

The Eternal Donut of the Soul

Questions and Answers

podcast@xor.com.au

Questions and Answers

These questions and answers are just another outlet allowing me to go off topic and discuss some more interesting and enlightening subjects.

Disclaimer: The content of these podcasts and papers are the sole view of Marcel and do not represent the views of the sponsor or anyone else, unless they otherwise stated. They at times aren't even the views from Marcel who if push comes to shove might disavow all content penned and claim it was induced during a donut consumption frenzy brought on by a sugar high. Unless of course you agree, in which case the arrogance kicks in, and he will say that he humbly and meekly accepts the acknowledgement of a job well done.

1.1 What is a lazy DBA and why is this good?

Simply, it's the ability to work hard in the short term, to make long term gains. To spend extra time and effort to write a program that when run will automate your job. The lazy DBA thinks long term, has the bigger picture in mind and has a good understanding of what is needed and is willing to do what it takes to get it done, even if it's on their own time and when chastised by co-workers for doing it.

1.2 Aren't technical skills the most important to have?

Technical skills are a crucial and important element. They are part of the solution but not the whole solution. I would say most DBAs know that having people skills are important, especially for tuning but might be reluctant to admit it, as it takes effort and can be stressful dealing with users, management and developers. It's typically out of their comfort zone. Being able to communicate simply to management, explain problems and help diagnose tuning issues with users and also talking intelligently but not arrogantly to developers is important. It's a very key skill to have and can take time to acquire. It's like public speaking, a lot of people hate it, don't want to do it, get nervous speaking, but only through practice and more practice is that fear overcome. It's also continual practice, every couple of months. Practice, practice, practice. Overcome the fear, get the rhythm going and keep at it. It's key to moving to a proactive database environment where problems are fixed before they occur and it helps big time in terms of gaining trust with all parties.

1.3 What's with the attack on middle management?

Out of all the groups I continually attack (more question) it's that of middle management. Not to be confused with HR, Sales or senior management (which is another story), you will see throughout the podcasts that I am constantly berating this group of individuals. Even reading many Dilbert cartoons and listening to the Cranky Middle Manager podcast has done nothing to help me change my mind in their role.

First, let's see what middle management is about. If I was in a small-ish company and reviewing everyone's performance to see who was earning money to keep the company going, how would I rate middle management? Well for starters they don't directly earn any income. Their position, salary, desk, office and perks consume a lot of money. So do we get value for money out of them? If we put a chimpanzee in their spot, would it do a better and altogether cheaper job? Would random gestures or letting the staff do their own thing be better? For what they are paid, could someone write a program to replace them and do a better, more consistent job? Would using the money for the middle manager be better spent on training the staff? To answer all these points, I would say, yes.

In addition, as a lot of middle manager are promoted through the technical ranks, they are just a drain on the existing talent pool. What do they really do?

In reviewing books, listening to podcasts and just looking, all I can see them being responsible for is:

1. Leadership. They are the head of the group, providing leadership. I'll get back to this.
2. Budget approval. Approve spending.
3. H/R. Approve leave. Administration.
4. Make decisions.
5. Establish the culture of the group
6. Architecture. Review and do some basic project management. Co-ordinate the team which is the same as leadership.

Am I missing anything? Out of these first five points not one earns income. Leadership is a very hard skill to acquire and most technical people are naturally introverted and make lousy leaders. Occasionally one who is good, gets to the top and sets an impossible standard for the others to follow. There is a myth that leadership can be taught through training courses. For some it's natural, for everyone else it's acquired through practice.

Approve spending and doing H/R tasks is best handled by people in those areas like accountants and H/R people. Shouldn't they be making these decisions? And as for making decisions, most organizations want managers to justify the decisions, have choices and most of the time managers assign the task of deciding these things to their staff.

The staff come up with a number of choices and in a nice executive summary they indicate which one is preferred. This is not an efficient use of anyone's time. There seems to be a fear of risk, the need for constant accountability, and not trusting one's staff to make a decision (even though as just pointed out, most of the time they make those decisions but are not allowed to sign off on them).

My personal view is that when it comes to cost cutting and reducing expenses, middle management should be the first to go and not the ones producing income. Organizations should be looking to question the need for their existence and coming up with a different strategy for how to replace this middle layer.

Though I am not skilled to say how they should be replaced, I have put some thought into it and have some ideas which should at least make people think.

1. It doesn't make sense to have a middle manager for every five or six people and be subjected to the inconsistencies of their behavior and attitudes. It's also unfair that some staff might get the luck of the draw and get a competent manager whilst others don't. So think horizontally. For every 100 people, hire a person who is a natural and skilled leader. Someone who can motivate staff, listen to their issues and get the groups talking to each other. No need for technical skills, all that is needed is the skill of getting the staff to be led. And pay good money to acquire this person.
2. For accounting tasks, get a manager who is skilled in this, who knows the organizational budget, the requirements, the pressures and get them to sign off on all expenses for the 100 staff. They will have a global picture and understanding and be able to listen to their various budgetary pressures and balance it.
3. For human resources, get someone in who is skilled in this area and knows how to deal with people. Get them to approve leave and be in a position to know when staff are overworked, stressed or in need of a break. Get them to regularly meet with staff, discuss work issues and have the authority to act on them and do something about them.
4. A technical manager. One that is responsible for all technical decisions. One that can make the decisions that are crucial to the running of the group (of 100). They are not leaders, they are not motivators their focus is on determining the technical merits of decisions that need to be made, then make them. They are also there to do basic project management on the staff and ensure they are all focused and aware of the work they need to do. They don't have to worry about leadership, admin tasks or motivation.

So that's 4 highly skilled and well paid people to look after 100, which is ten to twenty less than what is normally needed. It would be a tough job for them, but as it's their natural ability they would be able to focus on what they have to do and get the job done. No need for middle management at all.

And the icing on the cake is by doing this, the skills within the technical area are kept and not lost. Then promote and pay people more for those skills. Don't lose them to middle management.

And the other plus side is that as 90% of most meetings are seen to be redundant and not needed once can save more time. I put most meetings into a special category called management futzing. Futzing is wasting time on frivolous activities. Technical futzing occurs when you are very busy, seriously busy, but do not achieve anything, because the activity you are doing has no immediate benefit to the project you are working on. A good example is when your computer gets hit by a virus, you can waste days being very busy recovering from it. It's stressful, but you don't achieve anything. Or when a disk crashes, or when Windows makes you wait one minute to open a document, or when you spend an hour trying to work out why Word spell checks in American English and not British English, and then still not solving it.

So management futzing is attending meetings to say something that could have been said in 30 seconds, whilst munching on a donut and socializing with members of another team. Management futzing can occur in areas not just limited to attending meetings, it can also be when you have to write a justification to ensure there is accountability for a decision made. Especially when no one really cares about the decision, it's just done, just in case.

1.4 Demand food to do tasks. What's with that?

As mentioned, management and individuals are very good at putting barriers up. They can be physical ones like walls, buildings and partitions. They can be bureaucratic ones like paperwork, red tape and change management. To get the communication going between the various teams, these barriers need to go down. The easiest and simplest way to do it is via food. Donuts are the best as everyone wants them, people feel guilty about eating them (well I don't) and they are a great networking device and way for breaking the ice. You can always talk about donuts, and the conversations I have heard just on how people don't want to eat them because they are fattening. The standard tactic is to cut one in half and only eat that half, then eat the other half, *cause* we know you cut it in half and the calories disappear. But it gets everyone discussing, networking and hopefully passing information to each other. Beer works equally well, but as a lot of sites don't allow alcohol on the premises and it can also be hard to organize everyone to go to a pub for a brew. Actually beer can be better but we can leave that for another story.

1.5 Why would anyone want to make their position dispensable?

It goes against the grain of thinking, after all if our position is indispensable we can't lose our jobs, we hold an element of power, it feels good. Problem is, it's not healthy for the individual and for the organization. Having all that key knowledge focused in one person is dangerous, especially if they then leave (also referred to as being hit by a bus). Defence organization know this quite well, and are rigorous in their procedures to ensure that if someone goes they can be replaced by someone reading a manual. Going to this extreme doesn't work either as you just can't read a manual and know IT or how to program. So in this case there is a healthy compromise between the two extremes.

2.1 Are there any scenarios where foreign keys are OK to have?

Yes. In some rare cases the optimizer can use the knowledge gained from knowing a foreign key exists to determine a better strategy for optimizing that query. In some financial systems where it has been determined that the data has to be accurate 100% of the time, without fail then strict conformance might be seen to be acceptable.

2.2 You talk about two-tier versus three-tier and you don't think three-tier architectures are any good. Can you explain why?

First it is best that I clarify the tier structure I am referring to. Traditional two-tier is a client containing the interface and application code interfacing to a back end database. In three-tier, the application code is placed into a middle tier. It's possible to extend the database back end into multiple tiers by allowing multiple databases to talk to each other.

The tier structure I am talking about as the direction we should be looking at, is one that puts the interface only on the client side, and it communicates to a back end database containing the application code and database. The application code is kept separate from the data, but is tightly integrated with the database. This is done for performance reasons. The architecture looks two-tier and behaves two-tier, but is not the traditional two-tier we are used to seeing.

The core idea behind this architecture and why it is fundamentally different is that this structure moves more towards the idea of the data driving the application and not the application code. The mentality is to use the data to drive the app - not the other way round. A good example is hierarchical menus. You can build a standard pull down menu and hard code all the options in it, or you can store the information about the menu items as rows in a table, and then build a generic routine to display it. You can then display it as a pull down menu or hierarchical menu, or change it to new technology when it appears. It's the data which is important and it's stored in the database and managed there. In the case of a web-applications, PL/SQL code can be very easily written to generate HTML that displays the different menu structures.

Developers put the code first, DBAs put the data first. Being in both camps, I say data first, code is second. The data drives the app.

The other key concept behind this architecture is that it is focused on performance and scalability that is suited for today's hardware. The move towards multi-core chips now encourages database servers to do more of the grunt work (which is just like it was 25 years ago). The difference is that we give the clients complex instructions on displaying the information. In fact the clients are becoming very heavy. They have to run interpreters for many languages and generate complex graphics. They have to deal with HTML, XML, SVG, Javascript, Java, Flash and be able to handle plug-ins from Adobe, Office and a multitude of video and audio ones. They have to be able to multi-thread and deal with multiple

windows, quickly adapt to different sites and ensure the security of the data coming in. This means that the client interface, rather than being light weight is now requiring a computer with decent performance and memory. As more and more XML standards start to appear, the browsers will grow in complexity to deal with this requiring even more processing power.

Now back to the data driving the application and not the application code. It's a core and fundamental difference that not many people understand. When you use web based products such as ASP, JSP, Cold Fusion, PHP and PSP, they encourage static page development. You build the HTML then embed some code in it to talk to the database and get data out. This static page mentality has limitations in what can be delivered and built, but and this is the key but, it limits how the developers think and build pages. It's also lousy for doing complex interaction with the database as writing SQL to get the data out is primitive. The developers are encouraged then to access the database as a last resort and thus put control into the app.

When you develop a web based application that is generated out of the database using PL/SQL, the attitude to page development is very different. It's incredibly easy and very fast to get data out. So you put more in. You make more generic routines that generate pages, and you create web pages that can morph and change themselves as the data changes. You are not limited to static page mentality, you have more flexibility in development - and it's very fast.

For anyone who is thinking, well with PL/SQL you are locked into Oracle, where as if you use PHP its open, I refer you to the podcast on being locked into Oracle.

And this leads to another issue. All the models we use as a base for making application design decisions and justifying why we build applications a certain way are all flawed. OO, UML and Relational models do not give two hoots about performance and scalability. Most books give a nominal mention to indexes which is a start, but that's a hack. They never address the core fundamental issues with the design. They encourage building apps that are wonderful to look at on paper but are unrealistic to use. The mainframe programmers in the early days pointed this out about relational, and it's still true. The difference between now and then is that most database vendors have addressed performance issues and CPUs have become a lot faster. Buts it's still a serious issue.

Now onto data accuracy. All these models assume that the users know what they want, know their data and have accurate data. Nothing can be further from reality. How can a designer create an accurate model from inaccurate data? The users don't have accurate data and never will. Whilst we continue to think in the black and white thinking we will live in this magical world where one day we will obtain nirvana and get all the data accurate. It's not accurate and it never will be 100% accurate. Each table, each object will have grades of data accuracy. Some might be known, some unknown.

Here is an example. It comes from calculating row sizes in the early days. As a DBA I was responsible for working out the total storage down to the byte that a new application would use. We would review the application, get figures from users and developers and in the end there would be lots of guesses, which may or may not have been admitted to. But we would come up with a magical figure for database sizing.

There were even formula's provided by Oracle to help out the sizing and voila we would know how much data the application would use for the next six months. Problem was, it never did use it exactly. That's Boolean thinking for you, if used on its own its dangerous, leading to bad decisions and imprecise values.

Start using fuzzy logics concepts (and other AI concepts) and you realize its futile getting accurate data. You learn to live with data not being accurate and then focus on what is important, putting fences around it, controlling it and managing it. This is why statistics can be so important, its helps us handle and control imprecise data. This is another reason why I have issues with foreign keys, they give everyone the illusion the data is accurate.

So the models are flawed and following them exactly is dangerous. The idea of these models is to allow us to map real world concepts to them, but if they don't take into account performance of machines, then even a DBA guru can have trouble making them run fast. Let's look at another one of my favourites, XML. Incredibly simple to use. Really simple to use. Dead easy. Try passing volumes of XML data over the internet and we realize it's not that great. And that's because the person who came up with XML didn't care about network performance, that person assumed infinite bandwidth. So if we follow the XML mantra and put in namespaces everywhere and meaningful tag names and break up the data into a wonderfully looking hierarchical structure we might achieve something that looks pretty but from a performance point of view is horrible. So to make XML scale on the internet we need to come up with an ultra lightweight design, one that minimizes the tags and junk that surrounds it and one that passes the data back to the app. Case in point, for an Ajax application I built, just by changing the tag names from something friendly to something lightweight and moving some data into attributes and thus removing a number of elements, we improved performance from 45 seconds to retrieve 1200 records to under 3 seconds.

At this point in the discussion, developers will chime in and critique things by saying if we adopt my strategy we will never have accurate data and the world will be chaotic. We might as well not bother with relational, normalization, modeling and such. We should just live in the filth of inaccurate and uncontrolled data. And that is the reason why we should follow these models.

Hard to argue with that strategy, except if we live on the little hill that says we have to use traditional modeling, then this is in the only conclusion. If you are on another hill, living in a more realistic world, you come to different conclusions. You begin to see that these models are part of the solution, they are limited, better modeling can be done, but you don't rely on them and assume they are accurate. You come up with other strategies. By all means use normalisation or UML and come up with a pretty design and pretend you understand the data, but don't get attached to it, and don't for a moment think it's fixed, rigid and can never change. Don't every think that the structures have to be rigidly adhered to and cannot change, not even drastically change. Don't force the users to think that they have to change their lives to fit with the model, especially when the model deviates from their changing business practices. And believe you me, they change and change on a regular basis. The operating system changes, the database version changes, user requirements change, the browsers they use change, the hardware they run on changes, and so on.

Technology changes during the life of an app. If you build an app on lets say Oracle10GR2, then by the time it is developed (assume 6 months) the odds are pretty good that a new release is out. So what do you do?

Ignore the new release and hope it goes away? Bad mouth the new release – say its buggy, say it's the first version and you should never migrate to it, ignore the version, leave the upgrade to someone else to deal with, or try an upgrade without code change?

There are so many variables now that can result in change happening it's not worth the effort getting caught in a static and self imposed design.

So, let's embrace the fact there are always new releases, factor that into the design, then start reviewing the new release as it's being developed, look at what is coming in it, and start adapting your current app to be able to take advantage of those new capabilities when they are there. So when you migrate it becomes easier. So who does this? Hardly anyone because the culture of organizations don't allow for it, all the models we have don't factor it in, in fact it's not even encouraged as its adds to instability of an environment and being unstable is bad - isn't it? As I have shown, by changing this view point and staying stable is not the right way to go. By embracing instability and using it to your advantage, then you can have your cake and eat it.

So put the code as close as possible to the data. Developers need to wake up and smell the roses. Code is code and changes, but data is core. So wrap the data in the code and use the data to drive the app, not the other way around. Keep the two separate but close together for performance. Why performance? Because in a three-tier app, the cost of passing requests for data back and forth between the database and application layer can be very expensive and become a bottleneck. It limits the amount of data being passed and encourages developers to only call the database when need be. When you run PL/SQL in the database, you access the database when you need to. It's a different programming mentality and as such you can do much more.

On a quick side note, Developers also need to realize that web design and simplicity is best. The reason why HTML and web browsers worked and worked well was because HTML pages were simple and easy to use. As more new features are given to browsers, the more complex pages will become. They will become harder to use and we will get a paralysis of internet usage. Only the developers can be blamed for this and no one else. Saying users wanted it, is a lot of putz-bah. Users want easy to use web pages, they don't want to think, and the pages must be quick to download. Developers need to realize that web programming is not something to be done to boost their ego ("oh look Janice how spiffy I can make my page").

2.3 Why knock object-oriented programming?

Because of performance. The object oriented architecture, for the most part encourages bad programming that does not scale. When used well, you can build applications that run faster than relational. But for the most part, if developers are not reigned in and controlled by a dictatorial DBA then any app they build will be unlikely to perform well and will only be able to go onto the middle tier of a three-tier app.

There I said it. That's one main reason why we have three-tier, because of bad programming, because the application that is built was not built to scale, and because it runs so inefficiently it has to be isolated from the database so it will not impact its performance.

I will cover this topic in greater detail in another podcast. Lets just confuse everyone and say that when using OO you can get great performance, extremely fast applications, but when used incorrectly all hell can break loose.

2.4 Do you live by the mantra you proclaim of question everything?

Yes. I am questioning everything I do on a regular basis. It can be hard, but it does require a different thinking strategy to achieve it. It's one major reason why I have such an issue with the perception that binary logic is the only way to go. This is where the thinking is that the ultimate direction is to become a Vulcan of the mind and to think calmly and without emotion about all matters.

Edward De Bono knocked this concept on the head years ago and it wasn't just coining the term lateral thinking. Which is easier said than done, he actually described in later books how to think differently, how to train the mind and break out of the box. Basically, how to question things, and by questioning come up with new solutions, concepts or strategies.

One method to highlight this that I like, is when someone uses mathematics to prove something can't be done. They will say it's proven and that's the end of the story, no arguing there. Mathematics was used to prove rockets couldn't fly into space and ships couldn't carry enough coal to travel the oceans. It was only when one someone stepped back, looked at the assumptions that were used in the proof and questioned them did new ideas and strategies appear. What if we questioned the need to carry the whole rocket into space? We do that and we can get there. What if we change the shape of the steam engine and use a more efficient structure?

Do that and we can travel the oceans. In technology terms, the assumptions made are based so often on the technology of today, which is so quickly out of date, that it isn't hard to question an assumption and change it.

So when someone says categorically that it can't be done, and you know or you want to know that that isn't right, review the assumptions made that led to the decision and question it. Even if it all looks correct and above board, change the assumptions and see what happens. Use the "what if" scenario? See if that gets a new answer and goes down a new direction or path. That will then change the rules.

The other strategy I like, is to question the concept, "you can't have your cake and eat it". It's used in performance tuning, design and a number of computer architectures. I look at it and say, but what if you could? If you could what would happen, what results would occur, what new concepts would appear? These thought experiments are useful in that they form the basis of new ideas. Which initially might not be useful but one day could be.

Thinking, coming up with new ideas, it's a strategy. The obvious one is that "necessity is the mother of invention". It's the easy one, the one most people think about and come up with lots of ideas. What if we had no petrol, how would we run our cars? What if I had an air-conditioning unit that was put in the computer room resulting in downtime, and the DBAs were not told about this happening? – I know, I'd invent change management.

The one major lesson learnt when looking at the different thinking strategies in IT (and there are more), is that the best way to remain on top of the constantly changing technological world, is to always be unstable. From the Jackie Chan, drunken fighting style to the F/A-18 Super hornet, the most flexible and adaptable systems are one that are inherently unstable. They can maneuver quicker, change direction quicker. It's hard to do, requires skill and training (and at times a high speed computer). Being stable might make you feel good, but you will go out of business quickly if you adopt this strategy.

At the time of writing it was interesting to note how Digital Rights Management and music (the stability insisted upon by the music industry) was causing a lot of record companies to go out of business, purely because they were too stable and could not adapt. The irony is that Apple computers, who were once sued over their name by Apple Music, might via iTunes become more dominant in the music industry, just because they changed tact and threw out the DRM concept as being too rigid and unwieldy to work with (that last bit are my words not Apples).

This is why I believe pure research is so important. It allows ideas to come from left field. I am sure a number of people question the reason why researchers do research with no observable benefit.

How do you come up with new ideas – concepts? It's not by specializing. This only improves an idea. Equivalence is binary logic, it can only be used to refine and improve, but never can be used to come up with new concepts and ideas. Fuzzy logic is a step in the right direction as once we start to understand it, we don't get focused in the little details and we start to see the bigger picture. But how do we get there? Neural programming helps us get there with using random concepts. That is, we use our brain and some constructs to push us outside the box we are in.

The other analogy used is there is no right one answer. There is no perfect web page, there is more than one way to climb a mountain. If we can imagine multiple solutions, with each solution being a hill. The better the solution, the higher the hill. The aim is to find the highest hill – but how do we do that? Binary logic will get us to one hill and allow us to climb to the top. Once there we can't go any further. It's like specialization; we can only go so far. Once there we might see the other peaks, know they are there, but we can't get to them.

So how do we get to them? We need to think laterally. Which is easier said than done. "Junior, to solve that problem you need to think laterally – yes dad, oh problem solved". So how do we do it? The best method is randomness to enable us to break out of the mindset we are in. It's like being asked to be randomly placed on the map and hunt for a hill. We get to one and can try and see if it's taller than other ones (analogy here is brainstorming). Or we can try and guess where the highest hill is (focus groups) or we can get a number of people to scan out and find some hills and see if they are the highest ones (testing groups). One method is to start with a random word from a dictionary, doesn't matter what. Then relax then think of associations with that word, go from there. It's like a large funnel of random words, jump from one to another and they might settle on the right concept. It might not, but it should get you thinking in a totally different direction that could get you to come up with a lateral idea.

The solution to problems isn't by specializing in your field. It isn't by being the best XML programmer or SQL writer (or by becoming a PL/SQL guru), it's by looking at other fields and learning from them. For example, how do you design an efficient screen? It's easy to determine a bad one, but how do you come up with a good one?

This is why I like the idea of OZCHI. By using psychology to determine how people think, artists to come up with concepts from left field that may or may not be practical (who don't understand technology and aren't constrained by the self imposed limitations technology has) and by using the developers to see how they get integrated. You come up with different solutions. The best solution might not result, but it's a starting point. If you combine these three you will get great screens, but the group missing are the DBAs who can then determine if they are efficient and can scale. Don't forget the network and security people who can point holes in the architecture. So multiple disciplines working together can solve it.

3.1 What's with this Mr Whamo?

Mr Whamo is a device used to grab your attention. If you think I walk around to developers carrying a baseball bat and use it as a method for intimidating and controlling developers, then you are mistaken. It's just a fantasy. Like owning my own private donut shop.