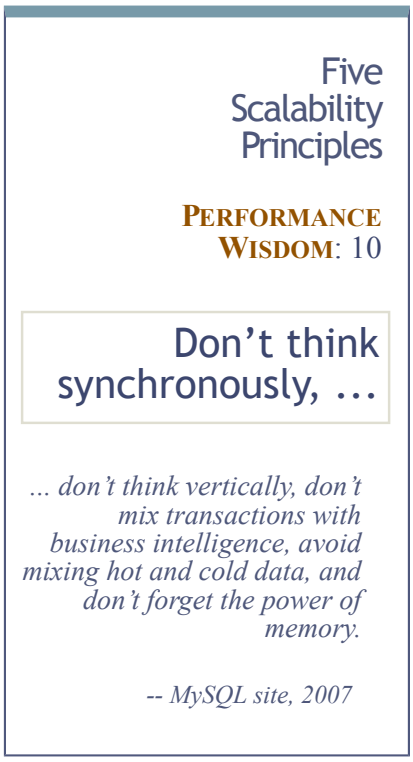
Five Scalability Principles

Friday, August 10, 2007 at 03:12AM

Chris Loosley in Articles and White Papers, Foundations of Performance, Optimization and Tuning, Performance Wisdom



The 12 Days of Scale-Out is a section of the *MySQL* site. It consists of a series of twelve articles, eleven of which are case studies describing large-scale MySQL implementations. *But Day Six* is a bit different -- it spells out five fundamental performance principles that apply to all application scaling efforts.

This subject is vitally important to MySQL, whose *server replication and high availability features ... allow high-traffic sites to horizontally 'Scale-Out' their applications, using multiple commodity machines to form one logical database -- as opposed to 'Scaling Up', starting over with more expensive and complex hardware and database technology*.

I know from first-hand experience that these claims are valid. At Keynote, my team used MySQL as the foundation for the **Performance Scoreboard**. In this **data mart** application, MySQL supports supports the continuous insertion of new measurements at the rate of several million per day, plus hourly aggregation into summary tables, plus the queries needed to support continually updated dashboard displays for every customer, plus any ad hoc queries generated by customers doing diagnostic investigations.

Learning the hard way

According to the article, MySQL's **database experts** ... *have seen many companies fall into a few common traps when they first design their systems, only to run into performance issues once the explosive growth hits*. So, adopting the ***anti-pattern* approach** to providing guidance, the article presents the principles as the **Top Five Scale-Out Pitfalls to Avoid**. These are:

- Don't think synchronously
- Don't think vertically
- Don't mix transactions with business intelligence
- Avoid mixing hot and cold data
- Don't forget the power of memory

It is common for people to get smarter about performance when they have to find and fix problems. I learned about database performance first-hand between 1970 and 1995, while designing and tuning IBM database systems -- first IMS, and then DB2. In the process I discovered -- often the hard way -- that all large computer systems are subject to the same principles. And over the years I ran across other authors and teachers who (not surprisingly)

had discovered the same things. Some had even invented memorable and insightful ways of describing their insights as "laws" or "rules".

When I wrote **High-Performance Client/Server**, I tried to capture these clever sayings as numbered guidelines. So in this post, I'm going to reprint each of MySQL's five scale-out pitfalls, followed by the corresponding guidelines and some related excerpts from the manuscript of that book (marked as "HPCL").

1. Don't think synchronously

> Thinking synchronous is the single biggest mistake in architecting a Scale-Out design. Generally, when load is added to an already-loaded system, some part of the system will become a bottleneck -- and response times will increase. In scale-out, with a large system consisting of multiple machines, thinking synchronously will add a lot of wait time and hurt performance. Any truly large scale-out design will have to introduce asynchronous communication, parallelization, and strategies to deal with approximate or slightly outdated data.
>
> --***Top Five Scale-Out Pitfalls to Avoid***, *The 12 Days of Scale-Out, Day 6*

Abandoning the Single Synchronous Transaction Paradigm

HPCL: The ... concept of a heterogeneous distributed database with synchronized updates is a vision of utopia that swims against the tide of computing technology. The tight controls over application processing that are possible on a mainframe are incompatible with many aspects of the move to widespread distributed processing... Decoupled processes and multi-transaction workflows are the optimal starting point for the design of high-performance enterprise client/server systems:

- **Decoupled processes.** Decoupling occurs when we can separate the different parts of a distributed system so that no one process ever needs to stop processing to wait for the other(s). The driving force behind this recommendation is the Bell's law of waiting: *All CPU's wait at the same speed* (Guideline 11.17)
- **Multi-transaction workflows.** Often, we can split up the business transaction into a series of separate computer transactions. We call the result a multi-transaction workflow. The motivation for this recommendation is the Locality Principle, *Group components based on their usage* (Guideline 10.1), and (one corollary of) the Parallelism Principle, *Workflow parallelism can lower perceived response times* (Guideline 13.7)

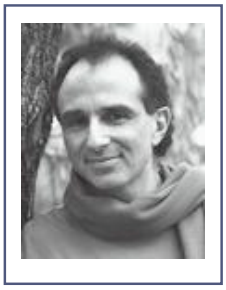
--***High-Performance Client/Server***, *Chapter 16: Architecture for High Performance, pp506-507*

2. Don't think vertically

> It's a mistake to think that a system can be grown by scaling vertically, that is, by buying bigger machines with more CPUs. Throwing more power at an existing

> implementation -- which is probably synchronous and most likely already suffering from lock waits -- is only going to make it 'wait faster'. By planning for horizontal scale-out, almost from the start, a business is already planning in the direction of distributed, asynchronous systems, which will make it easier to add more capacity later on.
>
> --***Top Five Scale-Out Pitfalls to Avoid***, *The 12 Days of Scale-Out, Day 6*

HPCL: In Chapters 11 and 12 we discussed the performance of shared resources, and how to minimize the delays caused by bottlenecks. When there is excessive demand for a single shared resource, one way to break the logjam is to divide and conquer. Dennis Shasha's database design principle, *Partitioning breaks bottlenecks* [1], concisely expresses this motivation for processing work in parallel.



[1] ***Dennis E. Shasha*** *in* ***Database Tuning: A Principled Approach***, *Prentice-Hall, 1992, p3*

The idea behind parallelism is simple: take several items of work and process them at the same time. Naturally, this is faster than processing the same work serially. But while it may be an obvious and attractive design technique, processing in parallel does introduce new problems of its own. Not all workloads can be easily subdivided, and not all software is designed to work in parallel. Processing related pieces of work in parallel typically introduces additional synchronization overheads, and in many situations produces new kinds of contention among the parallel streams. Connie Smith's version of the Parallelism Principle recognizes these complications: *Execute processing in parallel (only) when the processing speedup offsets communication overhead and resource contention delays* [2].

[2] ***Connie U. Smith***, *Performance Engineering of Software Systems, Addison Wesley, 1990, p55*

The Parallelism Principle

To sum up, Smith's and Shasha's two views nicely highlight the dilemma facing every designer when considering parallelism. Combining them gives us The Parallelism Principle (Guideline 13.1), *Exploit parallel processing*:

Processing related pieces of work in parallel typically introduces additional synchronization overheads, and often introduces contention among the parallel streams. Use parallelism to overcome bottlenecks, provided the processing speedup offsets the additional costs introduced.

--***High-Performance Client/Server***, *Chapter 13: The Parallelism Principle, pp382-383*

3. Don't mix transactions with business intelligence

> Many large systems are OLTP systems that do not have a data export phase inside the application. Therefore, they oftentimes contain vast amounts of business intelligence data. If an OLTP system, for the same number of users/articles/orders, grows over time, then it likely has a data warehouse or data mart struggling to get out. Separating the data onto different databases and/or

> servers will go a long way in improving performance for both the transactional application and analytic operations.
>
> --***Top Five Scale-Out Pitfalls to Avoid***, *The 12 Days of Scale-Out, Day 6*

HPCL: How are we to reconcile an immediate need to process large decision-support queries with an ongoing transaction workload that demands high throughput and short response times? The more resources consumed by queries, the fewer remain to process transactions, and the larger the impact of the query workload on transaction throughput.

On the other hand, if we try to maintain overall throughput by artificially restricting the resources allocated to queries, queries take much longer. This alone can cause political problems, unless expectations have been set properly. But there is a potential side effect that is even more damaging. If the DBMS holds any database locks for a long running query, subsets of the transaction workload may have to wait until the query completes, causing erratic transaction response times.

Inmon's rule

The usual resolution of this dilemma is to *avoid mixing short transactions with long-running queries* in the first place. I refer to this as *Inmon's rule* because the well known speaker and author **Bill Inmon** spent many years during the 1980s evangelizing this concept, making the rounds of the database user groups, talking about the importance of separating *operational* from *informational* processing.

Don't mix short transactions with long-running queries. When we have high-performance operational workloads, we should keep them on a separate processor, separate from ad hoc or unknown queries, which may have massive processing requirements. (Guideline 10.20)

--***High-Performance Client/Server***, *Chapter 10: The Locality Principle, pp310-311*

4. Avoid mixing hot and cold data

> Similar to #3 is mixing hot (frequently-changed) and cold (more static) data, especially when it comes to write activity. Since database writes are more difficult and expensive to scale, it is advisable to keep this type of data away from data that does not change that often. Again, separating the data onto different databases and/or servers can significantly enhance your application's performance.
>
> --***Top Five Scale-Out Pitfalls to Avoid***, *The 12 Days of Scale-Out, Day 6*

This one is interesting. In Chapter 12 on Database Locking, I discussed database hot spots:

HPCL: Although locking is essential for maintaining data integrity, excessive contention can occur when too many concurrent applications need to lock the same data item, or the same small set of data items. Database designers call this type of locking bottleneck a *hot spot*.

Because several applications are reading and writing to the same portion of a database, it is quite common for deadlocks to be caused by hot spots. A hot spot can arise for one of three reasons:

- **Natural hot spots exist in the business data.** It is very rare for activity to be evenly distributed across all areas of the business--recall the Centering Principle (*Guideline 4.9. Think globally, focus locally. This guideline is related to the **Pareto Principle**, or 80-20 rule.*). Sometimes, thanks to a highly skewed distribution of work, a large percentage of database updates apply to a small fraction of the business data. ... Automating these types of business applications tends to create database hot-spots naturally, unless we are careful to design the databases and the associated processing to eliminate them.
- **The application's design creates artificial hot spots.** When applications are designed to maintain the current value of a derived statistic like a sequence number, a total, or an average, instant hot-spots are created in databases because every instance of a program must read and update the same data item.
- **Locking protocols against physical data structures create artificial hot spots.** Even though applications are not manipulating the same data, they may well be reading and writing to the same physical data or index pages, which will cause contention if page level locking is being used. Common examples of this type of problem occur when all the rows in a table fit on a small number of pages, or when data is inserted sequentially on the last page of a table based on a time or a sequence number.

*--**High-Performance Client/Server**, Chapter 12: Database Locking, pp373-374*

All my recommendations (pp 374-380) involved ways to reduce database contention by *spreading out* the data elements that comprise the hot spot. In contrast, the MySQL suggestion is to *separate* hot (frequently-changed) and cold (more static) data altogether, and then to focus on special tuning to improve the performance of the hot data. While I did not propose this approach, I believe it does not conflict with my own advice. As I noted when discussing Inmon's rule, *Inmon's rule is not just for transactions and queries. It can be applied to any mix of workloads that have different performance characteristics.*

5. Don't forget the power of memory

> Data accessed in memory produces infinitely better response times than the same data accessed on disk. Once the most-often referenced/accessed data -- oftentimes called the "working set" -- exceeds the amount of available memory, a system runs the risk of becoming disk-bound, with the end result being poorer performance. When designing a Scale-Out architecture, care must be taken not to exhaust a single server's memory allocations so that it becomes disk bound. Instead, an application's working set must be smartly divided among the servers participating in a scale-out design so that data is always accessible in RAM.
>
> *--**Top Five Scale-Out Pitfalls to Avoid**, The 12 Days of Scale-Out, Day 6*

Substitute faster devices for slower ones

HPCL: The table (below) shows the typical hierarchy of computing resources found in an enterprise client/server environment, organized by relative speed with the fastest at the top. To improve an application's performance, we must look for design changes that will make its resource usage pattern migrate upwards in the hierarchy.

Device Type	Typical Service Time	Relative to 1 second
High Speed Processor Buffer	10 nanoseconds	1 second
Random Access Memory	60 nanoseconds	~6 seconds
Expanded Memory	25 microseconds	~1 hour
Solid State Disk Storage	1 millisecond	~1 day
Cached Disk Storage	10 milliseconds	~12 days
Magnetic Disk Storage	25 milliseconds	~4 weeks
Disk via MAN/High Speed LAN Server	27 milliseconds	~1 month
Disk via Typical LAN Server	35-50 milliseconds	~6-8 weeks
Disk via Typical WAN Server	1-2 seconds	~3-6 years
Mountable Disk/Tape Storage	3-15 seconds	~10-50 years

Substitute memory and processor cycles for disk I/O

Database caching or buffering reduces the cost of re-reading frequently reused portions of a database by retaining them in memory, improving responsiveness by trading off memory for processor resources and disk I/O. Of course, the costs of searching for data in cache are all wasted overhead when the data isn't actually there. This situation is termed a cache miss. The relatively small overhead of cache misses must be weighed against the larger savings we get whenever there is a cache hit. Typically, no matter how large the cache, cache hits consume fewer processor cycles and a lot less time than would the corresponding disk I/O.

*--**High-Performance Client/Server** , Chapter 14: The Trade-off Principle, pp432-433*

Parallel processing can be done on machines with SMP, MPP, or various hybrid architectures ... Regardless of the hardware architecture, the objective is to increase processing power by adding more processors. But depending upon both the workload and how the processors and other devices are interconnected, other hardware constraints, for example, the memory or I/O bus, can prevent the full exploitation of the additional processors. If some other component is the bottleneck, we obtain no benefit from adding more processors--in fact, we may even make the bottleneck worse.

One solution is to remove the constraint by giving each processor its own memory, and/or its own disk subsystem. This solution, however, complicates system software development. For example, database software must coordinate the contents of in memory data caches across the different processors if each processor has its own dedicated memory. Removing hardware constraints may simply move the problem to the software, unless the software has been specially written to overcome these problems.

*--**High-Performance Client/Server** , Chapter 13: The Parallelism Principle, pp417-418*

Your contributions ...

I've included just a few extracts from my book that came to mind as I was compiling this post. If you know of other informative discussions of the same principles, in a book or on the Web, please post a comment below sharing your knowledge. Of course, questions or observations of your own are also welcome.

Tags: ***scalability***, ***MySQL***, ***database***, ***performance***, ***performance wisdom***, ***Data mart***, ***Dennis Shasha***, ***Connie Smith***, ***Parallelism***, ***hot spot***, ***Bill Inmon***, ***Pareto Principle***, ***80-20 rule***, ***working set***, ***cache***, ***response time***, ***computer memory***, ***Performance Matters***, ***application performance***
