Applying Queueing Theory Analysis to Oracle Statspack Data

Henry Poras
ITA Software
henry@itasoftware.com
Last updated: March 2, 2009

(**Note:** Parts of this paper were included in last year's submission to the Hotsos Symposium. As some of that material is necessary in understanding this paper, I thought it made sense to merge everything into a single paper instead of submitting two papers, one just containing new material, and the other a prerequisite. For those of you who have to put up with the redundancy, I apologize – HP)

Part I - Introduction

Queueing Theory and Oracle: Interesting? Scary? Somewhere on my list of things to look at? As is probably the case for a lot of people here, my first exposure to this field came from Cary Millsap's book *Optimizing Oracle Performance Tuning* (Chapter 9)[1]. Intrigued, I followed his advice and read some of the authors he referenced. Ultimately, though, I always felt like I was missing something. The techniques and theory seemed powerful, but how could I apply this to my database backed applications?

What I needed was a system to play with, to see what useful data could be extracted. Finding myself in the middle of development of a new product, one with explicit throughput and response time SLA's, this seemed to be a good opportunity to move forward. My intent is to apply simple analysis to our load test experiments based on simple models in order to begin to get a better understanding of how Queueing Theory can be applied to Oracle applications. It should soon become obvious that I am by no means an expert in this field. Hopefully though, some of these ideas and techniques will become a starting point for others. Please also be aware that this paper is very much a work in progress. Experiments, analysis, and understanding of my tools and models still have a long way to go.

My approach to this problem roughly follows these steps. Some of these are obvious, others important but frequently sidestepped.

1. Why bother doing this? What do I hope to find out?
2. Construct a simple model/framework to expedite my understanding and analysis.
3. What do I expect to happen? Take a guess.
4. Collect and analyze data.
5. Compare results to my guess (steps 4 & 5).

Why bother? What can I get from this? Queueing theory deals with, among other things, throughput and residence time. It would be nice to understand how these metrics vary with load for our system. My three main goals are: quick triage, data validation, extrapolation beyond testing conditions.

[1] Millsap, Cary 2003. *Optimizing Oracle Performance.* Sebastopol, CA: O'Reilly & Associates, Inc

quick triage: Where do we spend most of our time? Is the bottleneck the database? If so, where in the database? What is the response time of our bottleneck? How much of our response time is due to the bottleneck? We should also be able to drill down into our environments to get more detailed resource utilization behavior (i.e. CPU, disk I/O).

Commonly, slow load test results lead to requests to 'fix the database'. I will show examples of how easy it is to determine when the database really is or isn't the bottleneck. This kind of triage has allowed us to focus more quickly on the actual source of issues.

data validation: Are we measuring what we think we are measuring? Do our system characteristics follow well understood behavior? Are we running a good experiment?

For example, we discovered an unplanned sleep time being applied by our load testing tool.

extrapolation: At what load will different resources saturate? When will we need to upgrade our hardware?

Construct a simple model. Start simple. Queueing theory offers some help here. My goal is to use similar pictures and approaches at different levels of granularity. Throughput can mean how many business functions complete per second, how many database transactions complete per second, or how many disk IOs complete per second. The analysis can be similar, just the labels on the pictures (and the measured metrics) will change. Does this simple picture really work? How much do we need to know about the internal workings of a system and how much of it can remain a black box? I hope to find out. This will help us decide which complexities can be eliminated on a first approximation.

Sometimes (often) a simple model can be more informative than a complex one. Too many details can obscure understanding. Someone (I wish I remember whom) once postulated that Kepler might not have developed his First Law (the orbits of planets are ellipses, with the Sun at one focus of the ellipse.) if Brahe's data had been more accurate and included all the different planetary perturbations.

Too many details can obscure information.

Take a guess (but in the spirit of BAAG[2]) This is one of the most important and most commonly overlooked steps. If you don't make a guess, how do you begin to understand your results? How do you know whether or not to believe the numbers? If there is a match between your guess and your results, it gives you more confidence in your understanding. If they don't match, either your gut is wrong, or your experiment is.

I first realized the importance of guessing when I was teaching a physics lab using spark tapes to measure acceleration. There was a weight which would spark about 10 times a second and each spark would leave a mark on a strip of purple tape. This weight was suspended about 2 meters off the ground and was then released. This left a big blob of marks at the top of the tape, with a few wobbly dots before good data started. One question on the lab was to determine how much time it took for the weight to fall from the initial blob to the first good data point. Well, it takes less than 1 second for the weight

[2] http://www.battleagainstanyguess.com/

to fall to the floor. The students saw the weight fall, they knew it was quick. Yet when it comes to solving problems, what do you do? Plug in to the equation, of course. The answer has nothing to do with reality. Easily one third of the class had answers of ~ 30 seconds. It would take less than 9 seconds for a weight to fall from the top of the Empire State Building (ignoring resistance, etc.). It would fall almost 3 miles in 30 seconds. A guess would have caught this mistake. Guess ½ a second and calculate 30 seconds, something is wrong. Is it my calculation or my understanding?

One way of forcing you to do this is to include the guess in your documentation. Following is a snippet of a high availability document of mine which does this.

> bonded NIC cards
> bring down one NIC card in the bond using */sbin/ifconfig down eth2 (or /sbin/ifdown eth2)*
> bring down both NIC cards in the bond using */sbin/ifconfig down bond1 (or /sbin/ifdown bond1)*
>
> | | |
> |---|---|
> | **Expectation**: | lose one NIC, system will continue to function normally. Instances and database are unaffected.
lose both NIC cards and node becomes unreachable. VIP should migrate. Check crs_stat. What happens on re-enabling NICs? Does VIP migrate back? Check timings of migration as best as possible (rerun crs_stat and generic connect) |
> | **Comment**: | The Virtual IP (VIP) checks its status by pinging the node's default gateway. On ping failure the VIP will migrate to another node. If a client attempts to connect using this migrated VIP, the connection will be unsuccessful. An error, however, will immediately be relayed to the client who can then attempt to connect via another VIP. This precludes waiting for a TCP timeout which is what would have occurred if the VIP had not migrated. |
> | **Result**: | When the bonded NICs are brought down in node1, the VIP migrates to node 2 in less than a minute. Listener1, ASM1 and instance1 go OFFLINE. After starting up the NICs, the VIP never migrates back. This must be done manually. |
> | **Observations:** | If the network connectivity to a node fails, the VIP migrates as expected. However, when connectivity is reestablished, it does not migrate back automatically. The VIP seems to migrate back to its source only on a reboot of the server. |

Collect and analyze data. My goal is to keep my experiments and analysis as simple as possible. I plan to get system throughput and residence time data from our stress test tool, Oracle throughput and timing data from statspack and awr snapshots, and OS data from the Linux servers using collectl (http://collectl.sourceforge.net). Unfortunately, I have not yet been able to consistently collect and analyze collectl data.

I started with a very simple picture and did some quick sanity check analysis. Models should start simple and grow in complexity only when necessary. If some of my assumptions and pictures seem overly simple, that is by design. It is difficult to fully understand a complex system. Determining which details can be ignored is important. If you are interested in playing with simple models (nothing database related at all), I highly recommend reading 'Consider a Spherical Cow'[3].

> "… it is preferable … to develop relatively simple, analytically tractable models, rather than complex ones requiring truckloads of parameters. The advantage of being able to 'tinker' mentally with a simple, penetrable model, and thus explore the consequences of a variety of

[3] Harte, J. 1985. *Consider a Spherical Cow.* Los Altos, CA.: William Kaufmann

> assumptions, outweighs in most cases the greater realism that might be attained with a complex model.
>
> "Thus the 'spherical cow' in the title of this book. The phrase comes from a joke about theoreticians I first heard as a graduate student. Milk production at a dairy farm was low so the farmer wrote to the local university, asking help from academia. A multidisciplinary team of professors was assembled, headed by a theoretical physicist, and two weeks of intensive on-site investigation took place. The scholars then returned to the university, notebooks crammed with data, where the task of writing the report was left to the team leader. Shortly thereafter the farmer received the write-up, and opened it to read on the first line: 'Consider a spherical cow….
>
> "The spherical cow approach to problem solving involves the stripping away of unnecessary detail, so that only essentials remain…The trick is to know which details can be stripped away without changing the essentials."

That is where I hope to get, (with help).

So let's get going.

Motivation

At ITASoftware we are currently stress/load testing a new airline reservation application. We have a homegrown load testing tool, Pounder, which sends requests to one or more application servers with a desired mix of business functions. At the end of a set time or set workload, our Pounder tool calculates the average throughput and response time of the run. This is a good start for understanding our system, but I also want to focus on bottlenecks, and dig in to the database layer. Throughput and Residence Time values can be obtained by running our application on production hardware with the anticipated mix of functionality, but this struck me as an experiment which would return a very limited amount of data, one which would be difficult to extend to new mixes, and new hardware. Since this is really just a single data point, it also becomes difficult to validate the experiment. It is tough to make a guess. Are we measuring what we think we are measuring? These limitations (along with just being curious) is what led me down the path of queueing theory. My plan is to

- break down the application mix into individual pieces of functionality before repacking them
- guess anticipated behavior (many of these coming from my readings of Neil Gunther[4]) in order to validate our experiments
- determine resource utilization (i.e. CPU, disk i/o) and bottlenecks if possible. This would also make it easier to predict results on different hardware (if CPU changes, and it is/isn't the bottleneck, then …)
- analyze the standard mix as most of our tests and examples will be run under these conditions. This raises the question of what we can tweeze from a heterogeneous workload.

Our situation

My setup consists of the following
<u>logical servers</u>

- database server

[4] Gunther, N.J. 2005. *Analyzing ComputerSystem Performance with Perl::PDQ.* Berlin: Springer-Verlag.

- application server (multiple application servers can be started on a single piece of hardware. For the purposes of this test, each application server represents a single user, an agent.)
- pounder (load generating application – home grown)

statspack/awr snapshots are taken at the beginning and end of each pounder run (and sometimes in the middle.)

collectl will be run on db server, app server, pounder server in future tests

pounder can be configured to run

- for a set period of time
- for a set workload
- for a mix of business functions
- using a modifiable sleep time

Most of the analyzed runs consisted of a standard, complex mix of functions. These were taken by the load test team and were run with up to 128 agents, on a 1-3 node RAC cluster. I also ran some simpler tests using two fundamental business functions for my test: flight information, inventory request. I chose these for two reasons. First, they both consist of only a single command. Secondly, one is read intensive and the other is write intensive.

Part II - 10 minute introduction to Queueing Theory (my informal interpretation)

This presentation is intended to be a first step to applying queueing theory to experimental results obtained from database related stress tests. Since simplicity promotes understanding, my goal is to keep the model, experiment, and analysis as simple as possible and still obtain useful results. (I am trying to avoid 'I have three disk arrays, some RAID1+0, others RAID5. They have different workloads and some make more visits to CPU than others. I also have different priority processes hitting my 8 core. Of course this is only on the database box. My app server makes numerous calls, some to the db, others to other servers. Some of these are sync and others async. …' I DON'T WANT TO MODEL THIS. I DON'T EVENT WANT TO UNDERSTAND ALL OF THIS. THERE IS TOO MUCH GOING ON.)

So let me give a 10 minute review of what I consider to be the basics necessary for my analysis. This is, by necessity, a brief review of the fundamentals of Queueing Theory. It is primarily based on my reading of Neil Gunther, Cary Millsap, and a dollop of Raj Jain[5]. Anything that is correct comes from them, the incorrect stuff is strictly mine.

Definitions

First of all, what is a queue? The site m-w.com states that a queue is "a waiting line, especially of persons or vehicles."

[5] Jain, Raj. 1991. *The Art of Computer Systems Performance Analysis.* New York, N.Y: John Wiley & Sons.

For example, think of a toll booth. People get in line, they wait their turn, they pay and leave. There are constantly new people getting in line, and once someone is finished, they leave and don't get back in line (simple case).

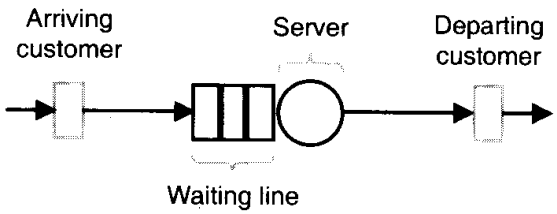

Fig. 1 From Ref. 4.

arrivals (A), arrival rate (λ)

A is the number of people joining the queue, while the arrival rate, λ, measures how quickly they join the line (arrivals per second).

completions (C), completion rate/throughput (χ)

C is the number leaving the queueing center, while the throughput measures how quickly they leave (completions per second).

busy time (B), Utilization (ρ)

These two metrics measure the amount of time for which the service center is busy. B is the total amount of busy time. Utilization is the percent of busy time (B/T).

service time (S)

This measures how long it takes the queueing center (i.e. the toll taker) to do the required job for each customer. This can by thought of as B/C.

queue length (Q)

The Queue is the number of people in line, including the person currently being helped.

residence time (R), waiting time (W)

The residence time is the time spent at the queueing center, the time it takes from joining the line to leaving it. This is made up of Service Time (S) and Wait Time (W).
$R = W + S = QS + S$

service demand (D), number of visits (V)

A request might visit a queueing center multiple times (i.e. one database transaction can have many visits to disk, to CPU). In this case, the service time (S) can also be defined as a service demand, the total time the request spends at the service center. If it makes V visits, $D = VS$.

Load (N)

This metric is a bit vague. Typically I will use it to mean the number of requestors (users, sessions, active sessions, user calls, …). We will return to this later. As Justice Potter Stewart said "I know it when I see it." It seems to me that if you find yourself talking about the load in a queueing center, that is no different from Q.

Conditions

To get useful information from our analysis and experiments we need to make a few assumptions. The first one is that we are in a condition of steady state.

Steady state: Steady state is important as it means the conditions we are measuring are not changing for the period of time in which we are interested. (how steady is steady? sort of relative). If our queue length is constantly increasing (arrival rate > completion rate) then we are not at steady state. So we need $\lambda = \chi$ for our measured time frame in order to assume a steady state condition.

If S and λ are constant, this is trivial. As long as $\lambda < 1/S$, the service center can keep up, and Q=0 (R=S). Once $\lambda > 1/S$, both Q and R eventually go to infinity. For $\lambda > 1/S$, we do not have steady state. In almost all physical cases, λ, S, and χ are not constant, but have a distribution of values. This is why we have finite queues and changing residence times even though the average value of λ is less than the average value of 1/S.

So for steady state conditions we assume:

- $\lambda = \chi$ (1)
- $\lambda < 1/S$ (2)

Memoryless distribution: In the most commonly used and analyzed distributions each arrival is independent of the preceding one. The distribution is memoryless. A Poisson distribution meets these requirements and is commonly found in real life situations (don't worry about understanding all of this. I am just adding it for completeness). Results associated with Poisson distributions can often still be used even if the arrival rate/service time is not strictly memoryless (i.e. feedback loops have some memory)

Relationships

There are a number of relationships by which the previously defined terms can be associated. Some are general, while others are identified with specific models.

General Relationships

Little's Law

The most important general equation is Little's Law. It states

$$Q = \lambda R \qquad (3)$$

Another formulation of this, Little's Microscopic Law, states:

$$\rho = \lambda S \qquad (4)$$

Model Specific Relationships

There are also a number of relationships which are sensitive to the specifics of a model. This raises the question of how well we need to understand our system in order to properly analyze it. This is potentially a serious restriction as many systems are black

boxes. Part of my goal is to determine experimentally how much detail is needed for basic, useful analysis (your definition of those terms can by plugged in anytime you wish). Consider a spherical cow …

open queue

The simplest model to consider (the one we have already considered) is an open queue (e.g. web application, supermarket checkout). Here the arrival rate is always constant. There is no constraint on the number of customers/transactions coming in.

Since $\lambda = \chi$, as we double our arrival rate, we double our throughput. This scaling continues until the server capacity is reached ($\rho = 1$ or $\lambda = 1/S$. Same thing)

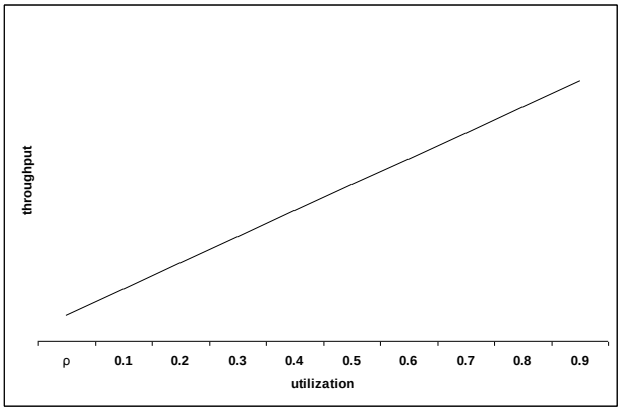

Fig. 2 For open systems, χ scales linearly with ρ until ρ=1. ($\rho = \chi S$)

So for a single Queueing Center, $R = W + S = SQ + S = S(\lambda R) + S$ or

$$R=S/(1 - \rho) \qquad (5)$$

If we have multiple Queueing Centers fed by a single queue (i.e. post office) we now have

$$R=s/(1 - \rho \text{ ^^m}) \qquad (6)$$

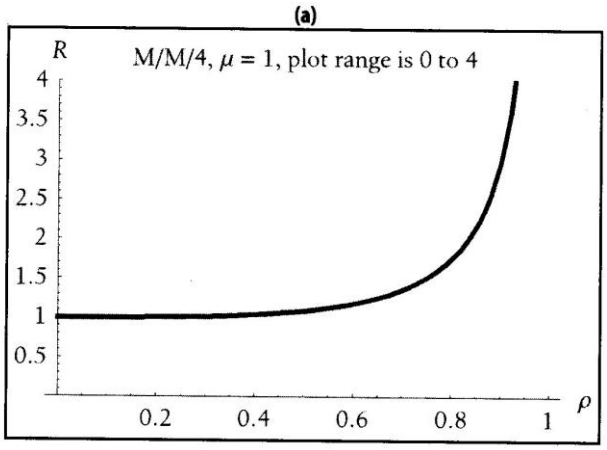

Fig. 3 Plot of R vs ρ (from Ref. 1). Note that R goes to infinity as ρ approaches 1.

closed queue

A closed queue (i.e. load test environment, batch server) has a limited number of customers. The arrival rate changes with time. When the system is at its heaviest load, when all available customers are in a queue, there are no more customers to enter the queue, so λ goes to zero. There is a built-in negative feedback loop.

Our standard metrics of Residence Time and Throughput are now dependent on the number of customers (N), and the think time (Z).

A closed queue is often referred to as a repairman's model. We have N machines, each of which tends to break down after Z days. When it breaks it is sent to the service queue. After being fixed, it again sleeps for an average of Z days. This can also represent a server with N processes, some sleeping, some running, some in run queue. So this model also fits many computer systems as well as most load tests.

Different mathematical expressions are now necessary as throughput no longer scales linearly until utilization equals one, it is limited by N.

The main point to keep in mind with respect to closed queues is that there is now a limit on the maximum queue size and on the maximum residence time. Some useful relationships include:

$$Q_{max.} = N \qquad (7)$$

$$\chi(N) = (N-Q)/Z \qquad (8)$$

$$\chi(N) = N/(R + Z) \qquad (9)$$

$$R(N)=(N/\chi\,(N)) - Z \qquad (10)$$

So now if we look at R or χ vs load, it is plotted as a function of N, not of ρ.

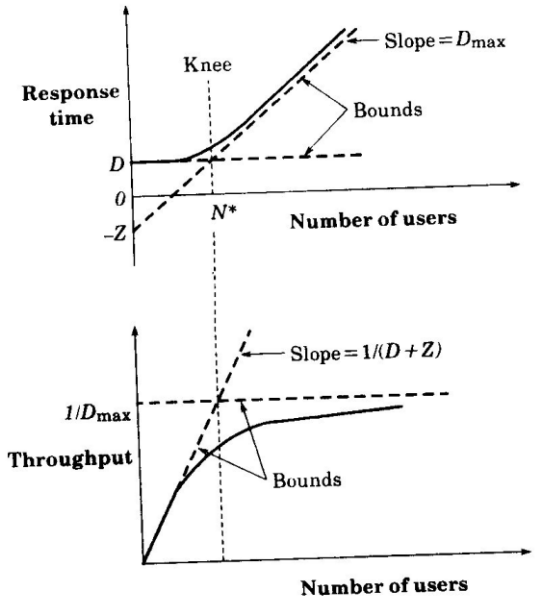

Fig. 4 Closed queue behavior of R and ρ vs N (from Ref. 5).

How does this impact my expectations? my guesses? At work, I have had discussions with people working on load testing who have been concerned that throughput doesn't scale linearly with the number of agents (N). They wanted to know why. The first question is, given our experiment, do we expect it to scale linearly? If Lambda→ 2*Lambda, should X go to 2*X??

feedback

Sometimes you get back in line. A call can request some CPU time. After completion, it may leave the system, or possibly make a disk call, which then returns to the CPU. At this time, the request can either leave the system, or return again to the disk. A single call can make multiple visits to a queueing center.

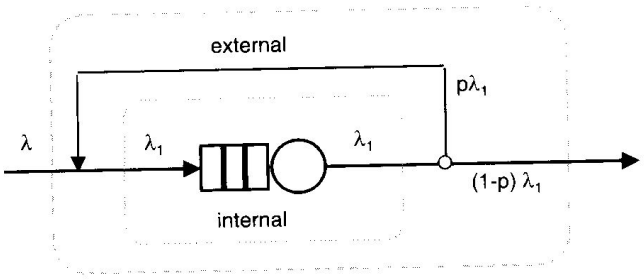

Fig. 5 A feedback circuit. The input throughput we measure might not be that seen by the queueing center. (from Ref. 4)

So one call can make many visits to a queueing center. A single business function can make many roundtrips between the app server and the database. A single db transaction can make many visits to CPU and disk. Because of this, we need to be a bit careful about our definitions of arrival rate and service time. Nothing too difficult, we just need to know what we are measuring.

The arrival rate we observe may not be that seen from the perspective of the queueing center. (see figure 5). We see an arrival rate of λ, while the QC sees λ_1. The residence time can be thought of as

$$R=V_iR_i \qquad (11)$$

and service demand is defined as

$$D=V_iS \qquad (12)$$

where V is the number of visits.

The number of visits can often be measured (disk reads, counts of SQL*Net message to client, …). Notice, however, that our arrival rate is no longer memoryless. There are theorems that still let us analyze QC as though they utilize Poisson distributions.

Additional stuff

There are two more points I wish to briefly mention.

multiple Queueing Centers

Typically we have multiple queueing centers. Unless resources are divided evenly among them (unlikely), one will saturate first ($\rho = 1$). This is the bottleneck. The maximum throughput from our system will then be

$$\rho = 1 = \chi_{bn} D_{bn}$$

or

$$\chi_{bn} = 1/D_{bn}$$

Plotting χ vs Load for our system will show us the service demand of the bottlenecking Queueing Center. This could help us find our bottleneck (see Fig. 4). If the Queueing Centers in our system are all in series, the throughput for all QC is throttled at χ_{bn}. Even though the Queueing Center with the longest Service Demand will be the bottleneck, this does not mean that most of our time is spent waiting for that resource.

multiple workloads

We also often have different workloads hitting the same queueing center. For example, I could run a mix of business functions simultaneously. CPU could be processing OLTP and batch, sorts and disk I/O, Each can have different arrival rates, service times, … In Oracle we are always dealing with mixed workloads in the form of foreground and background processes. They both impact underlying resources, but we don't want to lump together their residence times and throughput behavior. For my initial analysis I won't consider multiple workloads, though this is an important topic into which I would like to devote further attention.

General questions

- how much detail do I need to know about my setup? can it remain a black box?
- can experiment help me decide what is useful and relevant?
- understanding my system can lead to a model which yields predictions. Can this work in reverse? Can experimental data help to deduce a probable model? Sounds ugly.
- in a reservation system each agent must wait for a response from the system before issueing a new call. Does this call and response behavior need to be considered separately, or is it just a standard closed queue? Is a call and response system similar to a closed queue? Throughput is dependent on R, but …??
- can I use the same picture to describe different parts of my system? (Fig. 6)

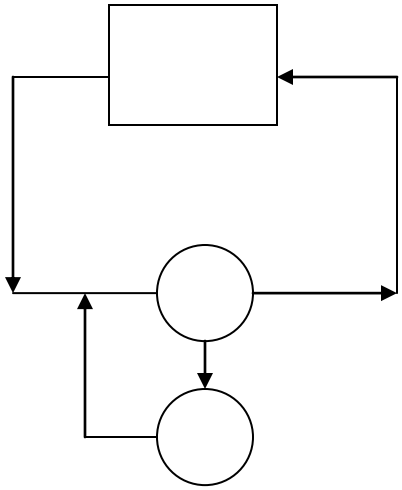

Fig. 6 This generic picture can represent the flow of users from Pounder to the App Server/Database loop, or the flow from the Database to the CPU/Disk loop.

Part III - Connect Theory to Experiment.

This section begins to tie together Queueing Theory with experiment. We now know many of the interesting parameters. How are they determined? What can we measure directly? What needs to be calculated? Many of these relationships are in the following table. OS parameters (i.e. cpu, disk io) will be added to the mix in the future.

Table 1: Simple Statspack Starting Point

- Q=Average # of Active Sessions = (DB Time)/(Elapsed Time)
- χ = txn/sec or (user calls)/sec
- R = sec/txn = (DB Time)/[(txn/sec)(Elapsed Time)] = Q/χ

Table 2: Relationship between Parameters and Measurements

Parameter	Symbol	Measurement (also see Notes following table 2)
Arrivals, Arrival rate	A, λ	Measure χ. Assume $\lambda = \chi$
Completions, Throughput	C, χ	System – data obtained from load tool (tool input parameter sets number of completions. It returns throughput) Database – txn/sec (for OLTP workload) user calls/sec (for SELECT heavy workload) (these come from statspack. Throughput*run time = # of completions)
Busy time	B	DB CPU from statspack (does not include background processes) sysstat 'CPU used by this session'
Service Time	S	S = B/C (direct measure) S = R (for Q = 0) (graphical measure) $S = \rho/\chi$ $S = R/(1 + R\chi)$ (model specific. From equ (5)) These measurements should be comparable.

		S_{bn} This can be determined graphically (see Fig. 4) System – S = R (for 1 agent) S = R/(1 + Rχ) Database – <u>direct method</u> S_{db} =(DB CPU)/(# of business functions completed) This is the service time of the database from the perspective of the system. (this is really service demand, D, as there may be multiple db visits for each business function completion) This can also be calculated in more database centric terms by using # of transactions or # of user calls for C. <u>indirect method</u> S = R (for 1 agent) See measurements of R.
Visits	V	<u>database visits per business function</u> counts of 'SQL*Net message to/from client' [each visit can have many db completions (txn, user calls)] <u>database completions per business function</u> # of txns (or user calls)/# of business functions <u>OS visits from database</u> Disk visits per db txn – PIO/txn (can also have LIO/txn or per user call depending on throughput metric)
Service Demand	D	See Service Time and Visits
Utilization	ρ	$\rho = S\chi$ system – have necessary data database – have necessary data cpu – see vmstat for ρ.
Residence Time	R	system – from load test tool database - (from statspack) DB time/(# of completions)
Wait Time	W	R - S = W
Queue Length	Q	Q = χR Q = N - χZ When the database is the Queueing Center, Q=AAS (Average Number of Active Sessions) = DB Time/Elapsed Time See Appendix B to see more on the Q – AAS connection.
Sleep Time	Z	Determine graphically from R vs. N and χ vs. N plots (see Fig. 4) System – configurable parameter in load test tool. Determine graphically Database – determine graphically.
N_{opt}	Optimal Load	Determine graphically (see Fig. 4)

Load	N	# of agents # of database sessions # of active database sessions (approximately ('DB time')/(clock time)

Notes on Parameters

Completions:	Is χ/agent a meaningful parameter? User throughput a.o.t. system throughput.
Busy time:	see Service Time notes Also, queueing centers other than CPU need different measurements.
Service time:	$S_{appsrv} = S - S_{db}$ Is DB CPU the correct measurement? I first saw the connection between service time and CPU made by Anjo Kolk in his "Yet Another Performance Profiling Method" paper whch states "The service time is equal to the statistic 'CPU used by this session' which is shown through entries in v$sysstat or v$sesstat". Oracle uses CPU to do its work, but is it really the correct measure for service time? Some CPU is associated with waits (i.e. latches). Also some db services aren't purely CPU (i.e. writes to redo logs)
Residence time:	I'm trying 'DB time' as a measure of R. According to the Oracle Database Performance Tuning Guide *"this statistic represents the total time spent in database calls and is an indicator of the total instance workload. It is calculated by aggregating the CPU and wait times of all sessions not waiting on idle wait events (non-idle user sessions)."* So DB time/completions seems like a good first cut at R.
Wait time:	From Optimizing Oracle Performance [Ref 1] p. 242 *"an Oracle wait time is not the W of an R = S + W equation from queueing theory."*
Queue length:	For Z=0, I expect Q~N
Sleep time:	Sleep time of the system isn't the same as sleep time of each component. Pounder can be configured to have agents sleep for some average time between each call. The database, on the other hand, can be sleeping while waiting for the application server to do its thing. Since R of app server is R(N), Z of database can also vary with N.
N_{opt}:	This increases with increasing Z.

Part IV - Experiment and Analysis

My data collection tools are statspack, awr, and pounder. I am not concerned about Statspack just providing averages of my data, as all I am doing is running the same business functions thousands of times. The weakness of averages hiding important data is avoided. (the average of 5,5,5,5,5,5,5,5,5,5 is still 5. That is different from the average of 0,0,0,1,1,1,1,1,0,45 being 5).

During application development numerous load tests have been run. The following examples were chosen to demonstrate the different kinds of analysis techniques we have used. The main goals are to show what can be accomplished from quick analysis as well as from some more detailed examination. The five examples which follow will look at

1. how characteristics differ when the database is/isn't the system bottleneck
2. quick analysis of a tricky problem
3. drilldown into the database using wait events

4. determination of unanticipated system sleep time
5. differences between RAC and non-RAC systems

Is the database the bottleneck?

There have been a bunch of times I have heard "our tests aren't scaling, things are slow. Can you look at the database?". Well, as you all know, there is always something to tune in Oracle, but my first question is to determine whether or not the database is the bottleneck. Using these QT based techniques this becomes easy to do since we see very different characteristics when the database is or isn't the system bottleneck. I'll show an example of each case and then discuss the similarities and differences.

Yes, database is the bottleneck

At the end of each run, our load testing tool, pounder, produces a text file containing the number of agents used in the run, the number of business functions (a.k.a. sequences) completed per second (throughput), and the average response time of an average business function. From Table 1, we know how we can use statspack reports to find Queue length, Throughput, and Residence Time for our database. So now we can calculate Q, χ, and R as a function of increasing load for our system and our database. We expect to see something resembling the curves in Fig. 4.

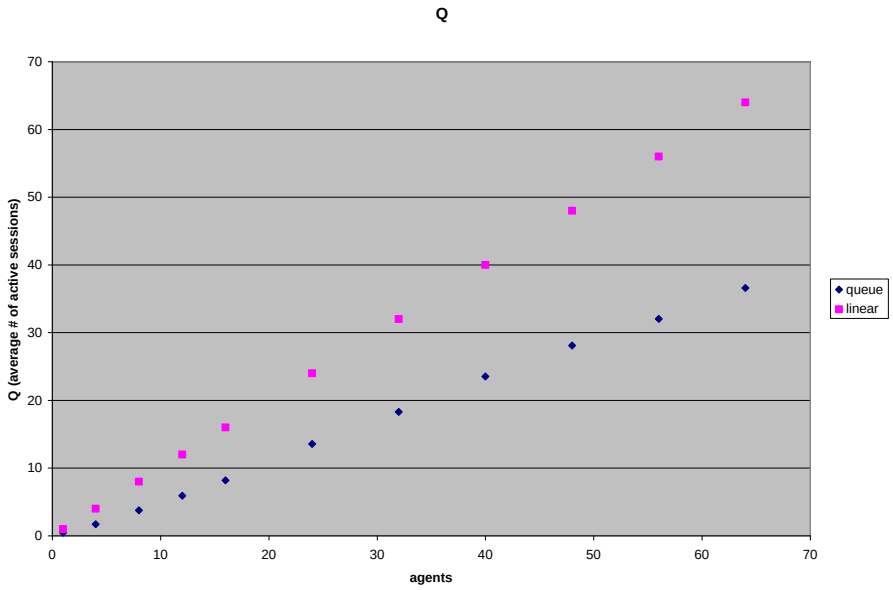

Fig. 7: Queue length within the database.

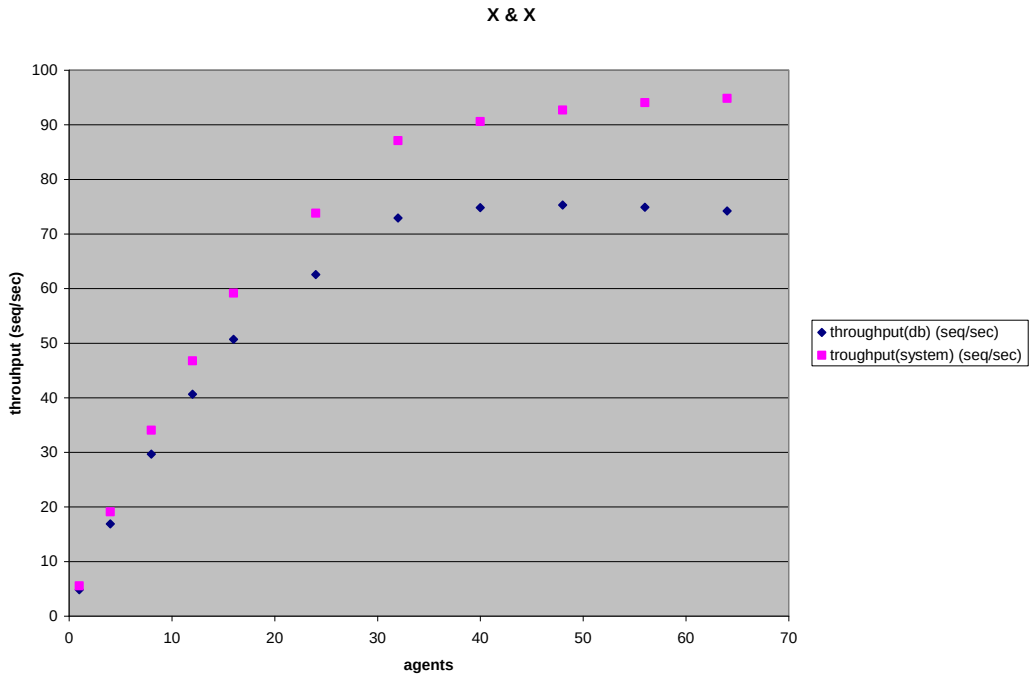

Fig. 8: Database and system throughput in units of sequences/sec.

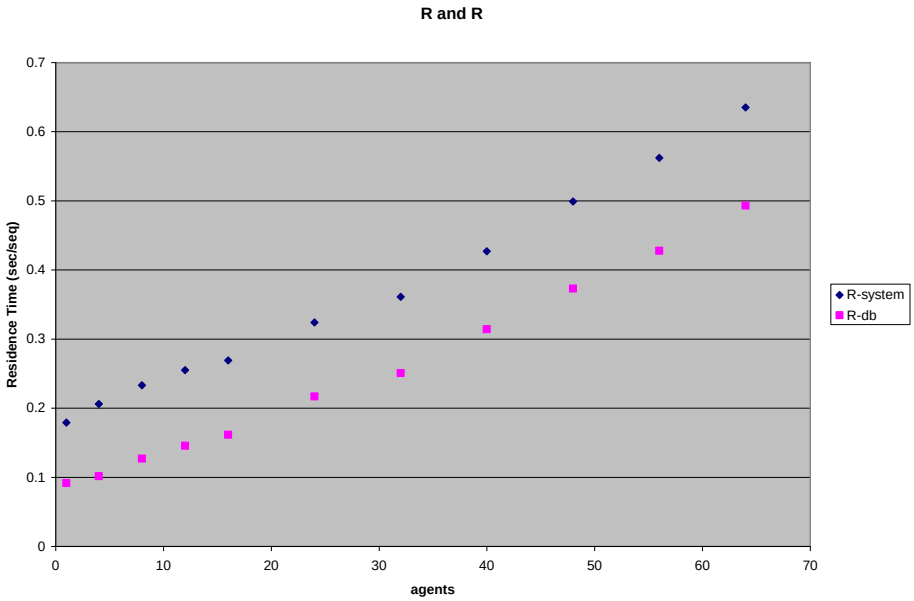

Fig. 9: Response time per business function for the entire system and for the database

Some things to notice about these plots:

- Our lineshapes are about as expected which gives me more confidence that we are measuring the right stuff.
    - Throughput goes through the origin
    - Throughput rises linearly, then saturates
    - Above saturation, Residence Time increases linearly

- Response times (Fig. 9) start increasing almost immediately. This implies queueing (not saturation) starts almost immediately.
- Our database Response time at low loads is ~70 msec/seq and our system Response time at low loads is ~166 msec/seq. These times are close to the Service time since at low loads we expect no additional wait time (R=S). Without any code, configuration, or hardware changes this is as fast as we can go!
- Using the system saturation level measured in Figure 8, we see the Service Demand of the bottleneck is 10.6 msec (see Figure 4 for calculation). This is much lower than the Residence times of either the database or the system. We see that the bottleneck, the queueing center which saturates first, is not necessarily where most time is spent. The bottleneck impacts throughput and impacts response time after saturation. It may have only a minimal impact on response time at lower loads.
- Service Demand of the bottleneck measured from the high load slope of system Residence Time (fig. 9) is 8.6 msec (close to that measured from throughput)
- As the number of agents increases, we see an increasing number of them queueing in the database (50% → 78%). Remember that the queue includes both those waiting and doing work.
- I expected the system and database throughput to match (Fig. 8). The shapes are close, but not identical, and they saturate at different levels. Examining our data revealed a disparity in run times for the pounder and statspack reports. This appears to account for the throughput differences.
- Make sure the same units are used when comparing system and database data. In this case I used (business functions/sec) for throughput and not txn/sec. (of course it was just an easy conversion using the number of transactions in one business function. Pounder gives us the number of business functions in the run, whereas statspack gives us the number of transactions. This conversion factor should and does remain constant across different loads).

No, database is not the bottleneck

I recently got an email at work which said "we are having issues in scaling up [our app server] system. … Could you help us analyze these reports for any DB bottlenecks…" It was already determined that our application server was healthy, with no signs of saturation.

Well, the Pounder based plots definitely showed saturation in the system as can be seen in Figures 10 and 11.

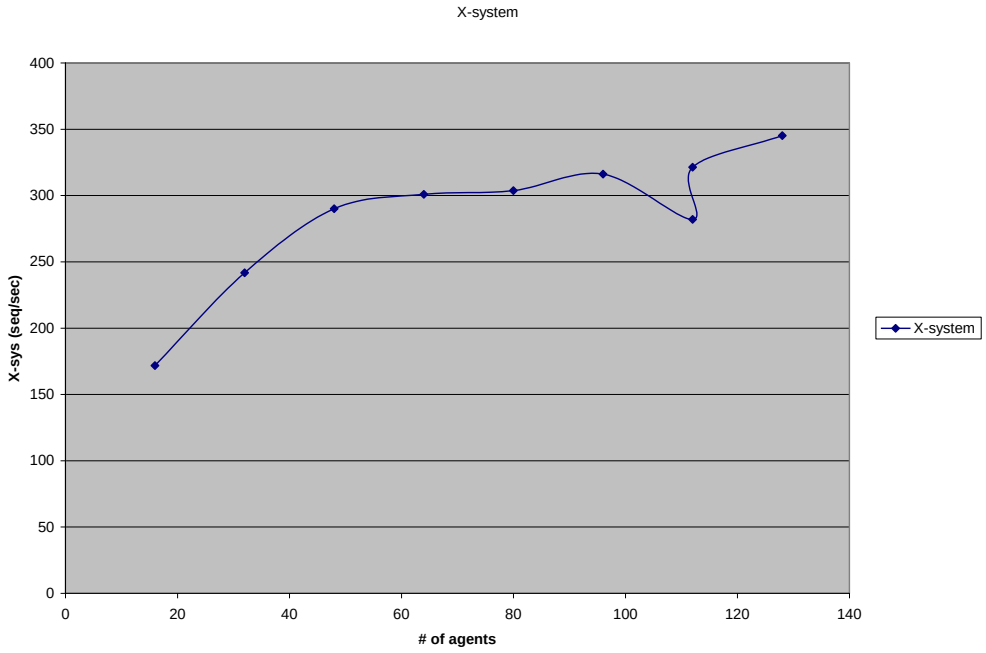

Fig. 10: System throughput

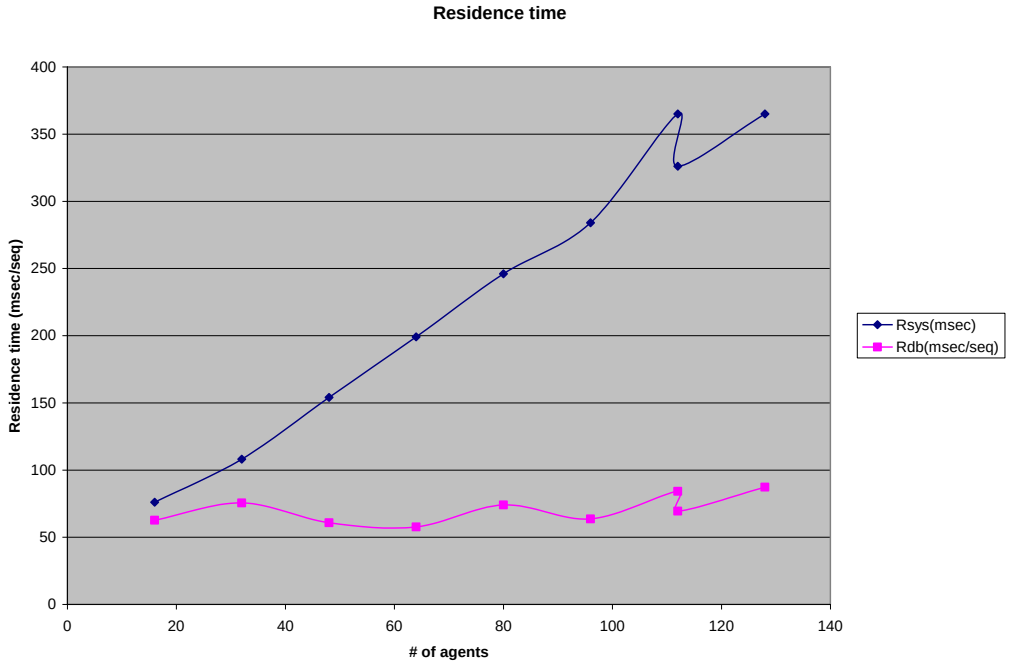

Fig. 11: Response time per business function for the entire system and for the database

We see a throughput saturation at ~40 agents with a constantly rising Residence time. Yup, things aren't scaling and something is a bottleneck. But is it the database?

Let's look at Q, χ, and R for the database.

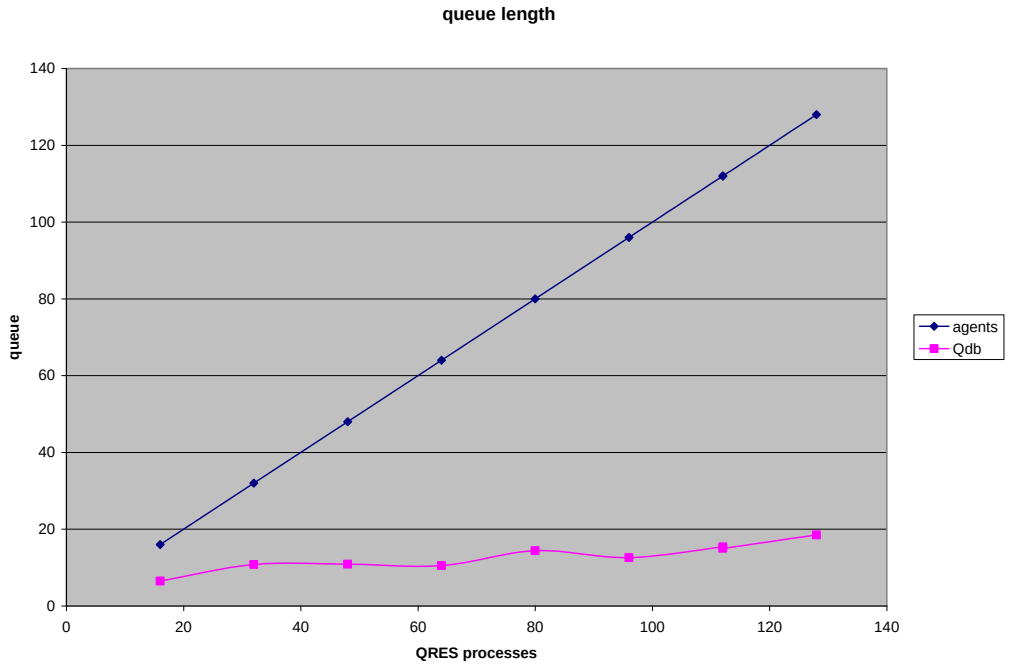

Fig. 12: Queue length within the database.

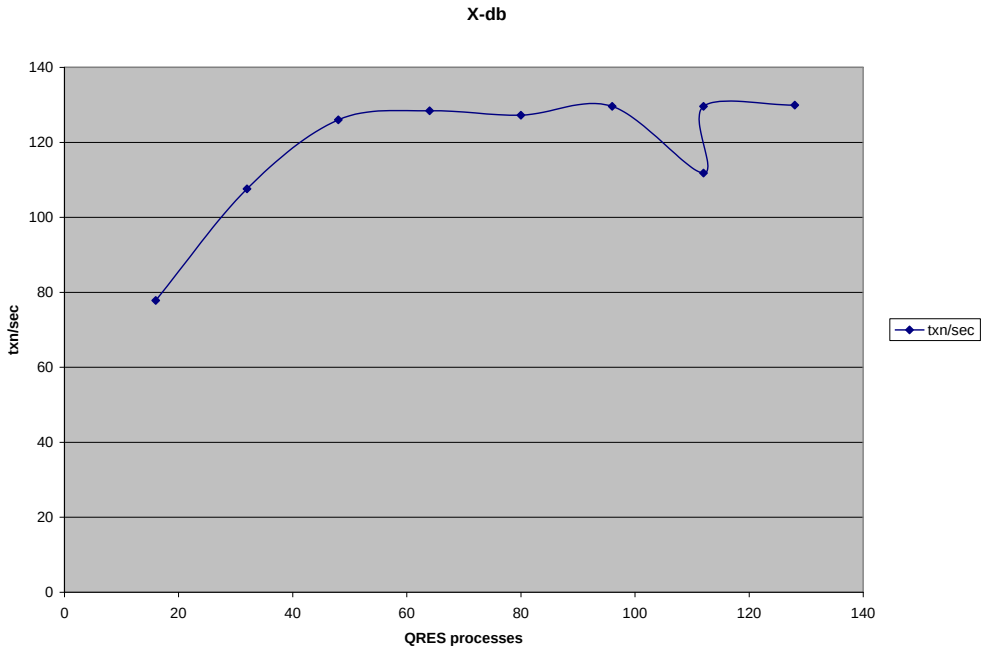

Fig. 13: Database throughput in txn/sec.

What do we see?

- Throughput saturates for both the database and the system at the same load (~40 agents)
- The Residence time for the system is increasing, but that for the database remains flat.

- Queueing in the database remains relatively constant as additional agents are added to the mix.

What is going on here and how does it relate to the first case? Here is my simple picture. Imagine that our work flows are circular, passing through each queueing center in series (pounder → application server → database → pounder). In reality there are additional loops and paths, but that is just an additional perturbation which can be dealt with later.

Throughput should increase linearly with load (# of agents) until saturation is reached, at which point the throughput remains constant (it actually peaks and begins to slowly decrease, as discussed by N. Gunther, but that also can be ignored for now).

Residence time remains constant until queueing begins. Prior to queueing, the Residence time at each queueing center is just the service time (of course there is some distribution to service times and arrival rates which affects Q). After queueing, we start to wait for our resource.

Regardless of whether the database was or was not the bottleneck, the db and system throughput saturate at the same load. Even if the bottleneck is outside of the database it will throttle the throughput to Oracle as work travels through the bottleneck on the way to/from the database. The Residence time, however, is different. If the bottleneck is within the database, as agents are added, they will be spending more and more time in the db as waiting time increases. However, if the bottleneck is outside of the database, these additional agents will just scoot through the db and the Residence time will hardly change. This can also be seen by the different behavior of Q with increasing load. Thus by looking at Q,R, and χ as a function of load, we can immediately see whether or not the database is the bottleneck.

Further investigation revealed the bottleneck in this example was in the firewall. Being able to ignore the database almost immediately saved us from wasting a good deal of time.

Quick Triage

We had a test comprised of two 'identical' runs, each using 24 agents with the same workload. The performance characteristics, however, were nowhere near identical. Run #1 completed in 9.41 minutes while Run #2 completed in 15.44 minutes. In addition, the statspack reports for Run #1 showed all sorts of gc contention. What was going on here?

This was only a single run so I couldn't examine any metrics as a function of load. I could, however, get the system and database throughput, residence time, and queue length for my single load point. This is what I saw:

Table 3

Run	Elapsed Time (sec)	Q in database	Response Time – system (msec/seq)	Response Time – db (msec/seq)	Time/seq outside of database
1	9.41	18.2	449	166	283
2	15.44	6.4	489	94	395

So in Run #1, about ¾ of the agents are working in the database, while in Run #2, only about ¼ are. The effective database load is not measured by the number of agents or users in the system but by the queue length, or average number of active sessions in the database. This also explains the increase in gc waits for Run #1. In this case, it appears as though something in Run #2 is stalling the agents outside of the database. Further investigation revealed a change was made to an external application called by our application server sometime between Runs 1 & 2.

Drilldown using Oracle wait events

Going back to our case where the database was the bottleneck, I wanted to drilldown further in order to discover the bottleneck *within* the database. In other words, change the labels for Figure 6.. Remember that the throughput began to saturate at about 30 agents (Fig. 8). Also, I/O was a problem on this hardware. I needed to set _disable_logging =TRUE to sidestep saturation from redo generation. (Since this was strictly a test system I could take this chance).

My first thought was to try to plot residence times for some of the top wait events (I peeked at the statspack reports to help pick and choose events, but this could be automated) and compare this to the database residence time.

First, a clarification. What is called a wait event in Oracle's documentation is not actually a wait in the way queueing theory uses the term. A read, for example, is a service necessary for database functionality. Getting the read takes some time. It is possible for a number of reads to queue up and wait for that service to be provided. Hence the wait time given by Oracle is actually a residence time at that queueing center.

Getting back to our drilldown, we find

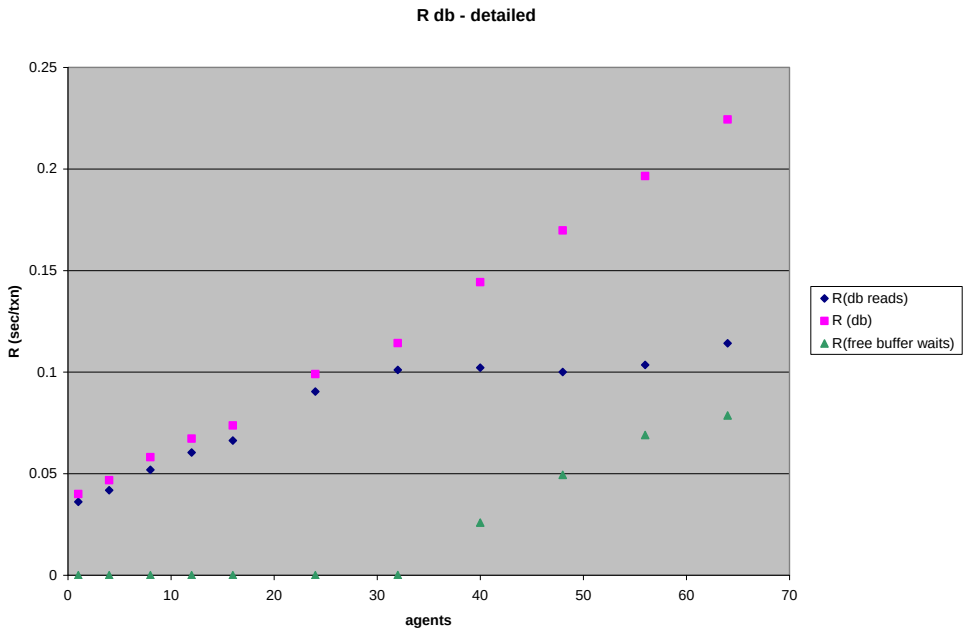

Fig. 14: Response time per transaction for the database and for a couple of queuing centers within the database.

Additionally, I plotted DB CPU as a function of load.

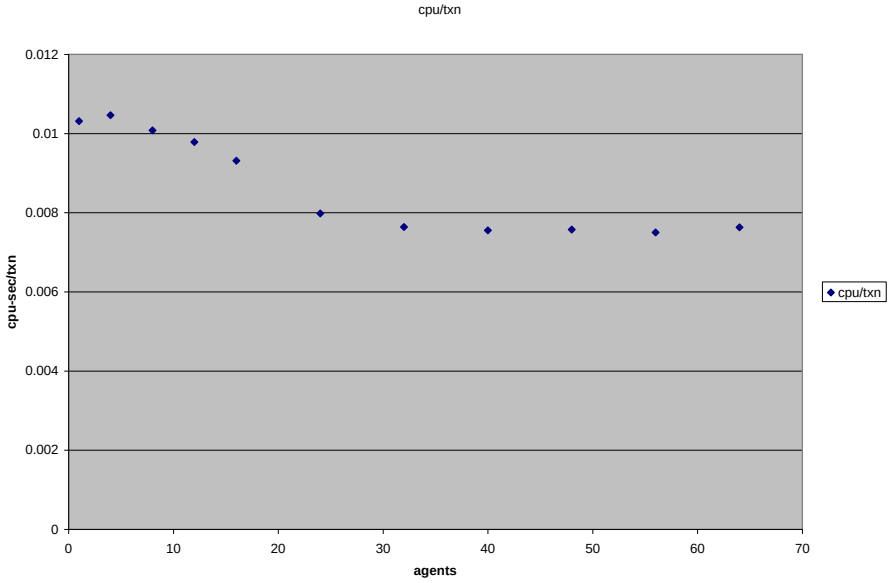

Fig. 15: CPU time per transaction as a function of increasing number of agents.

What is going on here? This is where our simple model begins to fall apart (given the complexity of Oracle, this isn't surprising).

- At low loads, prior to saturation, most of the time spent in the database is spent with reads (sum of scattered and sequential).
- At throughput saturation the time spent doing reads becomes constant. (It looks more like a throughput curve, but it isn't). This is not the expected behavior of Residence Time. As load increases, more agents make more calls, and the queue length waiting for a read should increase. This increases the wait time for a read and hence the residence time should increase linearly, not saturate. The saturation implies a choking of the number of read requests. As load increases above 30 agents, the number of read requests per second remains constant.

  This choking happens within the database. The number of agents within the database continues to rise (fig. 7). The Residence time of the database continues to rise (superlinearly). We see most of the difference between database residence time and read residence time being made up for by 'free buffer waits'. Could this be the throttler?
- Low load behavior predicts that database read time is the database bottleneck, but the saturation we see at 30 agents comes from a different mechanism
- Notice that most time isn't spent in the 'effective bottleneck'. Improving response time is often different from improving throughput.
- The service demand of the 'free buffer waits' bottleneck, using the rising slope of the residence time curve (see fig. 4) is ~28 msec. The average time of a free buffer wait (as seen from statspack) is 11 msec. This implies about 2.5 visits per transaction. The number of free buffer waits per transaction , however, is not constant with load. It increases as load increases, (from 2.5 at 40 agents to 7.9 at

64) with the average wait time remaining constant at 11 msec. How does this fit into the model? The aforementioned superlinearity of the database response time suggests a load dependent queueing center (residence time increases faster than expected with increasing load). Is this what we are observing?

- Why isn't the residence time of db-reads flat prior to saturation? Queueing is seen to start at low load levels. This could arise from the dependency of queueing on the service time and arrival rate distribution. A broad distribution can lead to queueing at low loads.
- CPU utilization is not a realistic reflection of database service time. The service time of the database (R as load approaches 0) as seen from figure 14 is ~40msec, while the time spent in cpu/txn is less than 10 msec (fig 15).

One other point is to be careful with the difference between service time and service demand. Initially when I saw a bottleneck service demand of about 10 msec, I thought I/O (especially since I knew I/O was an issue in this system). Additionally, with 6.3 physical reads per transaction (see the number of scattered and sequential reads), 6.3*10=63 msec. Pretty close to the database service time of 40 msec. However, this is a misinterpretation. With a service demand of 10 msec and 6.3 visits/txn, this yields a service time of 10/6.3=1.6 msec, much too short for I/O time.

Sleep time calculation

We can find an uninstrumented sleep time inherent in Pounder by using the data in Figures 8 and 9. The low load slope of system throughput from figure 8 yields the total round trip time (D+Z) (see figure 4) which we find to be 258 msec. Remember, this comes from Pounder data.

Now moving to our Residence time plot (figure 9), we can find the Service Demand of both the system and the database by looking at the residence time as load approaches zero. This is both Pounder and Statspack data.

Knowing both the round trip time (RTT) and service demand from the point of view of the system and the database, we can calculate the sleep time from both of these perspectives.

Table 4

msec/seq	Residence Time plot	Throughput plot
D+Z (system)		258
D(db)	70	
D(system)	166	
Z(db)	188	
Z(system)	92	

Thus we see that even though Pounder is configured to have zero sleep time, about 35% of the round trip time is spent sleeping. This won't affect our overall analysis, but it means that our effective load is always less than expected.

To further check this finding, we use the simple model that the total time for a round trip, R + Z. will be the same regardless of whether it is measured from the point of view of the system or of the database:

R(db) + Z(db) = R(system) + Z(system)

Rewriting this

Z(db) – Z(system) = R(system) – R(db) = 96 msec

Looking at figure 9, we see this is very close to our experimentally measured R(system) – R(db). Since the data used comes from multiple sources this is good confirmation of our analysis.

A few months after this analysis was done, saturation of the pounder box was confirmed. Pounder has since been changed to be a multi-threaded application spanning more than one server.

RAC Analysis

Load tests were also run to determine the effect of RAC vs. non-RAC systems. Because of high levels of gc contention, we also ran some tests where data was partitioned to particular nodes within our RAC database. A small number of tests were then run comparing three different scenarios:

a. single node database
b. 3 node RAC database with no data partitioning
c. 3 node RAC database with data partitioning

We compared performance characteristics across all three scenarios at a single load (24 agents), as well as examining case (c) at multiple loads.

Table 5

Run type	χ (txn/sec)	R (sec/txn)	Q
A	166.1	19.5	3.2
B	42.3/41.2/41.2 (=124.7)	93.3/97.8/98.5 (=96.5)	3.9/4.0/4.1 (=12.0)
C	49.1/49.2/49.0 (=147.3)	31.2/32.4/33.1 (=32.2)	1.5/1.6/1.6 (=4.7)

As expected (assuming pre saturation), the single node system has both the highest throughput and the shortest residence time. Even with data partitioning, our residence time is almost 50% higher than for the single node case. This might or might not be significant depending on the service times from the rest of the system. What RAC does buy us is seen from the queue numbers. In case (c), for each node the queue length is about half of that for the single node system. This means that our load is being spread across all nodes (the increased overhead of RAC means the total queue length in the database increases), allowing a more constant (albeit higher) residence time over increasing load.

Runs at different loads for case (c) didn't add much additional information except for confirming that the database was not the bottleneck and was not saturating at loads up to 64 agents.

Part V - Conclusion

This paper is a first attempt to apply queueing theory to database performance. It is only a beginning, with a lot left to be done. A few of the topics I have yet to examine include:

- Mixed workloads – how can we distinguish between different kinds of work happening simultaneously? Service times on the same queueing center can be different for different workloads. Given the presence of background and foreground processes in an Oracle database, mixed workloads are always present. Tanel Poder's sesspack scripts (http://blog.tanelpoder.com/2007/06/24/session-level-statspack/) might provide a good start to this issue.
- Relationship between Service Time (S), Service Demand (D), and Visits (V)
- Relationship between # of application sessions and # of database sessions – up to now, I assumed each agent spawns a single Oracle session. This is not necessarily the case, though typically there is a dominant Oracle session. What if some database work can be done in parallel? How would this impact analysis?
- Add OS drilldowns – I intended to include collectl data, but never had the time to do so
- Parallel queries – in a communication with Doug Burns, he suggested examining parallel query sessions in the context of a closed queueing system.
- SQL statements as queueing centers – since a database executes SQL, can we model our database queueing centers as SQL statements instead of as more common resources (i.e. cpu, disk i/o, …)?

Part VI - Appendices

Appendix A

What am I really measuring?

One piece that continues to make me uncomfortable is my lack of confidence in knowing what a particular metric actually measures. Is DB time or DB CPU what I think it is? (is DB time from v$sysstat the same as from v$sys_time_model the same as from statspack?) (for some discussions on DB time and CPU times see http://www.freelists.org/archives/oracle-l/11-2006/msg00573.html, http://vsadilovskiy.wordpress.com/2007/11/05/recursive-calls/) What about some of the data from iostat, colletl, etc? Does iostat tell you something different with/without AIO? It is easy to assume you know how the data is being collected, but it isn't always easy to know, and I hate guessing.

There are also complexities in associating the measurements to the parameters. For example, say one Oracle user uses 5% of CPU. Then 2 users would use 10%, 5 users use 25%. Say at 10 users we start to spin for latches. CPU utilization may now by 60%, not 50%. How do we account for these non-linearities?

We also have different workloads mixed in different ways. We have disk access via LGWR, DBWR, and user sessions for additional PIO (reads). Some of these are foreground processes, some background, sequential, scattered.

Maybe experiment will help bail us out with some of these difficulties.

Appendix B

Relationship between Q and AAS (Average # of Active Sessions)

After listening to Kyle Hailey's 2008 Hotsos presentation, and working a bit more with 'Average Active Sessions' (guess I looked at it more carefully when I realized other people were also using it), I finally realized that AAS is the same as Q. (Since then I found out, via a personal communication, that John Beresniewicz already made this connection).

The queue length, Q, is just the total number of sessions waiting and currently being served. This is also the definition of active sessions. Additionally,

AAS = (DB time)/(Elapsed time)

and from Little's Law (eq. 3)

Q = (throughput)(Residence time) = (txn/sec)[(DB time)/(# of txn)] = (txn/sec)[(DB time)/{(txn/sec)(Elapsed time)}] = (DB time)/(Elapsed time) = AAS

Acknowledgements: I would like to thank Luis Ordonez, Devon Krisman, Daniel Lowe, and Nirmal Veerasamy for running the load tests and producing the data used in this analysis.