pull and push approaches to keep the system reasonably in sync in the face of updates. Reducing repeated requests for the same data can and does make a substantial impact. More challenging is rapidly-changing, read-write data. For the most part, we intentionally sidestep these challenges at eBay. We have traditionally not done any caching of transient session data between requests. We similarly do not cache shared business objects, like item or user data, in the application layer. We are explicitly trading off the potential benefits of caching this data against availability and correctness. It should be noted that other sites do take different approaches, make different tradeoffs, and are also successful.

Not surprisingly, it is quite possible to have too much of a good thing. The more memory you allocate for caching, the less you have available to service individual requests. In an application layer which is often memory-constrained, this is a very real tradeoff. More importantly, though, once you have come to rely on your cache, and have taken the extremely tempting steps of downsizing the primary systems to handle just the cache misses, your infrastructure literally may not be able to survive without it. Once your primary systems can no longer directly handle your load, your site's availability now depends on 100% uptime of the cache - a potentially dangerous situation. Even something as routine as rebalancing, moving, or cold-starting the cache becomes problematic.

Done properly, a good caching system can bend your scaling curve below linear - subsequent requests retrieve data cheaply from cache rather than the relatively more expensive primary store. On the other hand, caching done poorly introduces substantial additional overhead and availability challenges. I have yet to see a system where there are not significant opportunities for caching. The key point, though, is to make sure your caching strategy is appropriate for your situation.

Summary

Scalability is sometimes called a "non-functional requirement," implying that it is unrelated to functionality, and strongly implying that it is less important. Nothing could be further from the truth. Rather, I would say, scalability is a prerequisite to functionality - a "priority-0" requirement, if ever there was one.

I hope that you find the descriptions of these best practices useful, and that they help you to think in a new way about your own systems, whatever their scale.

References

- eBay's Architectural Principles (video)
- Werner Vogels on scalability
- Dan Pritchett on You Scaled Your What?
- The Coming of the Shard
- Trading Consistency for Availability in Distributed Architectures
- Eric Brewer on the CAP Theorem
- SEDA: An Architecture for Well-Conditioned, Scalable Internet Services