

The Eternal Donut of the Soul

2. Question Everything

podcast@xor.com.au

DBAs and Developers should question the technology and the decisions behind design and programming techniques they use on a regular basis to see if the original assumptions for using it are still valid. Because technology, particularly Oracle technology, is evolving and changing on a regular basis assumptions once made might not be valid anymore. In this podcast we are going to review a couple of these assumptions and tackle one that is highly controversial, so contentious is this issue that I anticipate being severely rebuffed by all who listen. **So let's bring it on**

The traditional concept of never having a table grow beyond 5 extents, was fact ten years ago, but is now known to be invalid because the architecture of the Oracle database changed. The rules changed.

So what other rules that we were brought up on, once fundamental to our programming and tuning belief, are not valid? What are the fundamentals that are valid now yet may not be valid in a year from now?

A good example is, "We shouldn't program in PL/SQL because it's slow". This might have been valid in Oracle7, but by Oracle10 you will find it hard to justify this view. By Oracle11 we find its easily compiled allowing very fast performance. In Oracle10 it was shown that it runs incredibly fast run in semi-compiled mode (see podcast #5 on PL/SQL myths).

It's well known that in Oracle8 and Oracle9 I didn't think much of RMAN, in fact I recommended it should never be used. By Oracle10 I changed my tune, by 10.2 I am now recommending that it be used. This is because it matured and key issues seen in earlier releases were addressed by 10.2.

"The database is too slow to generate HTML pages," is another long held belief that should be reviewed. In fact, my view is that using PL/SQL to deliver HTML out of the database is the fastest method possible - faster than Java and any other tool in the market.

Another traditional view that is not valid anymore is that "The database is too slow to retrieve images from the database. Images should be stored in the file system" (see podcast #6 on Intermedia).

There are a large number of features that I am sure we once dismissed as being useless, buggy, flawed or immature, that are now worth re looking at again. From the DBA side to the programming side, there are features, styles and programming techniques in PL/SQL that I am sure we program to because of problems encountered and performance issues seen.

So, my goal in this podcast is to focus on belief couple of core beliefs, ones we have taken for granted, ones that we would never, ever question. To even think about questioning is sacrilegious and will get

most people upset even reviewing. The first involves foreign keys and the second looks at the myth behind why 3 tier is the best thing ever.

Foreign Keys

To make it clear (and I have had numerous discussions and debates on this point so I believe I need to exactly clarify what I am saying), I believe that the enabling of foreign key constraints within the database needs to be reviewed. My view is that they should be always disabled, and enabling them needs to be fully justified. In some cases they should be enabled, but in the vast majority they don't and just get in the way.

And now that everyone is in shock and awe at this heretical concept I'll try my best to justify why.

My top ten reasons for not using foreign key constraints:

#10 Relational might say we need foreign keys, but it doesn't say when it should be enforced. If you say immediately, then you must enforce them always and immediately meaning:

- You can never use deferred constraints
- Warehouse apps, bulk loading, data management are all compromised

#9 It's a hidden layer. Just like triggers, when they are enabled they add a hidden layer that can make it hard to diagnose and debug when problems occur. Let's say you add a foreign key to an existing application and problems occur inside the app. Is it because the foreign key has returned errors to the program in an area of code it was never expecting it? Same for tuning. When looking at a complex update statement, you have to look at the constraints around it as well as the actual SQL to tune it. Triggers are one those blessings and curses. They can solve a lot of problems because they are transparent and hidden, but they can cause a lot of problems because of this. If you are no aware they are there, tuning can become very difficult and goes into a multi-layered approach. In this case tuning isn't simple it's very complex, takes longer and is more frustrating to resolve.

#8 Don't you trust your developers? Are they there as a safeguard because developers can't write efficient code? And if developers think they are so great, just check to see if they use them in their development environment. I bet they are turned off because they get in the way of development. A key database feature disabled because it gets in the way? They mustn't think highly of it them. If the developers want it so much, enforce them to the letter in all environments. Then when the screaming stops, review their use.

#7 Performance problems. Inserts, updates, deletes, checking integrity, updating indexes. All to be done within the transaction. It can slow down transactions. In some cases, I begrudgingly accept that having the constraint there can help the optimizer for some types of queries (to me that is the exception rather than the rule).

#6 Are they value for money? If they never existed and then someone invented them, would we use them? I think not. I think it would be a hard sell to convince anyone they were needed. And then can we really trust them if we do have them? Look at deferred constraints. Just by allowing them it means we can create a constraint that might not be valid. It says, "I can live with possible invalid data, just enforce anything new".

#5 The whole notion of having data accuracy is the "golden fleece" we are all trying to chase. This is not correct. When you start looking at the concept of fuzzy logic you begin to realize that perfect data accuracy is not only a myth it can be inefficient trying to achieve it. Let's look at the humble date value. When we enter in a date, should we store the seconds, the sub seconds, the milli-seconds? How accurate do we need it? In theory it should be as accurate as the computer allows, because that is exact. When we store someone's name, do we enforce it to make sure it is spelt correctly? What about the street? Can we live with typing errors in comment fields made by users? If we want perfect data then we have to be as anal about keeping the data accurate here, as we are about keeping foreign keys accurate. There is no difference. The foreign key concept is one which we are locked into thinking as having to be perfect, whereas spell check errors in comment fields are ones that we can live with because it is what we are used to. My view is that spelling mistakes in a comment field is more dangerous than an orphaned record. There is a balance that needs to be achieved when enforcing data accuracy and it depends on the application. For a financial system, data accuracy might be seen as crucial, no mistakes allowed, so we should pull out all stops to maintain it, even at the expense of application management.

#4 Maintenance. Once foreign keys are enabled then the flexibility of the design of the application is weakened. Try using transportable tablespaces, oracle streams (data guard), replication, materialized views and RAC. With some blood, sweat and many tears you might get foreign keys working, but they will always be a point of failure, which will add additional costs to the maintenance of an application, and increase the length of time to do maintenance.

And on the issue of maintenance. Lets say a DBA has to do maintenance on app. They want to move data out from a table to another area. They delete the data first (as they can't do a truncate because of the constraints), and unknown to them a cascade fires. They then do their maintaining and re-insert the data. They have just caused major data failure and not even known it. To prevent this from happening, safeguards have to be put in place, but all DBAs are now living in a minefield, where the wrong move when doing legitimate maintenance can result in the destruction of the application.

Even moving a table between tablespaces (dropping it, even setting up partitioning) can be hampered by constraints.

#3 Those that want them are usually not the same group of people as those that have to manage them? Try doing database maintenance on a 300 table app with foreign keys. Try reorganizing a table, maintaining changes to it, even trying to move copies of data in it. You soon realize that you have to disable the keys. In some cases disabling the keys means dropping indexes or disabling the indexes. Which causes problems when they are re-enabled. Time. And do this at 3am when you only have a 2 hour window to do the work and the pressure is on. Then let's see how keen on foreign keys you are.

#2 Locking. Has anyone seriously looked at how expensive it is to manage a locking strategy with cascade deletes? And for a RAC environment, how inefficient are multiple locks, across multiple blocks in RAC? This is a complex and lengthy topic and I will not go into it further.

#1 They encourage relational thinking, when the database is object relational. I mean honestly, using object oriented structures in the database can be more efficient than using a traditional relational structure. We should be developing apps that use the best of both relational and objects (and use them that work well.) And that brings up another side issue, why are people so resistant to using the object oriented features inside the Oracle database? It's not a battle between relational and OO, it's not that one is better than the other, it's not one is right the other wrong, and one should not just be using relational because you don't understand or are not comfortable with the OO features. Ignorance is dangerous, especially from a performance perspective.

So that's my viewpoint on foreign keys. It's not pretty, it's very controversial and open to debate, a debate most people would think is not needed. But being a donut eating, arrogant, DBA, I do think it's needed.

3 Tier

On to another discussion point, and it's a real beauty. It's the notion of 3-Tier architectures being the natural way to go. When I see a 3-Tier application I see a legacy system. I see a system being cobbled together to try and get it to scale just a little bit more; I see a system being cobbled together to try and get some more grunt out of it. I don't see an efficient application. I see a very costly solution that is hard to maintain, hard to upgrade and horribly inefficient.

I have a strong belief that client/server or two tier is the correct way to go. But with a minor adjustment. Traditionally in two tier, the user interface and the application logic were stored on the client, and the data stored in the database. In the two tier what I see as the future, is that just the user interface will be stored on the client. The application logic and data is stored together on the database server. Additional tiers can spring from the database – allowing database to database communication like web services, but essentially it's the client talking to the server.

The worst thing to do is to put onto a separate machine the application logic. Yes separating the data from the logic is good, but putting them on separate nodes is not good. This just leads to inefficiencies galore. Starting with the network performance. Application logic requires data to be delivered to the client. The more dynamic the nature of the logic, the more data it needs to query. The application and the data communicate, chat, the more vibrant the chat the better the app. Its easier to chat with someone when they are next to you and not in the next room. The act of querying the database, sending it SQL statements is inherently slow. The SQL statement has to be wrapped up in a network call, passed down numerous layers and sent to the server, which has to unwrap it, then parse it, then run it, then return the data back again. It all takes time, and it adds up as more and more users access the system. It's called scalability. The other notion which to me is very strange is that it is a wise thing to completely separate the application logic from the data. That way you can swap out the database or change the application tool. The same people who follow this principal I dare say, also subscribe to the one that says

it is good to keep the DBAs separate from the Developers (This view is exposed in Podcast #1). It is not - it's just bad practice. It is inefficient and like the use of object oriented programming, it lends itself to an architecture that doesn't scale and will experience performance issues.

The advantage of using PL/SQL (and Java) is that the application logic is held tightly together inside the database, allowing the two structures to scale together. An application that is designed and built for this architecture will naturally scale and scale well (and has been proven by Apps written). This is based on experience.

Conclusion

The aim of this podcast is to highlight that even core cherished notions, ones we take for granted, ones we would never dream of questioning, should be reviewed on a regular basis to see if they are still valid in the current environment. Technology is changing, software is changing – the rules are changing. If after a serious review there is justification for keeping them, then great, leave them be, but if the justification is “we know no better, we don't understand why it's there, or that's the way it has always been”, then there is a good reason for reviewing and possibly removing or replacing it. This is not just limited to technology features but should extend to the culture of an organization, its attitudes, procedures and methodologies.

Question it, review why it's there and if it cannot be rationally justified then remove it. Remove the red tape, do the spring clean and simplify the structure of the organization. In doing so we will create a simpler, more efficient work place.