WebAnalytics

# WEB APPLICATIONS PERFORMANCE OPTIMISATION

April 1, 2022

# Contents

# Chapter 1
# How Do You Approach Optimising Web Application Performance?

## 1.1 Be Clear about the Objective

Very often the objectives for optimisation or remediation are left fuzzy, but making people happy in a crisis is not an actionable goal and does not provide guidance for prioritising work. On the other hand, delivering unspecified but measurable improvement in some number of areas as part of an on-going process of improvement is a perfectly reasonable aim. The objective and the problem need to be matched.

There are really only two kinds of objective in this area: meeting some quantitative system specification or reducing user pain. Improving response time, fitting an application into a hardware budget, improving scalability, or reducing variability in response time could all fall into either category, they are the next level-down in the problem solving process. Prioritisation for pain-reduction work will include reducing the occurrence of large outliers as much as it is given to reducing times overall; it requires some understanding of how users react to performance and the immediate objectives will need to be re-visited frequently. For a quantitative objective, there needs to be an understanding of the interaction of system components (a model, more or less explicit depending on the complexity of the system and the experience of the people doing the work) and a clear awareness of the options and probability of success.

The ideal operational model in either case involves splitting the work into an acute-phase activity to reach some agreed immediate goals followed by an on-going and probably fairly low-level and low cost optimisation and maintenance activity. The reason for this is that system performance changes as data and workload evolve, even without changes to the software. Staying on top of that evolution and making regular small improvements to performance ensures that existing irritations are gradually removed and that the system does not accumulate new ones.

Regardless of the objective, this paper is about how to identify where problems are, how to work out what changes will provide some benefit, and how to check that you got the benefit you expected from the changes that you made.

There are a few points that should be kept in mind:

- The first is that *elapsed time is a proxy for system resource consumption*. While the user is waiting for a response, the running transaction is consuming system capacity of some kind. This may be CPU or IO, but it can also be locking bandwidth or delays associated with memory. This means that for most system designs, system load correlates with transaction elapsed time. Reduce one and you reduce the other.
- Another key idea is that *the relationship between response time and resource consumption is not linear*. Elapsed time rises sharply, non-linearly, when some component of the system capacity becomes overloaded: small changes in load can cause large changes in response time. This is why some transaction-level problems are not visible in development (with no competition for resources) or at low levels of load, but emerge quickly as soon as the load rises high enough to cause contention

for some component of the system's capacity. Load can result in contention for resources and contention has a cost, so reducing load can reduce contention which in turn can reduce load. This occurs most obviously when memory demand causes paging, but is also the case with database or application locking, database log contention, garbage collection or too high processor demand.

- System remediation is likely to require multiple production changes. Trying to avoid deployments by bundling everything into a single release risks significant work being spent on fixes which may not work and depends on no new issues being discovered.

The discussion in this paper is based on use of the CRAN WebAnalytics package, but the general principles can be applied using other tools. The WebAnalyics package focusses on the initial problem measurement in systems that do not have complex (and expensive) monitoring infrastructure supporting them, and is also useful for analysis of long term changes in performance that may be difficult to carry out with more complex tools.

## 1.2 Be Systematic

Once you have clarity about the problem, approaching the problems systematically is vital. A useful model for structuring this type of problem solving is the Six Sigma DMAIC model. The key idea in DMAIC, and the key difference from a lot of activity in practice, is the focus on measurement. DMAIC is a cycle consisting of:

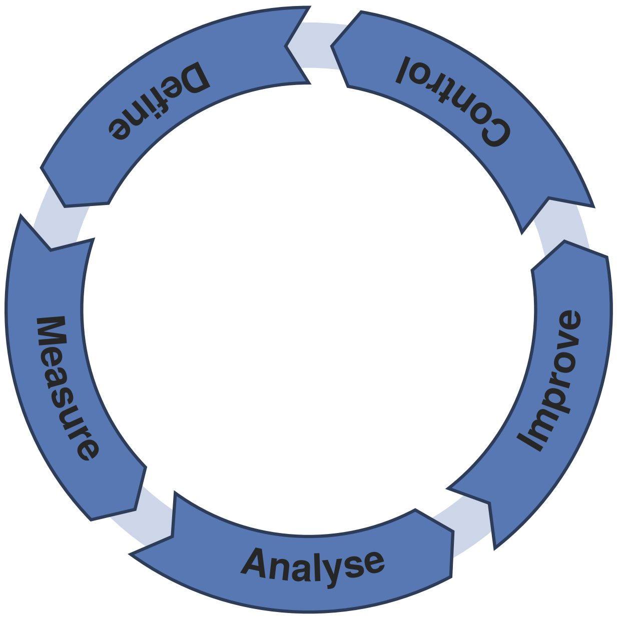
**Define:** be very clear about what the problem is. Make sure that you have identified all of the symptoms. Do not discard symptoms because they do not appear to be related to the problem.

**Measure:** remedial action must be based on measurement and analysis. The temptation to start to fix problems is often very strong, but it has to be resisted until measurement has been done, analysis indicates which fixes have the highest return on effort, and the fixes have been prioritised.

**Analyse:** identify the problems, the underlying technical causes of the problems and the mechanisms by which delays are introduced. An output of the analysis is an understanding of what you are going to get from improvements and how they are likely to interact with other parts of the system.

**Improve:** apply changes to the system and measure what you achieved.

**Control:** maintain the system to ensure that the problems do not recur and continue to carry out regular measurement.



Performance problem solving iterates because understanding performance is iterative: the process starts with clarifying top-level problems and breaking them down to technology problems which are themselves broken down into their components, each one of which requires one or more cycles of measurement. Complex systems will also typically have multiple problems and the relative importance of problems changes as they are resolved. For example if you have high concurrency as a result of a poorly performing transaction, this may have the side effect of causing database contention or IO contention and may make pre-existing race conditions appear more frequently. When you fix that concurrency problem, a different problem may become visible and the priorities are likely to change. For these reasons, priorities should be re-evaluated after each iteration, with new measurements and analyses carried out to support them.

# Chapter 2
# Define the Problem

Start by collecting known problems with the system, those that are clearly performance and capacity related and also those that appear not to be, those that have been formally recorded and those that are more vaguely described. Performance problems in an application can trigger odd symptoms and defects can cause performance problems. Once there is a view of the problem space, the targets can be defined: what do we *have* to achieve and what *should* we achieve. The objective will usually be some combination of improving response time, fitting an application to a processor, RAM and IO budget and making the application work as well as possible for its users. All of these interact, the better the response time you achieve, the less hardware that will need to be dedicated to the application and the less annoyingly variable the response time is likely to be.

## 2.1 Target Response Times

Often, system specifications and vendor contracts will specify system response times in a fairly loose way - mean response times measured in seconds are common, mean response times calculated over defined intervals are better and also common, with the mean usually being the response time that the client wants to see as the most common time, perhaps with some more or less random extra amount added for luck.

What environment the times are measured in may or may not be specified, load test environments tend to be very well behaved, outlier values in production will usually be larger than those in in a test environment. Ideally performance targets do not include WAN times because these can be variable, but if they are included, there will need to be specification of request/response sizes and agreed transfer times.

Sadly, the client doesn't get what they want from either way of specifying the response time. The mean will be somewhere above the 50th percentile (more than half of the times will be under the mean) but that says almost nothing about the distribution of high times.

### 2.1.1 Use Percentiles Not Means for Response Time

The problem with mean values is that they do not represent variability, and they hide outliers. A mean response time of 3 seconds can be achieved by a system with 90% of requests taking 1 second and 10% taking 21 seconds, which is probably not what is wanted. This kind of distribution is actually quite common for a system with significant performance problems. This was the problem seen in a large retail site, a small percentage of transactions were displaying very poor response times and the majority were processing very quickly. The mean response time in that system was something like 3.5 seconds with the poorly performing transactions (a subset of product searches, about 1% of the total) taking 30 to 300 seconds to complete. Contractually that system's performance was within tolerances, performance was specified as a daily mean response time, but in practice it was a major problem.

How should response times be represented? This depends which side of the fence you are on. As a software supplier, a mean response time calculated over 24 hours or a week or a month will look just fine and represent a fairly low risk. As an end user, you want the response time to be small and the variability of the response times to be both minimised, and to be understandable. For example users are unsurprised if a report containing a substantial amount of data does not appear instantly, but are likely to wonder what is going on if a login page takes more than a fraction of a second to display. It is reasonable to define separate categories of transaction performance, interactive transactions, transactions that have time made up mostly of communication with slow external systems, and reports, are not unusual.

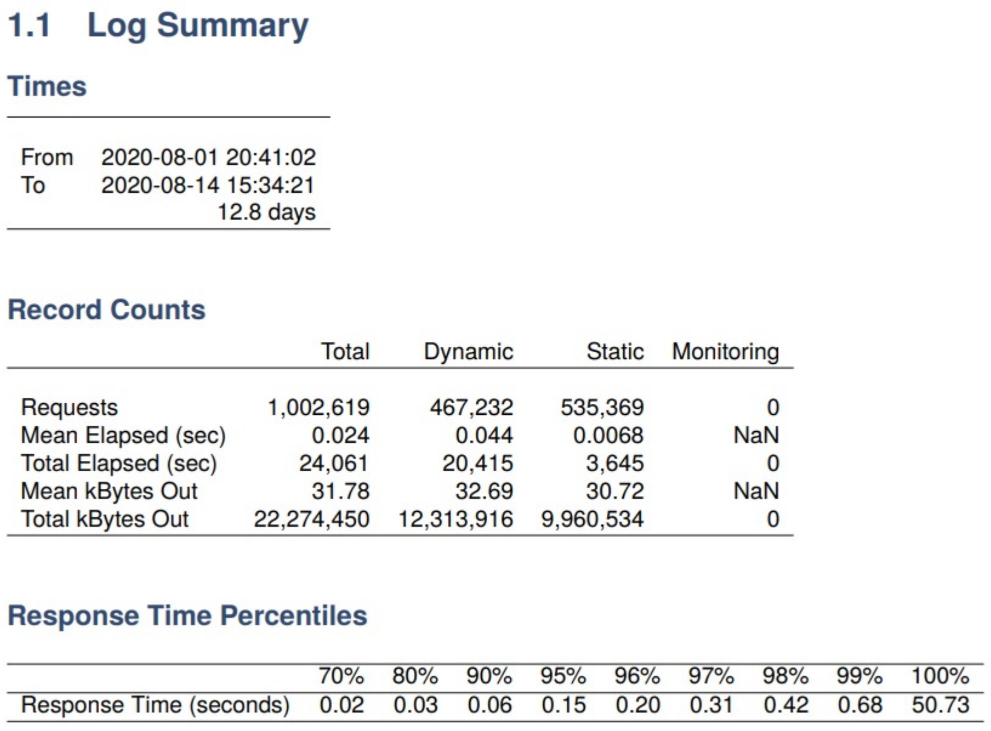
Percentiles do not suffer quite so badly from the same problems as mean values. The promise that they make, that a user will see a defined percentage of responses will take less than whatever the threshold is, remains true over any reasonable interval and can be experienced by the user directly. What a mean value means to a user over an hour, a day or a week is harder to see. A workable approach is to define a target response time for an application and specify that as a percentile of the overall response times. A 90th percentile of 3 seconds means that one in ten transactions can exceed 3 seconds. This does not consider the mean, just that 90% of transactions must complete in under 3 seconds. Sometimes a maximum response time is specified as well, but in practice that threshold cannot be guaranteed for complex applications in a production environment, and is usually entirely achievable in a test environment, so it acts as a check on production performance more than a meaningful target in test.

There is still the problem of dilution of outliers, but quantifying what is acceptable or actionable in production is very difficult because transient events in the environment (Internet or WAN) can cause delays outside of the control of the application, leading to large response time variables.

# Chapter 3 Measure the Problem

There are two kinds of measurement, measurement of the symptoms, and later, after analysis, more detailed measurement of the system to find the cause of the symptoms. There is a degree of overlap between symptom measurement and cause measurement. In measuring the symptoms you are looking for quantitative data that corresponds with the problem reports which will be needed to track progress. For performance problems this is usually elapsed time, and assuming that the system performed acceptably in development, the problem is likely to be in the application not in the client or network. Begin by checking that there are no known page rendering time delays in the application that could be being reported as problem. Measurement at the browser is harder than measurement at the server, particularly for Internet-facing applications.

The WebAnalytics Package provides a number of views of system performance that help with characterising the performance and identifying which transactions contribute to the problem. The top level view is:



## 1.1 Log Summary

### Times

| | |
|---|---|
| From | 2020-08-01 20:41:02 |
| To | 2020-08-14 15:34:21 |
| | 12.8 days |

### Record Counts

| | Total | Dynamic | Static | Monitoring |
|---|---|---|---|---|
| Requests | 1,002,619 | 467,232 | 535,369 | 0 |
| Mean Elapsed (sec) | 0.024 | 0.044 | 0.0068 | NaN |
| Total Elapsed (sec) | 24,061 | 20,415 | 3,645 | 0 |
| Mean kBytes Out | 31.78 | 32.69 | 30.72 | NaN |
| Total kBytes Out | 22,274,450 | 12,313,916 | 9,960,534 | 0 |

### Response Time Percentiles

| | 70% | 80% | 90% | 95% | 96% | 97% | 98% | 99% | 100% |
|---|---|---|---|---|---|---|---|---|---|
| Response Time (seconds) | 0.02 | 0.03 | 0.06 | 0.15 | 0.20 | 0.31 | 0.42 | 0.68 | 50.73 |

With a variety of more detailed data available after that.

## 3.1 Measure the Symptoms

People are bad at identifying performance problems, they miss instances of good performance, tend to over-state some problems and under-state others and complex patterns are missed completely, what they are good at doing is identifying their own pain. People react to unpredictable response times more badly than they do to consistent but slow response times. Quantitative measurement of the behaviour is necessary to make sense of this, and the first level of measurement, done in detail, must be the response time of the system. This measurement is then used to prioritise remediation and to identify what gets measured and analysed next. It is also used as a baseline to compare fixes with.

For a web application the place to start is the response times of the requests as seen at the system front-end, the web server. Network infrastructure problems can cause response time problems for clients, but these are not common and will become obvious once the server response time is measured (there is no reason to start by looking at the unlikely problems).

There are many tools for measuring performance, the WebAnalytics package is oriented toward those applications that do not have supporting monitoring infrastructure and uses data from Web Server logs which are usually already being written, so adding the elapsed time counter to them has effectively zero overhead. The package also provides a highly detailed comparison of baseline and current data, making identification of changes very easy.

## 3.2 Measure the System

Its not actually necessary to measure everything that might be the problem at once, An analytical approach, identifying and adding up the times of components and works well. The overheads of something like Windows Perfmon collecting and logging a few counters are usually overstated, monitoring tools like DynaTrace and AppDynamics can impose a high overhead on a system and this can be a severe problem for very heavily loaded systems. The best approach, if you don't have a detailed monitoring tool in place is to identify where the problem is and then collect statistics in just that area. Summary statistics collection using Perfmon on a five second interval will represent no more than one percent or so of the capacity of a smallish system. Similarly on Linux vmstat or iostat on a 1 second interval logging to a file represents a negligible overhead and provides a useful summary of resource consumption.

### 3.2.1 Time Components

The time components that are likely to be interesting are:

- Client time - DNS lookup, rendering, Javascript processing time - this is client specific and for any one transaction is likely to be more or less constant, its not impacted by concurrency or usually by back-end server performance. It can be measured using the developer tools now provided by many web browsers. Javascript components fetched from or interacting with third party servers are one potential source of problems, these can have intermittently very poor performance that users will see as poor application responsiveness. Unless there is some reason to think otherwise, treat this as a constant overhead.
- Network time - for a public-facing web application this will include home network time, ISP network time, network between ISPs, the time across the network that the application is embedded in and the network time across the DMZ that the application is hosted in. The behaviour of these networks in combination is complex and they must be regarded as unreliable, resulting in hugely variable performance that often cannot be effectively diagnosed. It is frustrating to have a problem raised

saying that an important external user experienced poor response time at a point in time (and have in that case partial DynaTrace data supplied), but not be able to find any evidence of it in the response times of the system itself. In that case it appeared to be a transient network issue somewhere between the system and that user's home PC. In a corporate WAN environment these factors are less important, but they are not absent.

It is possible for network infrastructure problems to impact performance, and these can be isolated by checking metrics at the devices themselves (packet drops, retransmits), measuring response time across different network segments to avoid specific network devices, and running traces using tools like Wireshark. This type of problem is often application data rate or response size related but is possibly also triggered by external factors like network backups or can be the result of randomly failing hardware. We still occasionally see full and half duplex type problems but these are rare now.

- Web Proxy or Web Server time - often there will be a reverse proxy or web server acting as a front-end to the application and this provides a useful point to collect elapsed time measurements. This the point where you are likely to have good control over the system performance. For IIS the elapsed time is the time between the arrival of the first byte of the request and the last TCP send or the ACK to that send (depending on the IIS version), for Apache it is the time between reading and interpreting the request line, and the last send of the response, an immaterial difference between the two unless there are very severe problems with the network such as contention, or the request is extremely large. Note that the timestamp recorded in the IIS log is the time that the response send completed, for Apache version 1 it is the time that the request was completed and for Apache 2 it is the time of receipt of the request, for long running transactions and analysis of concurrent transactions this can make a difference to how you interpret the data.
- Web Application Server time - if time measured at this point is not the majority of the elapsed time of a request then may be something wrong with the infrastructure. In an IIS-based system this is the web server.
- Memory Management - garbage collected heaps in both Java and .Net can have significant impact on system response time and throughput, but parallel garbage collection has reduced the severity of that problem. Understanding the overheads and introduced pause times is however still very important. The trade-offs between full collections and young generation collections may not work in the expected way, for systems with a high proportion of long lived large objects it may be desirable to minimise the size of the young generation space.
- Back-end systems time - database or web service time. Database time is usually a large fraction of the application server time and it is useful to be able to account for total SQL time. SQL Server profiler can be used to collect this type of information and traces from Oracle are also available. SQL Server has a number of interesting delay related performance counters and the Oracle Enterprise Manager ARM reports have top lists and delay metrics available too, but the same information can be got from the V$ statistics tables with a little extra work.
- Storage IO time - typically database IO, this is one area where IO delay or contention (database log contention or log IO saturation) can have a large effect on performance. On Windows this is best identified through seconds per write and seconds per read counters along with reads and writes per second. The disk data rate counters can also be used to identify contention which sets in as the IO channel becomes congested due to to high a transaction or data rate. On Linux, iostat can be used to obtain similar information but in both cases care needs to be taken in interpreting the percentage utilisation metric whose definition originated in a period when disks were not capable of parallel operations.

# Chapter 4
# Analyse the Problem

The main questions are: "what matters?" and "what is most likely to pay-off?". Is the problem system wide or localised to a number of transactions? How much improvement do you need and where? Are you likely to get enough improvement out of the problem areas you have identified? Keep in mind that fixing very high cost transactions will improve the performance of other transactions, but you need to be able to identify the mechanism that connects the performance of the transactions in question. For that connection you need a model of the system. This can be as simple as a back-of-envelope calculation of x number of requests cost y CPU each so total CPU demand over a period is Z which is comfortably, or not, less than the CPU available, or it may include estimates of costs (CPU, IO) across the technology stack with some queueing modelling applied (the PDQ-R queuing package is useful for this) and a more sophisticated view of peak load.

The saying, that "all models are wrong but some are useful" applies. A model will fail in a number of ways: poor inputs - you don't know what the workload really is over short intervals because the mean transaction costs over a sort interval can be a long way from the mean overall and the transaction rate can be very unpredictable, or an incomplete model - contention will do funny things to the service demand, pushing it up in some areas and pulling it down in others and missing parts may be more important than expected. What you do get from modelling is an understanding of the ballpark that the system lands in and you can do some sensitivity modelling from there: are you dead already? What range of transaction costs can the system sustain? What range of request rates can the system sustain? What parts of the system are at risk if there is a pile-up of work in the system? What is the transaction cost proportional to? Some examples of problematic systems only understandable though having at least a mental model of what is going on:

- transaction cost proportional to the number of users - a fat-client system with an update notification system that caused transactions to be re-executed by each client. The incremental throughput of each added client rose quickly initially and then fell, eventually becoming negative. This was a COTS package, the behaviour was uncovered through analysis of incremental throughput as clients were added and the fix required changes to the application.
- a system dependent on a very high cache hit ratio on infrequently accessed pages (cost of uncached requests was many times the cost of cached requests) but the cache was not distributed, so adding servers to handle the workload increased the system load by lowering the cache hit ratio, resulting in worse and worse performance as the cluster was expanded. This was a COTS package with extensive customisation, a small number of requests displayed two response times, one, most of the requests completing very quickly with a smaller number orders of magnitude larger. In this case, the system load would fall over time as the cache was populated. The pages were rewritten to make them faster to create, make them more cacheable, and a process was added to pre-populate the cache.
- a system where transactions were received through a web server, passed to an application server and then routed back to the web server to serve objects that were incorporated into the pages in the application server (web server load was a multiple of the external request rate and contention in the web server meant that the multiple secondary or back-end requests would not be completed, preventing more work from entering the system, a case of contention causing run-away contention).

- there are numerous examples of systems where a high IO-cost database access (a table scan on a not very large table) is not visible until the system goes into production. In development the response time is slow but not extremely slow and the database caches the table data after a burst of high IO. In production the database buffer pools are not so empty and concurrent hits on that access path cause a multiple of the development environment's response time. If you are unlucky, the transactions run often enough that the data is fetched over and over, slowing down later fetches so that they overlap with following fetches, in the worst case the transaction may be running continually. In a Microsoft CRM example, this effect combined with a much higher than expected request rate from another service caused the response time of that transaction to grow to about 20 to 40 times the time seen in production. The fix in this case was to improve the response time though better database table indexing and to cache the results of the service call for a period of time so that rapidly arriving requests were served quickly and did not load down the database.

Having measured the system behaviour and identified the areas needing improvement through consideration of a model of the system, the top-level performance problems can be broken down into lower-level, more technology oriented problems that can then be themselves measured and analysed. If there is a system-wide problem, look at things like heap management (garbage collection), database log contention, slow IO, excessive CPU demand and related issues. If the problem is localised to a handful of transactions then look for commonalities between them (similar slow database accesses, database object contention, common code) and estimate how big a benefit you might get from improving them. Is it big enough? How much improvement do you need?

## 4.1 Hygiene

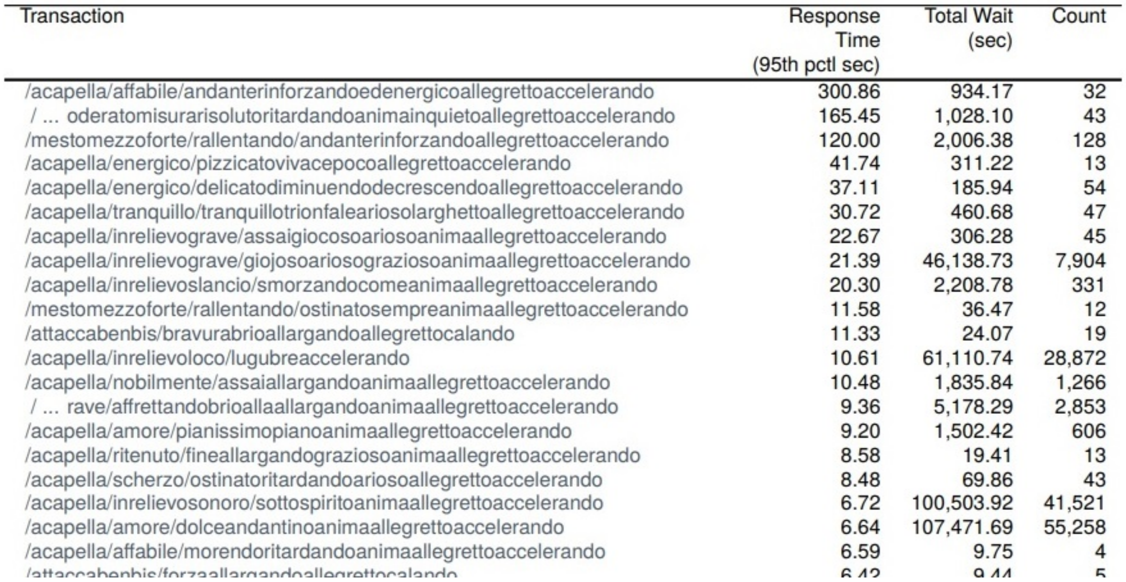
There is a great deal of material around about the design of web pages to maximise their performance including compression, minification, caching, the use of CDNs and the like. Some of those are just good practice, they are hygiene issues that may have been incorporated during development. There is a good deal about writing clean efficient code, and that type of hygiene also needs to be applied but at the moment that you are trying to optimise the application, the time for wholesale code change is past. Many development teams attempt to deal with performance problems by applying good code hygiene after the fact, applying changes that "everyone knows" will improve performance. Sometimes that works, but that approach tends to waste time because it isn't based on an assessment of how big the pay-off is. While good hygiene helps avoids illnesses, you're unlikely to get better by washing your hands more when you are sick.

## 4.2 Some Things to Look For

The kinds of behaviours that you should look for are:

- Which are the High Response Time Transactions?

  This question is basically, which transactions are a problem. Simply selecting transactions that have the highest elapsed times will usually identify these.

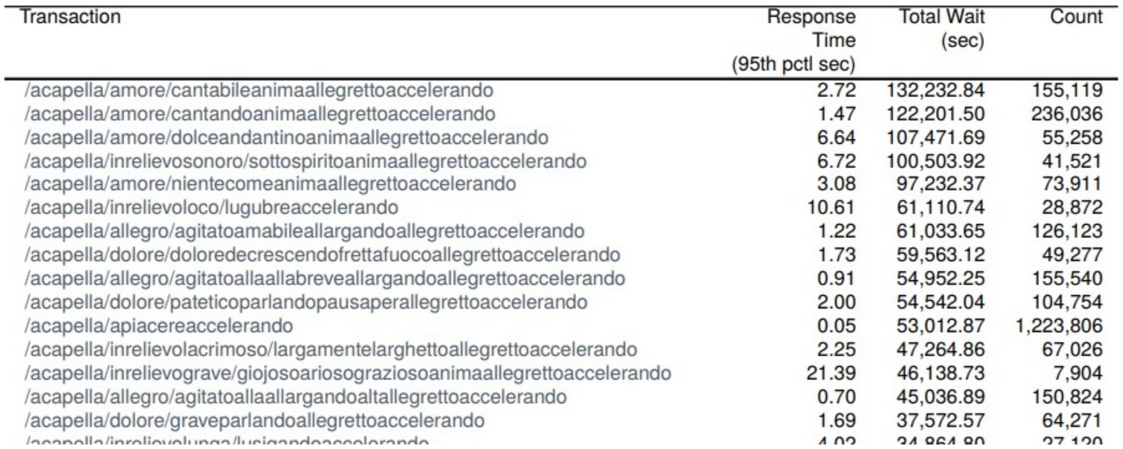


| Transaction | Response Time (95th pctl sec) | Total Wait (sec) | Count |
|---|---|---|---|
| /acapella/affabile/andanterinforzandoedenergicoallegrettoaccelerando | 300.86 | 934.17 | 32 |
| / ... oderatomisurarisolutoritardandoanimainquietoallegrettoaccelerando | 165.45 | 1,028.10 | 43 |
| /mestomezzoforte/rallentando/andanterinforzandoallegrettoaccelerando | 120.00 | 2,006.38 | 128 |
| /acapella/energico/pizzicatovivacepocoallegrettoaccelerando | 41.74 | 311.22 | 13 |
| /acapella/energico/delicatodiminuendodecrescendoallegrettoaccelerando | 37.11 | 185.94 | 54 |
| /acapella/tranquillo/tranquillotrionfaleariosolarghettoallegrettoaccelerando | 30.72 | 460.68 | 47 |
| /acapella/inrelievograve/assaigiocosoariosoanimaallegrettoaccelerando | 22.67 | 306.28 | 45 |
| /acapella/inrelievograve/giojosoariosograziosoanimaallegrettoaccelerando | 21.39 | 46,138.73 | 7,904 |
| /acapella/inrelievoslancio/smorzandocomeanimaallegrettoaccelerando | 20.30 | 2,208.78 | 331 |
| /mestomezzoforte/rallentando/ostinatosempreanimaallegrettoaccelerando | 11.58 | 36.47 | 12 |
| /attaccabenbis/bravurabrioallargandoallegrettocalando | 11.33 | 24.07 | 19 |
| /acapella/inrelievoloco/lugubreaccelerando | 10.61 | 61,110.74 | 28,872 |
| /acapella/nobilmente/assaiallargandoanimaallegrettoaccelerando | 10.48 | 1,835.84 | 1,266 |
| / ... rave/affrettandobrioallaallargandoanimaallegrettoaccelerando | 9.36 | 5,178.29 | 2,853 |
| /acapella/amore/pianissimopianoanimaallegrettoaccelerando | 9.20 | 1,502.42 | 606 |
| /acapella/ritenuto/fineallargandograziosoanimaallegrettoaccelerando | 8.58 | 19.41 | 13 |
| /acapella/scherzo/ostinatoritardandoariosoallegrettoaccelerando | 8.48 | 69.86 | 43 |
| /acapella/inrelievosonoro/sottospiritoanimaallegrettoaccelerando | 6.72 | 100,503.92 | 41,521 |
| /acapella/amore/dolceandantinoanimaallegrettoaccelerando | 6.64 | 107,471.69 | 55,258 |
| /acapella/affabile/morendoritardandoanimaallegrettoaccelerando | 6.59 | 9.75 | 4 |
| /attaccabenbis/forzaallargandoallegrettocalando | 6.42 | 9.44 | 5 |

This table from the WebAnalytics package shows which transactions have high 95th percentile response times along with their total elapsed time and request counts. URLs with high values for all of these metrics, like the 21 second 95th percentile response time transaction that occurs nearly 8,000 times and is responsible for 46,000 seconds of wait time is a good startting point.

- Which are the high total delay/cost transactions?

  Optimising slow transactions that are hardly used is not a good use of time and not all transactions can have their performance improved without a lot of effort or redesign. Comparing the slow transactions with the transactions that consume take significant time in total gives you a set of targets to work on that have some chance of providing a good ROI for the development effort. For a resource constrained system, small optimisations of very high frequency transactions can provide some relief, but keep in mind that a system under stress is likely to have some amount of unmet demand that will move the problem to another part of the system when the constraints are removed.

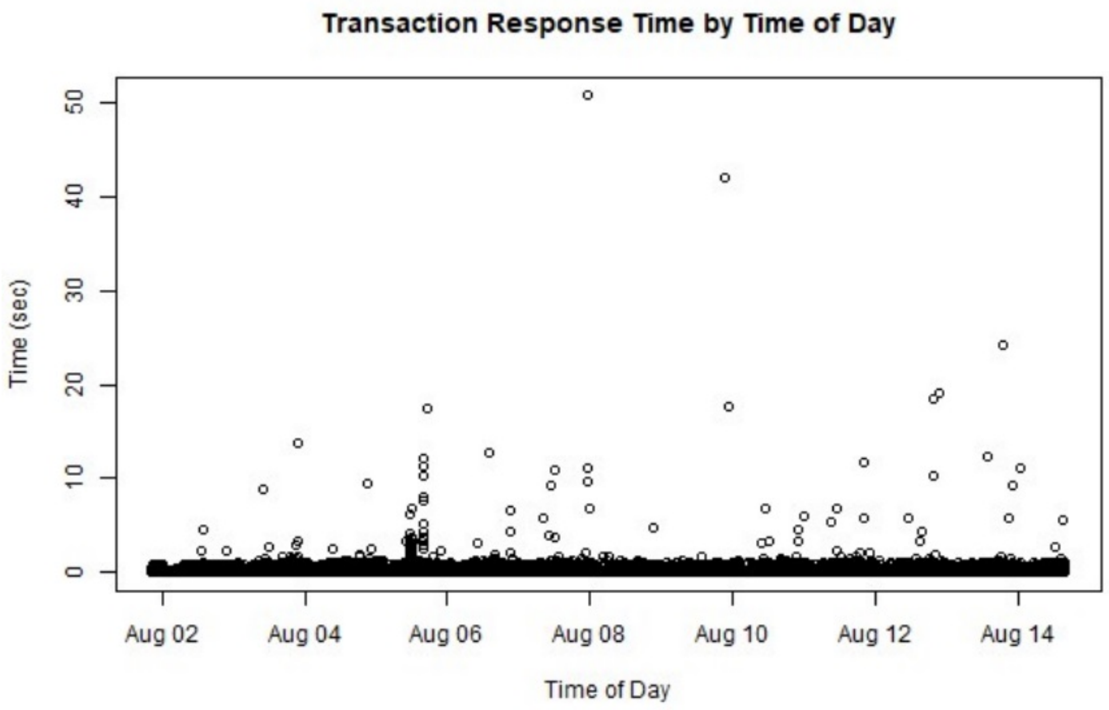


| Transaction | Response Time (95th pctl sec) | Total Wait (sec) | Count |
|---|---|---|---|
| /acapella/amore/cantabileanimaallegrettoaccelerando | 2.72 | 132,232.84 | 155,119 |
| /acapella/amore/cantandoanimaallegrettoaccelerando | 1.47 | 122,201.50 | 236,036 |
| /acapella/amore/dolceandantinoanimaallegrettoaccelerando | 6.64 | 107,471.69 | 55,258 |
| /acapella/inrelievosonoro/sottospiritoanimaallegrettoaccelerando | 6.72 | 100,503.92 | 41,521 |
| /acapella/amore/nientecomeanimaallegrettoaccelerando | 3.08 | 97,232.37 | 73,911 |
| /acapella/inrelievoloco/lugubreaccelerando | 10.61 | 61,110.74 | 28,872 |
| /acapella/allegro/agitatoamabileallargandoallegrettoaccelerando | 1.22 | 61,033.65 | 126,123 |
| /acapella/dolore/doloredecrescendofrettafuocoallegrettoaccelerando | 1.73 | 59,563.12 | 49,277 |
| /acapella/allegro/agitatoallaallabreveallargandoallegrettoaccelerando | 0.91 | 54,952.25 | 155,540 |
| /acapella/dolore/pateticoparlandopausaperallegrettoaccelerando | 2.00 | 54,542.04 | 104,754 |
| /acapella/apiacereaccelerando | 0.05 | 53,012.87 | 1,223,806 |
| /acapella/inrelievolacrimoso/largamentelarghettoallegrettoaccelerando | 2.25 | 47,264.86 | 67,026 |
| /acapella/inrelievograve/giojosoariosograziosoanimaallegrettoaccelerando | 21.39 | 46,138.73 | 7,904 |
| /acapella/allegro/agitatoallaallargandoaltallegrettoaccelerando | 0.70 | 45,036.89 | 150,824 |
| /acapella/dolore/graveparlandoallegrettoaccelerando | 1.69 | 37,572.57 | 64,271 |
| /acapella/inrelievolunga/lusigandoaccelerando | 4.02 | 34,864.80 | 27,120 |

The top list for aggregate response time shows that there are other transactions, beside the 21 second response instance, those at 6 and 10 second 95th percentile response times, that may be worth investigating to remove workload form the system.

- Does Response Time Change Over Time?

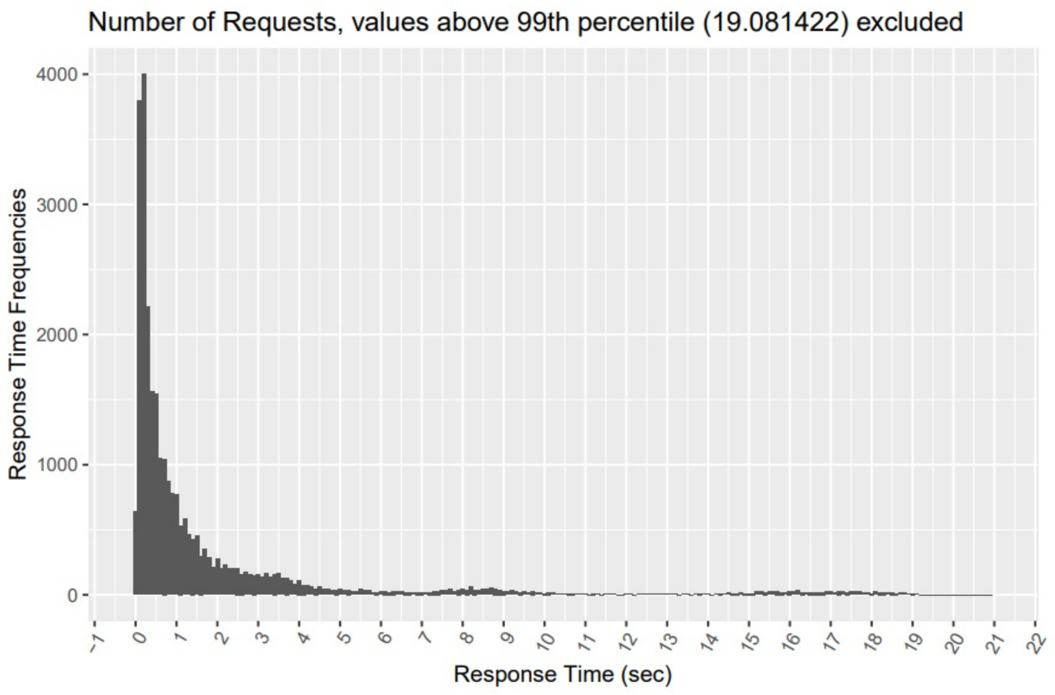
  Do the performance problems occur as random incidents during the day or do they occur regularly? Is there a time of day when they tend to happen? Network backups can have significant effects on overall systems performance, but will impact high disk IO or high network IO transactions more than others.



  The graph above indicates that there are rare excursions in the response time of that that do not seem to be cyclical. The report can be re-run filtering data down to individual days to look for daily patterns.

- Are there Multiple Identifiable Response Times for a Single Transaction?

  Developers may be certain that a transaction performs fine in development but is inexplicably slower in the production environment. Being able to say that there are two, or three, or more distinct response times for a transactions can help with understanding how many code paths there are and whether they have all been tested and optimised in development.

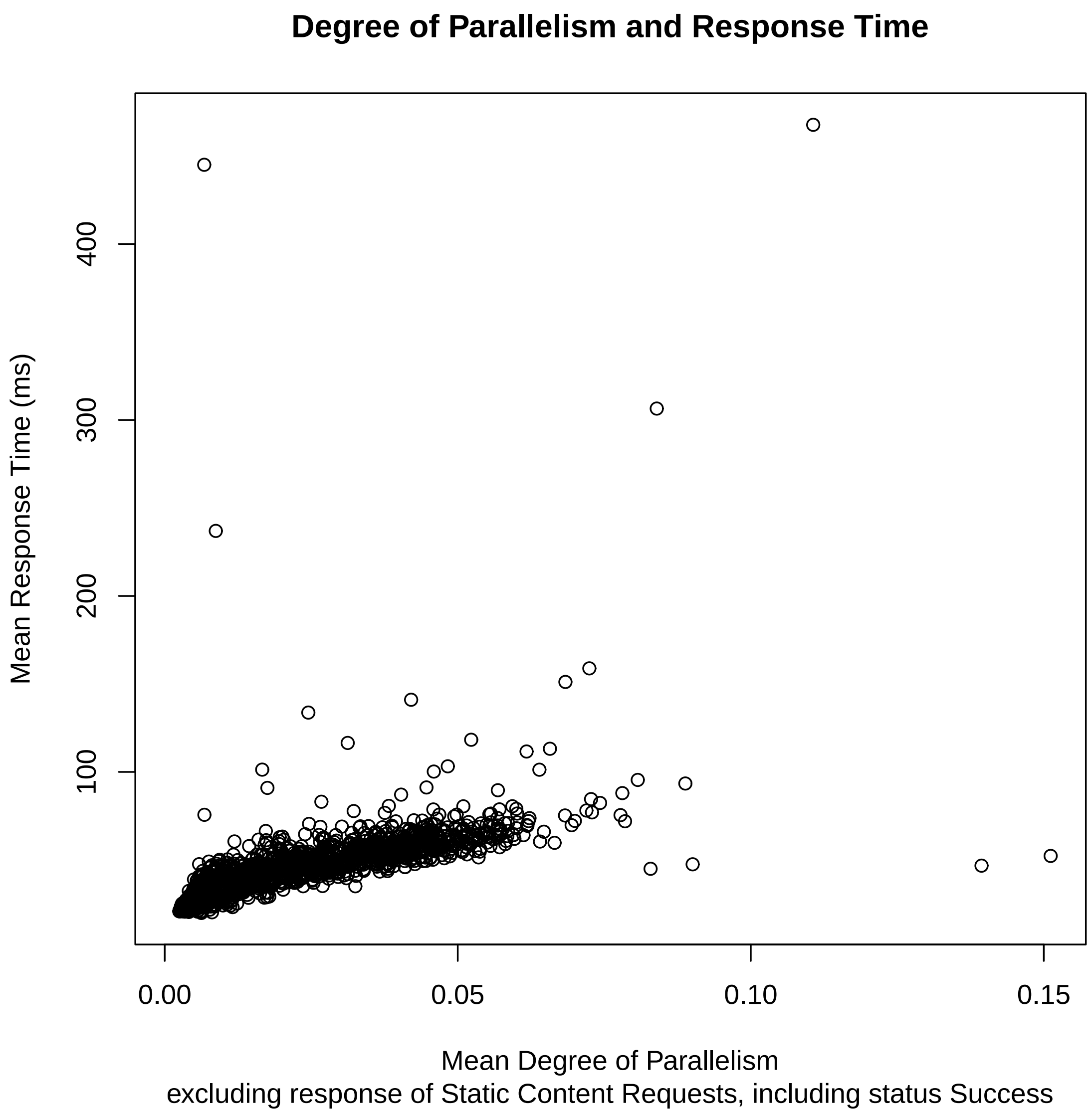


This transaction appears to have distinct response times at about 0.5 seconds, 3.5 seconds, 8.5 seconds and around 17 seconds. It is quite possible that developers are not explicitly aware of the path with the 17 second response time.

- How does response time respond to transaction concurrency?

  High contention, high concurrency environments will perform poorly, but before a system reaches that point it is possible to identify problems with transaction concurrency by identifying the way that response times rise as transaction concurrency rises. In an environment with low contention response time should remain flat as the transaction rate rises until the system starts to see rising concurrency, at which point response time will start to rise faster and faster.

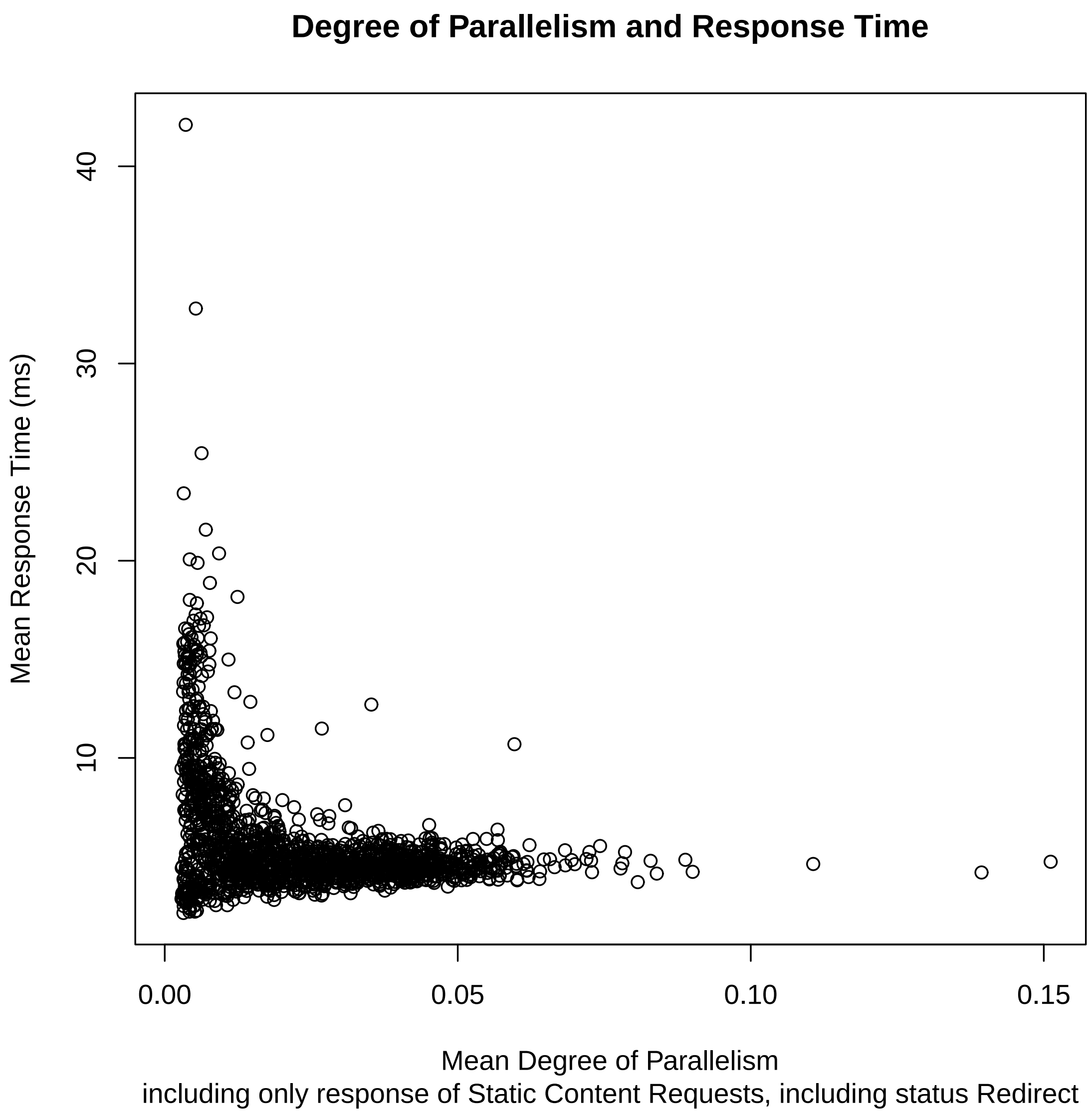
  This example is from a WebAnaytics report for an application with some serious contention issues due to very inefficient SQL. Even at a very low level of concurrency and very good response time, the transaction time is visibly rising with rising concurrency. Underload the system experienced severe problems as a result of that SQL combined with an unexpectedly high transactiohn rfate from another system.



- How does response time respond to network load?

  As with transaction concurrency above, network contention shows up in the same way. This is less likely these days since network bandwidths tend to be relatively large, but the potential for contention is still there. Comparing data rate with the response time of static content requests (which require very little CPU or IO) can indicate problems.

  The graph below is the WebAnalytics report graph of static response time compared with network load. The time does not seem to increase significantly as network data rate rises. It is not unusual for very low request rate or concurrency level times to be highly variable. Long delays between requests allow cached data to be aged out or connections deallocated, requiring more time for reinitialising the network connectivity.



- Poor error handling can look like poor performance.

  Catch blocks that silently swallow errors, or missing catch blocks in some environments (IIS applications are a recurring problem) can result in delays being introduced or in unexpected behaviours. Effective, well structured error handling and error logging is essential and it is necessary to review the code for problems in this area and to fix them early in the remediation process.

- Variable response or throughput is a side-effect of resource contention.

  A system that is under stress will display more variability in response times than an un-stressed one will. Whether outliers in response times are interesting depends on whether they are occurring in a system with known capacity contention or whether there is no known reason for them.

Adding concurrency is sometimes attractive, but is often not an answer, optimise SQL, optimise or remove application processing, and model the workload before considering parallelism.

- In a system that is limited by CPU capacity, adding parallelism, or too high a level of parallelism will make the system perform more badly and will add complexity. In one case some time ago a team was using 24 threads in a 4 core database system, reducing the parallelism to degree 4 helped considerably in the short term, but rewriting the code to make the database access more efficient was the longer term solution.
- In an IO bound system, adding parallelism can help provided that the IO subsystem can sustain the required combination of data rate and degree of parallelism, eventually the system becomes CPU bound or you run out of IO data rate

Too high a level of parallelism in either case will damage performance.

## 4.3 Prioritising the Fixes

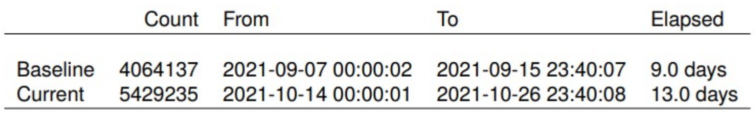
The objective will always be to demonstrate some improvement quickly, so prioritising to get the best payoff with the least effort is critical. The list below of things to focus on is in order of importance.

- Good error handling is non-negotiable and should be given a high priority in the parts of the system that have problems. Its not uncommon to find that there are unidentified errors occurring in a system that is under a lot of stress. This may look like an odd thing to put at the top of the list, but incorrect error handling may be a source of poorly described or currently unrecognised problems. Finding and clarifying user problems early in the process is vital to getting resolution of the problems as quickly as possible.
- Pick a number of transactions with both a high response time and a high total elapsed time to fix first. Pick a number of them because not everything that is slow can be speeded up quickly and you will need some quick results. Getting the most elapsed time out of the system will be important if the system is heavily loaded in some area (database IO is a common example). That will stabilise the performance of other transactions in the system. Iterate over the set of high elapsed/high total impact URLs, picking candidates and fixing what you can until you are dealing with relatively rare transaction problems.
- Pick the least scalable transactions. The WebAnalytics package's report template includes some concurrency/scalability graphs for the overall system but does not graph concurrency for all transactions because the calculation is quite slow. It is a simple matter to either filter the input data (quickest) or to add the concurrency graph to the URL report section (likely to be very, very slow). Where there are transactions (or jobs) that are processing variable size batches of data, and you have some variability in the batch sizes, comparing time cost per record with record count provides a view of how performance responds to batch size. A properly constructed system, including database, will show a more or less constant cost per record over a very wide range of record counts. Scalability optimisation was carried out on a major government batch processing system over a period and by the end almost all of the processing, except for processes that accounted for only a few minutes per day, were operating with an approximately constant per-record cost and a reduction in processing time of about 80%, a reduction in daily processing time from 20 hours to 4.
- Tackle the transactions that you know are hard or risky to optimise last.

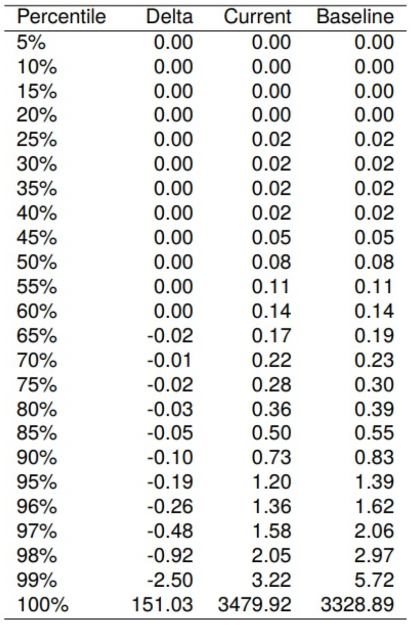
# Chapter 5 Improve Performance

What the code or infrastructure changes are is not the focus of this paper, what they are depends on the technology and the analysis. This is about measurement. Have you go to where you need to be?

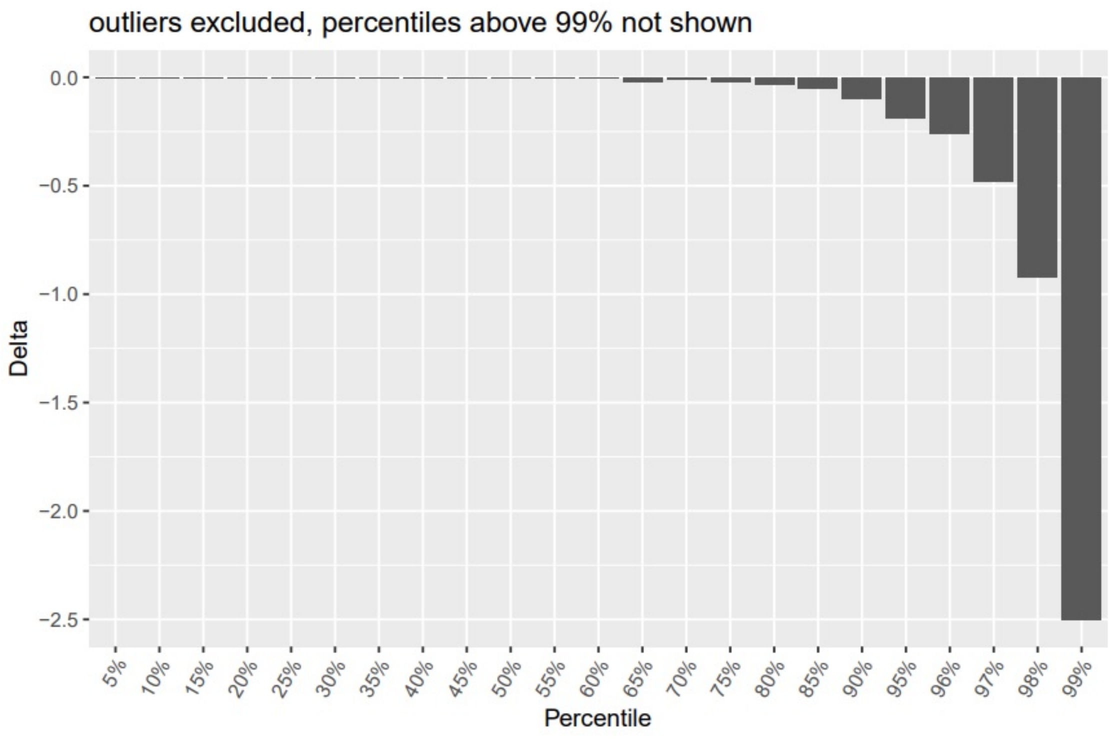
Once the changes have been developed and deployed, comparison of the performance baseline with the revised code will show whether you are getting the improvement you expected from the changes that have been made. If the performance improvement is not what you were expecting, the analysis is likely to be wrong and needs to be revisited, possibly with some new measurement.



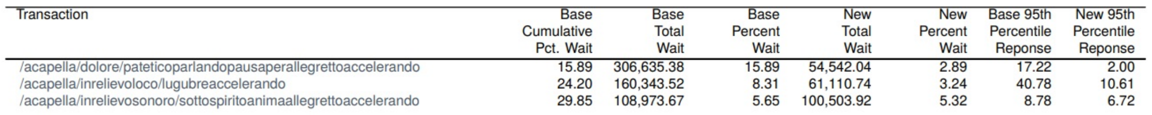
| | Count | From | To | Elapsed |
|---|---|---|---|---|
| Baseline | 4064137 | 2021-09-07 00:00:02 | 2021-09-15 23:40:07 | 9.0 days |
| Current | 5429235 | 2021-10-14 00:00:01 | 2021-10-26 23:40:08 | 13.0 days |



| Percentile | Delta | Current | Baseline |
|---|---|---|---|
| 5% | 0.00 | 0.00 | 0.00 |
| 10% | 0.00 | 0.00 | 0.00 |
| 15% | 0.00 | 0.00 | 0.00 |
| 20% | 0.00 | 0.00 | 0.00 |
| 25% | 0.00 | 0.02 | 0.02 |
| 30% | 0.00 | 0.02 | 0.02 |
| 35% | 0.00 | 0.02 | 0.02 |
| 40% | 0.00 | 0.02 | 0.02 |
| 45% | 0.00 | 0.05 | 0.05 |
| 50% | 0.00 | 0.08 | 0.08 |
| 55% | 0.00 | 0.11 | 0.11 |
| 60% | 0.00 | 0.14 | 0.14 |
| 65% | -0.02 | 0.17 | 0.19 |
| 70% | -0.01 | 0.22 | 0.23 |
| 75% | -0.02 | 0.28 | 0.30 |
| 80% | -0.03 | 0.36 | 0.39 |
| 85% | -0.05 | 0.50 | 0.55 |
| 90% | -0.10 | 0.73 | 0.83 |
| 95% | -0.19 | 1.20 | 1.39 |
| 96% | -0.26 | 1.36 | 1.62 |
| 97% | -0.48 | 1.58 | 2.06 |
| 98% | -0.92 | 2.05 | 2.97 |
| 99% | -2.50 | 3.22 | 5.72 |
| 100% | 151.03 | 3479.92 | 3328.89 |

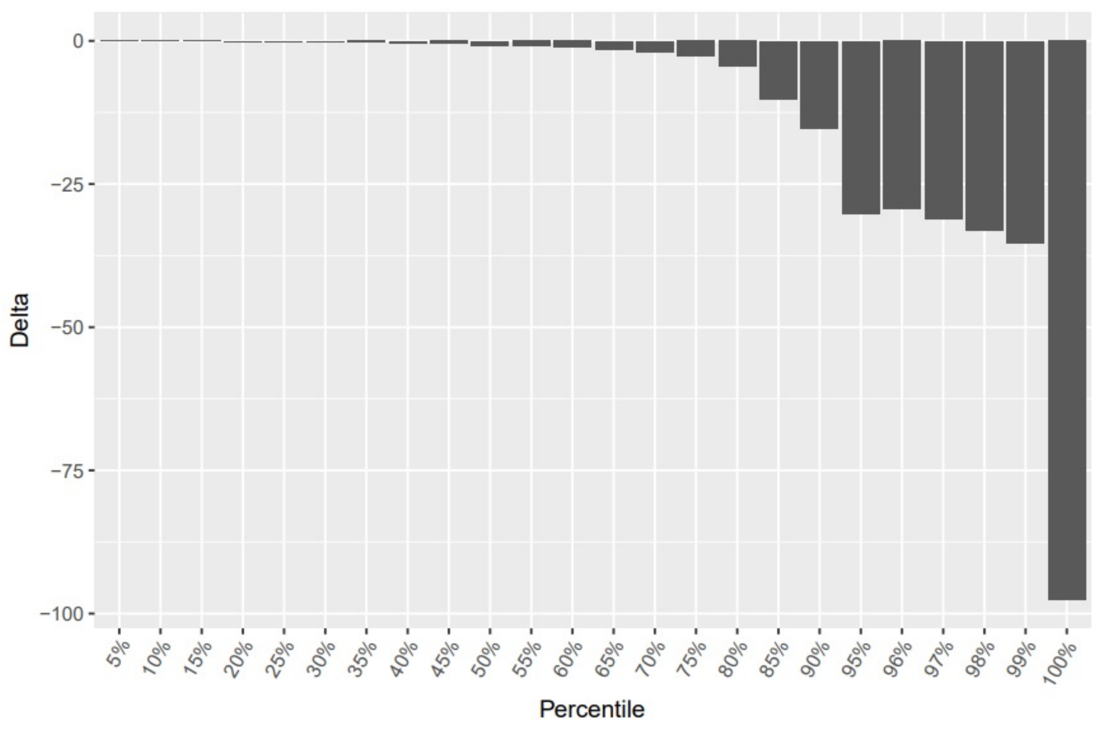


The data above is comparing response times before and after a system release that included some performance optimisation. The total wait comparison table below shows that the highest and second highest total delay transactions have had their 95th percentile times significantly reduced, taking out about 18% of the total processing time in the system.



| Transaction | Base Cumulative Pct. Wait | Base Total Wait | Base Percent Wait | New Total Wait | New Percent Wait | Base 95th Percentile Reponse | New 95th Percentile Reponse |
|---|---|---|---|---|---|---|---|
| /acapella/dolore/pateticoparlandopausaperallegrettoaccelerando | 15.89 | 306,635.38 | 15.89 | 54,542.04 | 2.89 | 17.22 | 2.00 |
| /acapella/inrelievoloco/lugubreaccelerando | 24.20 | 160,343.52 | 8.31 | 61,110.74 | 3.24 | 40.78 | 10.61 |
| /acapella/inrelievosonoro/sottospiritoanimaallegrettoaccelerando | 29.85 | 108,973.67 | 5.65 | 100,503.92 | 5.32 | 8.78 | 6.72 |

The URLs in that table, as they are in all tables, are hyperlinks to the detail page for that URL where the result of the optimisation can be seen, reducing the 100th percentile time by nearly 100 seconds. Comparisons with baseline times are generated for all URLs



At this point the objectives and pain sources should be reviewed. Priorities may change.

For a pure optimisation process, not one aiming to hit a numeric performance target but aiming to provide overall improvement, how do you know that you are finished? The saying that "better is the enemy of good" is particularly true of performance optimisation, but it needs to be modified slightly to: "better is the enemy of finishing". If the problem is one of reducing end-user pain then there will always be more that can be done, there will always be some transaction displaying high response times or large variability. The model suggested earlier, an intense acute problem solving phase followed by an on-going low intensity remediation program is useful and the remediation can be folded into the regular work of a development team. For more straight-forward contractual requirements meeting an agreed threshold is all that is required.

# Chapter 6
# Control the System Performance

This part is unlike the earlier parts, it is an on-going activity. It is also about avoiding doing major problem solving by solving small problems.

System performance changes over time as data sizes change, as business processes evolve and as function is added or changed. Users are often reluctant to report the merely annoying, or the inconsistent as a defect. A statement like "sometimes this transaction is slow" is not likely to be accepted as a valid defect in the same way that an intermittent fault is likely to be responded to by the developers with something along the lines of "works for me" or "unable to reproduce". For many mature systems, performance problems accumulate and grow slowly over time, either though system change or increasing irritation of the user population and often development teams are unresponsive or slow to respond, in part because they don't know how to respond. This can lead to an accumulation of long standing problems being regarded suddenly as a crisis, much to the surprise of the team. An on-going performance management process, or better, a quality management process, including using tools like WebAnalytics to measure performance, a process that takes problems and addresses them as part of normal development activity, not as defect repair, avoids that type of crisis.

For Application Maintenance Services engagements there is often an undertaking to improve the system as part of on-going maintenance, but usually neither quality nor performance management are carried out, despite being the most obvious things to do to meet that obligation. On-going improvement makes fixes cheaper because implementation of fixes as an emergency drags in additional resources (a direct cost), the repair has to be done at awkward times (likely to be a direct cost) and it impacts other activities (an opportunity cost) and it reduces client trust (another opportunity cost), whereas progressive improvement of this type can be fitted around normal maintenance and development work. It leads to fewer surprises and the relationship with the business owner is better, there is higher trust. Having them say "I don't think we need regular catch ups any more, just let me know if there is a problem we should talk about" or "its the least of my worries these days" indicates a preparedness to let the team manage the system.

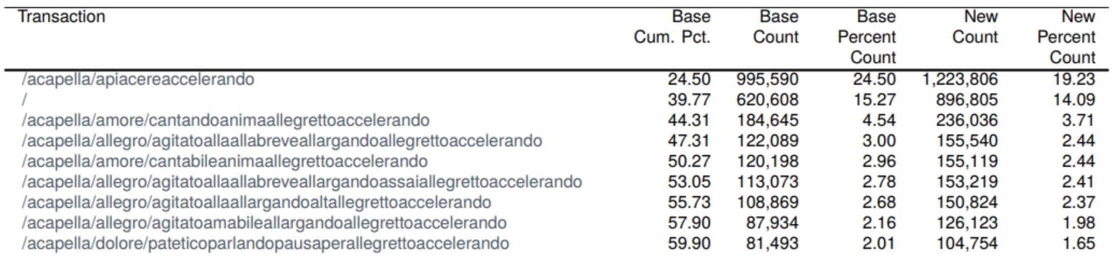
For on-going performance management, the detailed comparisons of frequencies and distributions of response times for before and after changes in the previous chapter are used to monitor changes over time. Regular review of daily scatter plots for transient performance excursions can indicate whether performance issues are present below the level of a major problem. Post-release or load test performance data can be used as a baseline to detect changes over time in performance and the PDF report can be used to document both the initial state and the improvement. If logs are retained, and they are usually only a few MB of tens of MB per day, they can be used to compare performance between specific points in time: "is this Easter's performance different to last year's?" , "is performance during end of month processing degrading over time?", "are all Fridays like this?", questions that are difficult if not impossible to answer with other tools.

# Chapter 7
# Notes on Load Testing

Demonstrating performance using a load test is a useful thing to do, but it is likely to produce different, faster, results compared with a production environment. Load testing is part of performance management. Performance metrics can be collected from development and functional test environments (even if they are lower capacity than production) and used to identify potential performance problems. A transaction running by itself, even in a low capacity environment, should perform well. If it doesn't, it can't magically perform well in a production environment with some amount of contention making its performance worse. Simply collecting logs from development and test environments and feeding them into the WebAnalytics package enables you to identify poorly performing transactions before formal performance test begins. This will hold down the amount of fixing or rework that comes out of performance and load testing and enables simpler load test scripts to be written.

A workload definition will form the basis of a load test for a new system or for a significant release. The key point is to provide a more or less realistic combination of transactions in the system while balancing that against the complexity of the scripting.

Any existing load test should be reviewed, and for simple systems the possibility of building a simple load test should be considered. The workload will need revision to ensure that it matches current production behaviour. The WebAnalytics workload table can be used to compare request frequencies by URL in production (the baseline) and test (the current data) workload.



| Transaction | Base Cum. Pct. | Base Count | Base Percent Count | New Count | New Percent Count |
|---|---|---|---|---|---|
| /acapella/apiacereaccelerando | 24.50 | 995,590 | 24.50 | 1,223,806 | 19.23 |
| / | 39.77 | 620,608 | 15.27 | 896,805 | 14.09 |
| /acapella/amore/cantandoanimaallegrettoaccelerando | 44.31 | 184,645 | 4.54 | 236,036 | 3.71 |
| /acapella/allegro/agitatoallaallabreveallargandoallegrettoaccelerando | 47.31 | 122,089 | 3.00 | 155,540 | 2.44 |
| /acapella/amore/cantabileanimaallegrettoaccelerando | 50.27 | 120,198 | 2.96 | 155,119 | 2.44 |
| /acapella/allegro/agitatoallaallabreveallargandoassaiallegrettoaccelerando | 53.05 | 113,073 | 2.78 | 153,219 | 2.41 |
| /acapella/allegro/agitatoallaallargandoaltallegrettoaccelerando | 55.73 | 108,869 | 2.68 | 150,824 | 2.37 |
| /acapella/allegro/agitatoamabileallargandoallegrettoaccelerando | 57.90 | 87,934 | 2.16 | 126,123 | 1.98 |
| /acapella/dolore/pateticoparlandopausaperallegrettoaccelerando | 59.90 | 81,493 | 2.01 | 104,754 | 1.65 |

## 7.1 Define the Test Workload

The next part of the problem, and this applies to both load test and production, is what is the workload that the response times should be measured against. This is usually easy for a production system, you already know what the workload is. For a new system it may not be possible to know precisely what the production workload will look like, the transaction mix will be uncertain, the number of concurrent users will be unclear, the daily peaks will be unclear. Often the workload is specified in terms of business processes or throughput per day, week, month or even year and it is necessary to estimate the transaction rate. This is usually more tractable than people think. Transaction concurrency (leaving aside a DDOS) is limited by the number of staff and the number of potential users in the general population, for any business process these are knowable numbers - for freight, how many parcels are in flight and how often do people check?

For courts, how many court cases are there? For a retail website, how many customers does the company have for its current business? Completely open-ended user populations and transaction rates are very unlikely, and if you genuinely have that then the requirement is to provide linear scalability (addressed below) and a high degree of elasticity in the infrastructure.

Typical office-hours business transaction patterns have morning and afternoon peak transaction rates that represent a ratio of between two and about five over the daily average. Whether that is true for any one application depends on many factors, but these ratios work well enough as a starting point. Longer term cycles in workload: weekly (Fridays and Mondays), monthly, quarterly, annual, biennial sales, annual license renewals, Christmas, Easter, and Idul Fitri holidays, and the like can be factored in to derive a reasonable daily workload. Government services websites and retail websites have peaks that correspond with breaks in the working day and the early evening with usage tapering off into the night. Given all that however, peak transaction rates (not necessarily concurrency levels) will be much higher over short intervals and the target workload will be a small multiple of the short term projected peak transaction rate that has been worked out from all of these factors. Its not terribly precise, but it does provide a reasonable ball-park workload.

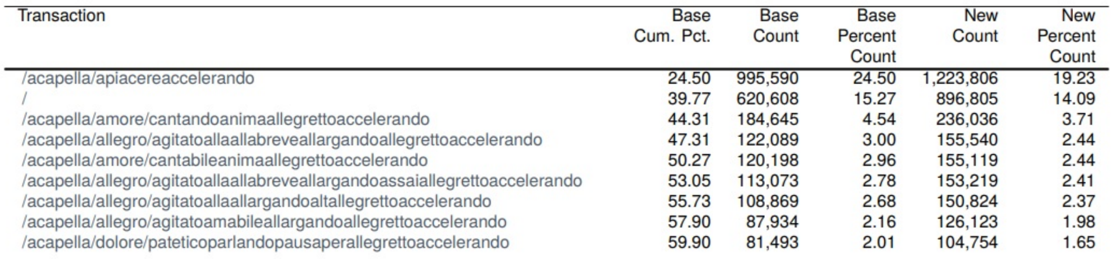
Estimation of think time is always somewhat fraught, the think time metric determines the throughout of the total business system, staff number and computer system response time and think time are the three things that determine how much work can be processed in a period of time. This is typically two to five seconds for business transaction processing systems. Measurement of this from production data depends on the existence of a user or session identifier.

### 7.1.1 Testing Transaction Combinations

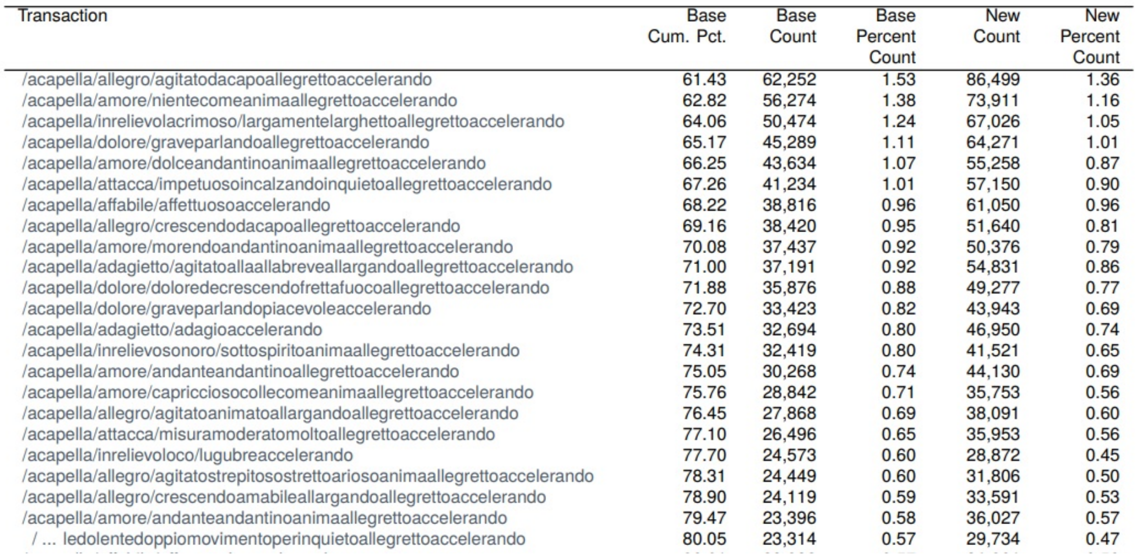
The objective of a load test should not just be to push some number of transactions through the system in an interval, but also to explore what happens when combinations of transactions run together to discover locking or other types of contention. This is very difficult to test manually because there may be very specific sequences and overlaps that trigger problems. There needs to be some degree of intentional randomness introduced into the test to avoid transactions falling into cyclical patterns. This can be introduced using a random delay timer (preferably a Poisson delay representing the think time with a large lambda value) which may be combined with random selection of different process scenarios in the script. A load test with some probabilistic behaviour has a better chance of finding these overlaps than manual testing will.

Don’t try to test everything, load test scripting is expensive and maintaining it even more so. A very complex, very comprehensive load test script will not be maintained and the effort will be wasted when something smaller and less comprehensive, but focussed on the high frequency parts of the workload will be as effective and be more able to be reused in the future. Many transactions or code paths through transactions are used quite rarely, and provided that they perform reasonably well in development (that is, they respond in under the target response time when run alone) then they will be tolerable in production even if their performance is actually not very good. The objective must be to exercise the most common business scenarios and the most common transactions and ensure that those perform well. Identifying and fixing things that are used rarely should be a lower priority than the core business process. Keep in mind that developers may not be able to identify all of the code paths that they have constructed, and its not uncommon for the known code paths through a transaction to perform well but for there to be a poorly performing one that emerges under load.

The WebAnalytics package includes a report of URL frequencies and cumulative percentages that can be used to identify the key transactions to be included in the load testing of a production system or to adjust a load test to correspond with the real world load. The cumulative percentages help in identifying how much of the workload is covered and enables exclusion of very low frequency transactions. Typically, even for very complex systems with hundreds of URLs, a small number of URLS (likely to be a few tens) account for 80% to 90% of the requests.



| Transaction | Base Cum. Pct. | Base Count | Base Percent Count | New Count | New Percent Count |
|---|---|---|---|---|---|
| /acapella/apiacereaccelerando | 24.50 | 995,590 | 24.50 | 1,223,806 | 19.23 |
| / | 39.77 | 620,608 | 15.27 | 896,805 | 14.09 |
| /acapella/amore/cantandoanimaallegrettoaccelerando | 44.31 | 184,645 | 4.54 | 236,036 | 3.71 |
| /acapella/allegro/agitatoallaallabreveallargandoallegrettoaccelerando | 47.31 | 122,089 | 3.00 | 155,540 | 2.44 |
| /acapella/amore/cantabileanimaallegrettoaccelerando | 50.27 | 120,198 | 2.96 | 155,119 | 2.44 |
| /acapella/allegro/agitatoallaallabreveallargandoassaiallegrettoaccelerando | 53.05 | 113,073 | 2.78 | 153,219 | 2.41 |
| /acapella/allegro/agitatoallaallargandoaltallegrettoaccelerando | 55.73 | 108,869 | 2.68 | 150,824 | 2.37 |
| /acapella/allegro/agitatoamabileallargandoallegrettoaccelerando | 57.90 | 87,934 | 2.16 | 126,123 | 1.98 |
| /acapella/dolore/pateticoparlandopausaperallegrettoaccelerando | 59.90 | 81,493 | 2.01 | 104,754 | 1.65 |



| Transaction | Base Cum. Pct. | Base Count | Base Percent Count | New Count | New Percent Count |
|---|---|---|---|---|---|
| /acapella/allegro/agitatodacapoallegrettoaccelerando | 61.43 | 62,252 | 1.53 | 86,499 | 1.36 |
| /acapella/amore/nientecomeanimaallegrettoaccelerando | 62.82 | 56,274 | 1.38 | 73,911 | 1.16 |
| /acapella/inrelievolacrimoso/largamentelarghettoallegrettoaccelerando | 64.06 | 50,474 | 1.24 | 67,026 | 1.05 |
| /acapella/dolore/graveparlandoallegrettoaccelerando | 65.17 | 45,289 | 1.11 | 64,271 | 1.01 |
| /acapella/amore/dolceandantinoanimaallegrettoaccelerando | 66.25 | 43,634 | 1.07 | 55,258 | 0.87 |
| /acapella/attacca/impetuosoincalzandoinquietoallegrettoaccelerando | 67.26 | 41,234 | 1.01 | 57,150 | 0.90 |
| /acapella/affabile/affettuosoaccelerando | 68.22 | 38,816 | 0.96 | 61,050 | 0.96 |
| /acapella/allegro/crescendodacapoallegrettoaccelerando | 69.16 | 38,420 | 0.95 | 51,640 | 0.81 |
| /acapella/amore/morendoandantinoanimaallegrettoaccelerando | 70.08 | 37,437 | 0.92 | 50,376 | 0.79 |
| /acapella/adagietto/agitatoallaallabreveallargandoallegrettoaccelerando | 71.00 | 37,191 | 0.92 | 54,831 | 0.86 |
| /acapella/dolore/doloredecrescendofrettafuocoallegrettoaccelerando | 71.88 | 35,876 | 0.88 | 49,277 | 0.77 |
| /acapella/dolore/graveparlandopiacevoleaccelerando | 72.70 | 33,423 | 0.82 | 43,943 | 0.69 |
| /acapella/adagietto/adagioaccelerando | 73.51 | 32,694 | 0.80 | 46,950 | 0.74 |
| /acapella/inrelievosonoro/sottospiritoanimaallegrettoaccelerando | 74.31 | 32,419 | 0.80 | 41,521 | 0.65 |
| /acapella/amore/andanteandantinoallegrettoaccelerando | 75.05 | 30,268 | 0.74 | 44,130 | 0.69 |
| /acapella/amore/capricciosocollecomeanimaallegrettoaccelerando | 75.76 | 28,842 | 0.71 | 35,753 | 0.56 |
| /acapella/allegro/agitatoanimatoallargandoallegrettoaccelerando | 76.45 | 27,868 | 0.69 | 38,091 | 0.60 |
| /acapella/attacca/misuramoderatomoltoallegrettoaccelerando | 77.10 | 26,496 | 0.65 | 35,953 | 0.56 |
| /acapella/inrelievoloco/lugubreaccelerando | 77.70 | 24,573 | 0.60 | 28,872 | 0.45 |
| /acapella/allegro/agitatostrepitosostrettoariosoanimaallegrettoaccelerando | 78.31 | 24,449 | 0.60 | 31,806 | 0.50 |
| /acapella/allegro/crescendoamabileallargandoallegrettoaccelerando | 78.90 | 24,119 | 0.59 | 33,591 | 0.53 |
| /acapella/amore/andanteandantinoanimaallegrettoaccelerando | 79.47 | 23,396 | 0.58 | 36,027 | 0.57 |
| / ... ledolentedoppiomovimentoperinquietoallegrettoaccelerando | 80.05 | 23,314 | 0.57 | 29,734 | 0.47 |

The table above is from a fairly large system with nearly 2000 distinct URLs, but 80% coverage is provided by 32 URLs. Scripting will need to exercise more than that, and balancing the scripts so that they provide the right counts and proportions is an iterative process. Keep in mind that a test script covering 2000 URLs is likely to be impossibly expensive, time consuming and a maintenance nightmare for all but the largest commercial package development teams.