Acht große Oracle-Datenbank-Mythen

Robert Barić, ITGAIN Consulting Gesellschaft für IT-Beratung mbH

Die IT steckt voller Mythen. Man denke nur an die sich in den Köpfen von Anwendern noch immer haltende Notwendigkeit, Datenträger regelmäßig defragmentieren zu müssen. Mit einer Solid State Disk ist dies überflüssig. Mancher Anwender glaubt auch noch, er surfe im „Privat Modus" wirklich anonym, was definitiv nicht der Fall ist.

Technische Mythen basieren oft auf falschen Annahmen oder auf einem Wissensstand, der inzwischen durch weitere Entwicklungen überholt ist. Auch die Welt von Oracle steckt voller solcher Mythen. Höchste Zeit, sich einmal mit den überraschendsten eingehender zu beschäftigen.

Mythos 1: Count(*) ist böse

Für lebhafte Fachgespräche sorgt immer wieder die Frage, ob „count(*)", „count(1)" oder „count(rowid)" performanter sind, wenn es darum geht, die Anzahl der Zeilen in einer Tabelle zu bestimmen. Die beiden Letztgenannten gelten bei einigen geradezu als Geheimtipp. Ein Mythos! Das Ergebnis und der Zugriffspfad sind bei allen drei Abfragen absolut identisch. Die Annahme, dass der Stern dabei alle Spalten berücksichtigt, ist schlicht ein Irrglaube.

Doch was passiert, wenn die Funktion „count" über eine Spalte ausgeführt wird? Im Fokus stehen jetzt die Inhalte der jeweiligen Spalte und nicht die Zeilen. In diesem Fall werden nur die Zeilen gezählt, in denen die Werte der Spalte bekannt (also „not null") sind. Unbekannte Werte („null") werden nicht berücksichtig. Dazu ein konkretes Beispiel, für das man eine neue Tabelle erstellt *(siehe Listing 1)*.

Jetzt erfolgt die Abfrage der Zeilen mit vier Varianten des Count-Befehls:

1. Select count(*) from mytable;
2. Select count(1) from mytable;
3. Select count(rowid) from mytable;
4. Select count(x2) from mytable;

Tabelle 1 zeigt, welches Ergebnis die verschiedenen Abfragen liefern. Das vielleicht auf den ersten Blick überraschende Ergebnis der vierten Variante mit „count(x2)" erklärt sich dadurch, dass in diesem Fall keine Null-Werte betrachtet werden und nun dieser Index, der keine unbestimmten Werte enthält, genutzt werden darf *(siehe „https://docs.oracle.com/database/121/SQLRF/functions003.htm#SQLRF20035")*.

Ändern sich jedoch auch die Ausführungspläne, wenn in der Spalte „x2" keine Null-Werte zugelassen werden? Die Antwort lautet schlicht „ja". Wenn die Spalte „x2" mit einem „not null"-Constraint versehen ist und damit nur mit bestimmten Werten gefüllt werden kann, greifen alle vier Abfragen auf den Index zurück und liefern auch das gleiche Ergebnis. Mit „sqlplus" und „autotrace" lässt sich diese Aussage einfach verifizieren.

Variante	Select	Operation	Ergebnis an Zeilen
1	Count(*)	Full Table Scan	300.000
2	Count(1)	Full Table Scan	300.000
3	Count(rowid)	Full Table Scan	300.000
4	Count(x2)	Index Fast Full Scan	270.000

Tabelle 1

```
Create table mytable (x1 number, x2 number, x3 date)
Create index ind2 on mytable(x2)
Insert von 300.000 Zeilen mit beliebigen Werten.
```

Listing 1

```
create index i3 on mytable (x2,'1');
create index i3 on mytable (x2,1);
```

Listing 2

Mythos 2: Indizes speichern keine Null-Werte

In einigen Fundstellen der Literatur existiert immer noch die Aussage, dass Null-Werte nicht in Indizes gespeichert werden *(siehe „https://hoopercharles.wordpress.com/2012/02/28/repeat-after-me-null-values-are-not-stored-in-indexes" und „http://www.toadworld.com/platforms/oracle/w/wiki/5874.indexes-and-null.aspx")*. Ein Mythos!

Null-Werte werden durchaus im B-Tree-Index gespeichert, sofern zumindest eine Index-Spalte keinen Null-Wert besitzt. Der Mythos hält sich wahrscheinlich nur deshalb, weil ein Index mit nur einer Spalte nie einen Null-Wert aufnimmt. Nachfolgend zwei Kommandos, bei denen ein Index Null-Werte enthalten kann. Das Kommando „create index i1 on mytable (x1,x2);" erstellt einen zusammengesetzten Index, der Zeilen der Tabelle

referenziert, sofern nicht „x1“ und „x2“ gleichzeitig unbestimmt sind. In diesem Fall erfolgt kein Eintrag im Index.

Eine der beiden Varianten des Kommandos erzeugt einen zusammengesetzten Index mit einem Literal *(siehe Listing 2)*. Da das Literal ein bestimmter Wert ist, wird jeder Wert in „x2“, egal ob bestimmt oder unbestimmt, im Index aufgenommen.

Listing 3 zeigt, wie die Ausführungspläne für die oben erwähnten Abfragen mit „count“ aussehen, wenn jeweils ein solcher Index für die im eingangs gezeigten Beispiel angelegte Tabelle „mytable“ ohne „not null“-Constraints erstellt wird.

Die Unterschiede bei den Kosten sind deutlich. Das kommt daher, dass bei der ersten Variante in beiden Spalten unbestimmte Werte vorkommen können und die Datenbank einen Full Table Scan verwendet, um alle Zeilen zu zählen. Bei der zweiten Variante mit einem Literal werden alle Zeilen im Index gespeichert und die Datenbank nutzt diesen, um alle Zeilen zu zählen.

```
create index i1 on mytable (x1,x2);
select count(*) from mytable; -> Resultat ist Full Table Scan

Execution Plan
----------------------------------------------------------
| Id  | Operation           | Name     | Rows  | Cost (%CPU)|
----------------------------------------------------------
|  0  | SELECT STATEMENT    |          |   1   |   249 (1)    |
|  1  |  SORT AGGREGATE     |          |   1   |              |
|  2  |   TABLE ACCESS FULL | MYTABLE  | 299K  | 249 (1)      |
----------------------------------------------------------

create index i3 on mytable (x2,1);
select count(*) from mytable; -> Resultat ist Fast Full Index Scan

Execution Plan
----------------------------------------------------------
| Id  | Operation               | Name  | Rows  | Cost (%CPU)  |
----------------------------------------------------------
|  0  | SELECT STATEMENT        |       |   1   |   208 (1)    |
|  1  | SORT AGGREGATE          |       |   1   |              |
|  2  | INDEX FAST FULL SCAN    | IND1  | 210K  |   208 (1)    |
----------------------------------------------------------
```

Listing 3

Mythos 3: Deadlocks sind tödlich

Geraten zwei Sessions in der Oracle-Welt in eine Deadlock-Situation, endet dieser Zustand dadurch, dass eine der beiden Sessions terminiert wird und der Konflikt damit gelöst ist. Dieser automatische Killer-Auftrag ist ein weiterer Mythos, wie Charles Hooper aufgezeigt hat *(siehe „http://hooper-charles.wordpress.com/2012/01/04/faulty-quotes-7-deadlock-kills-sessions“)*. *Tabelle 2* zeigt das Beispiel.

Session 1	Session 2	Kommentar
Create table T1(C1 Number Primary Key, C2 Varchar2(10));		
Insert Into T1 values (1,’1’);	Insert Into T1 values (2,’2’);	
	Insert Into T1 values (1,’3’);	Session 2 hängt
Insert Into T1 values (2,’2’);		Session 1 hängt
	ORA-00060 deadlock detected	Sesson1 hängt weiter, Session 2 ist frei

Tabelle 2

Das möglicherweise überraschende Ergebnis: Weder Session 1 noch Session 2 werden terminiert. Und keine von beiden wird vollständig zurückgerollt. Lediglich die letzte Deadlock-Aktion wird nicht ausgeführt. Fragt man die Tabelle „T1“ in Session 2 nach dem Deadlock ab, erhält man folgendes Ergebnis *(siehe Tabelle 3)*.

Session 1	Session 2	
…	…	
	ORA-00060 deadlock detected	
	SELECT * FROM T1;	
	C1	C2
	--------	-------
	2	2

Tabelle 3

Mythos 4: If Exit = Rollback

Eine der vier sogenannten „ACID-Eigenschaften“ einer Datenbank besagt, dass für den Fall, dass eine Transaktion nicht ordentlich abgeschlossen werden kann, diese zurückgerollt wird. Ein Verhalten, das immer beobachtet werden sollte. Das sollte ja auch bei einer Oracle-Applikation gelten, wenn diese vorzeitig geschlossen oder absichtlich abgeschossen wird. *Listing 4* zeigt das Beispiel eines Mythos.

Das Ergebnis: Die Insert-Operation wurde persistent übernommen, obwohl „autocommit“ abgeschaltet ist. Dabei handelt es sich nicht um einen Fehler in der Datenbank, sondern um ein Standardverhalten vieler Oracle-Tools. Beim Verlassen der Applikation werden die Änderungen festgeschrieben. Dahinter steht wohl die Annahme, dass bei vielen Oracle-Tools das Beenden der Anwendung als ein vom Benutzer gewollt abschließendes Ereignis gesehen wird und deswegen kein explizites Commit erforderlich ist *(siehe „http://docs.oracle.com/cd/E29505_01/server.1111/e25789/transact.htm“)*.

Mythos 5: Weniger ist mehr

Ein klassischer Ansatz, eine Abfrage zu optimieren, besteht darin, die Abfragemen-

ge zu beschränken. Das lernen heute Kinder bereits in der Grundschule, wenn es um das Thema „Suchmaschinen" geht. Die Oracle-Welt kennt solche Beschränkungen ebenfalls, damit etwa ein Index verwendet werden kann. Aber können zu viele Bedingungen in der „Where"-Klausel zu einer Verlangsamung der Abfragen führen? Dazu ein Beispiel von Marcus Winand *(siehe „https://use-the-index-luke.com/3-minute-test")*. Dabei werden zwei Abfragen angelegt, wobei die zweite durch eine zusätzliche Einschränkung noch präziser ist *(siehe Listing 5)*. Die Abfrage liefert 100 aus 1.000.000 Zeilen.

Die Abfrage aus *Listing 6* liefert 10 aus 1.000.000 Zeilen. Inwiefern hat aber die weitere Einschränkung Auswirkungen auf Kosten beziehungsweise Performance? Läuft die Abfrage schneller, langsamer oder macht das keinen Unterschied? Oder hängt das Ergebnis von den Daten selbst ab?

Die eigene Intuition dürfte den meisten sagen, dass eine größere Einschränkung positiv ist. Aber ist die Performance nicht vielleicht doch auch eher von den Daten abhängig? Das überraschende Ergebnis: Die weitere Bedingung in diesem Beispiel verschlechtert die Performance. Es ist also ein Mythos zu glauben, dass mit zusätzlichen Einschränkungen in den Abfragen die Ergebnisse grundsätzlich schneller oder kostengünstiger vorliegen.

In diesem Beispiel muss durch die zusätzliche Einschränkung auch auf den Index der Tabelle zugegriffen werden. Der Aufwand durch den Tabellen-Zugriff steigt erheblich, wie sich durch einen Vergleich der beiden Ausführungspläne einfach zeigen lässt *(siehe Listing 7)*.

Mythos 6: Der veraltete Index

Ob es etwas damit zu tun hat, dass viele Anwender ein personalisiertes Verhältnis mit Computern pflegen? Der Gedanke, dass Software und Programme wie wir Menschen altern, ist nicht nur im Umgang mit Windows, Word & Co. anzutreffen. Auch manche Administratoren machen sich immer wieder Gedanken über das Alter einer Datenbank. So sollen Indizes mit der Zeit altern und aus der Balance geraten. Wächst der Index stärker auf einer Seite in einem bestimmten Werte-Bereich, vergrößert sich auf dieser Seite auch die Tiefe. Unterschiedlich tiefe Baum-Strukturen erhöhen aber den Verwaltungsaufwand.

Ein weiterer Faktor, der den Alterungsprozess des Index in die Wege leitet, besteht darin, dass gelöschte Elemente im Index verbleiben und nur als gelöscht markiert werden. Anders als beim Menschen gibt es hier zumindest Linderung dieser typischen Altersbeschwerden. Der Index-Rebuild sorgt für Heilung *(siehe Abbildung 1)*.

Die Sache mit der mangelnden Balance ist ein Mythos. Er gründet auf einem Missverständnis. B-Tree steht bei Oracle nicht für „Binary", wie erstaunlich viele Nutzer fälschlicherweise annehmen, sondern für „Balanced" *(siehe „http://docs.oracle.com/cd/E11882_01/server.112/e40540/indexiot.htm#CNCPT1170")*. Ein nicht ausbalancierter balancierter Baum? Das klingt geradezu nach einem Widerspruch in sich. Ein einfacher Algorithmus sorgt dafür, dass es keine unbalancierten Bäume geben kann.

Die Erklärung: Wenn neue Einträge einen Index-Block vollständig auffüllen, wird der Block geteilt. Es entstehen zwei neue Blöcke und damit ein Split. Allerdings entsteht nur beim Split des ersten Blocks (Root) eine neue Ebene. Alle anderen Splits geschehen auf dem gleichen Level *(siehe Abbildung 2)*.

Beim Verlauf des Vorgangs wird deutlich, dass bei diesem Verfahren nur balancierte Bäume entstehen können. Auf allen drei Ebenen werden Splits bei den Blöcken „R", „A" und „4" ausgeführt. Nur bei „R" entsteht eine neue Ebene und damit eine größere Tiefe des Baumes *(siehe Abbildung 3)*.

Doch auch die Alterung und der Verlust der Performance durch nicht mehr gebrauchte Index-Einträge ist ein Mythos in der Oracle-Welt. Der Gedanke basiert auf einer oder mehreren der nachfolgenden, falschen Annahmen:

- Entleerte Index-Blöcke können nicht wiederverendet werden
- Gelöschte Index-Einträge werden nie wiederverwendet
- Ein Index-Eintrag kann nur wiederverwendet werden, wenn ein neuer Eintrag in eine Lücke passt

```
SQL> set autocommit OFF
SQL> show autocommit
autocommit OFF
SQL> create table TEXIT (a number);
SQL> Insert into TEXIT values (1) ;
SQL> Exit;

#> SQLPLUS <User/Password>
SQL> select * from texit;

A
--
1
```

Listing 4

```
create index tab_idx on tb1 (a, date_column);
Select date_column, count(*)
From tb1
Where a=123
Group by date_column;
```

Listing 5

```
Select date_column, count(*)
From   tb1
Where  a = 123
And    b = 42
Group by date_column;
```

Listing 6

- Index-Blöcke können nur wiederverwendet werden, wenn neue Einträge in die erstellte Lücke passen
- Es ist ein (regelmäßiger) Index-Rebuild notwendig, um den Index performant zu halten

Tatsächlich werden Einträge wiederverendet und sie müssen auch nicht genau in die zuvor ausgefüllte Lücke passen *(siehe „https://richardfoote.files.wordpress.com/2007/12/index-internals-rebuilding-the-truth.pdf")*.

Mythos 7: Ein Index-Rebuild tut immer gut

Treffen Administratoren und Entwickler aufeinander, ist die Diskussion rund um den Rebuild nicht fern. Schaden wird der Rebuild ja wohl wenigstens nicht. Das ist aber gerade nicht korrekt. Denn ein Index-Rebuild versucht Kosten:

- Der gesamte Index muss gelesen und neu geschrieben werden
- Kurzfristig wird bis zum Doppelten des Speicherplatzes benötigt; dabei kann der Tablespace anwachsen, wenn die Funktion „Autoextend" aktiviert wurde
- Änderungen am Index generieren entsprechend Redo
- Beim Offline-Rebuild wird die Tabelle gesperrt
- Insert-Operationen benötigen Platz; ein Rebuild hingegen macht den Index in der Regel kompakter, wodurch weniger Platz zur Verfügung steht. Steht davon nicht genügend zur Verfügung, muss er erst durch neue Index-Block-Splits geschaffen werden, die wiederum Redo und zusätzliches I/O verursachen. Und genau das führt zu längeren Laufzeiten.

Mythos 8: Der Index-Rebuild ändert den Clustering-Faktor

Der Clustering-Faktor ist eine wesentliche Kennzahl für die Entscheidung des Optimizers, ob ein Index-Zugriff oder besser ein Full-Table-Scan angewendet werden sollte. Der Clustering-Faktor ist ein wesentlicher Faktor für die Kosten beim Index-Zugriff. Mit einem Index-Rebuild wird der Clustering-Faktor optimiert – oder doch nicht? Da der Clustering-Faktor ja zum Index gehört, muss sich dieser ja verändern, wenn der Index optimiert wird.

```
select date_column,count(*)
from tb1
where a=123
group by date_column;

-------------------------------------------------------------------------
| Id  | Operation            | Name     | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------
|  0  |   SELECT STATEMENT   |          |   100 |   2200 | 3  (0)|
|  1  |   SORT GROUP BY NOSORT |          |   100 |   2200 | 3  (0)|
|* 2  |   INDEX RANGE SCAN   |TAB_IDX   |   100 |   2200 | 3  (0)|
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("A"=123)

select date_column,count(*)
from tb1
where a=123
and b=42
group by date_column;

-------------------------------------------------------------------------
--
| Id  | Operation                     | Name    | Rows | Bytes |Cost (%CPU)|
-------------------------------------------------------------------------
--
|  0  | SELECT STATEMENT              |         | 126  | 4410 | 28  (0)|
|  1  | SORT GROUP BY NOSORT          |         | 126  | 4410 | 28  (0)|
|* 2  | TABLE ACCESS BY INDEX ROWID|TB1 |         | 126  | 4410 | 28  (0)|
|* 3  | INDEX RANGE SCAN              |TAB_IDX |  30  |      |  3  (0)|
-------------------------------------------------------------------------
--

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("B"=42)
   3 - access("A"=123)
```

Listing 7

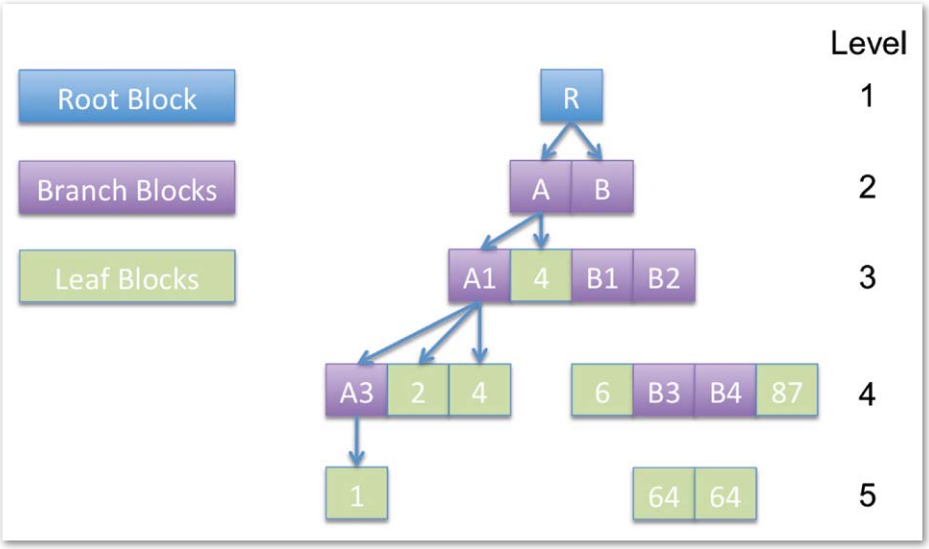

Abbildung 1: Ein unbalancierter Baum mit Blättern sind auf unterschiedlichen Ebenen (Level)

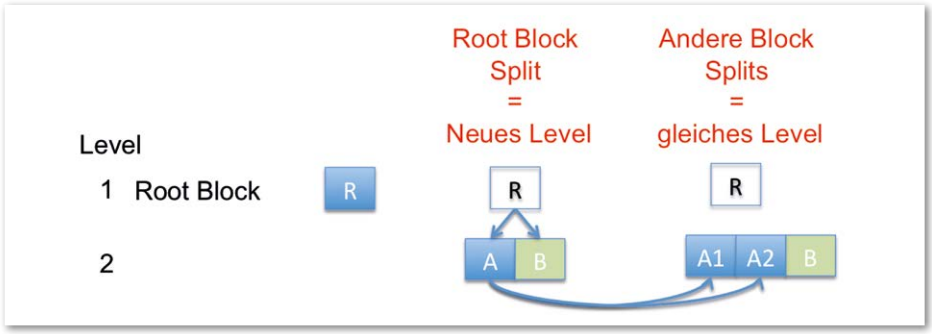

Abbildung 2: Der Split erfolgt nur einmal auf einer neuen Ebene

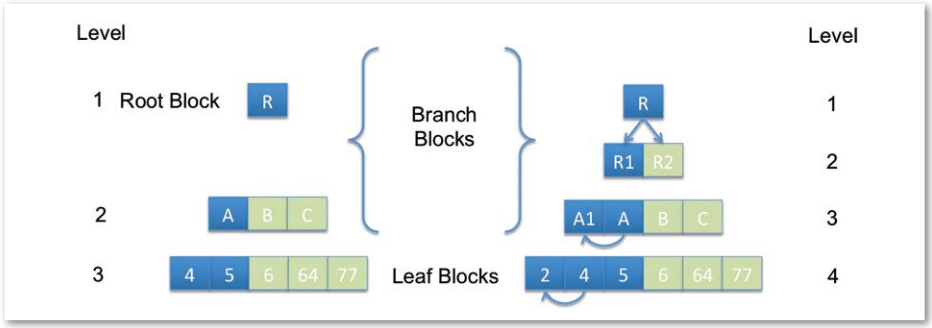

Abbildung 3: Nur bei „R" entsteht eine neue Ebene

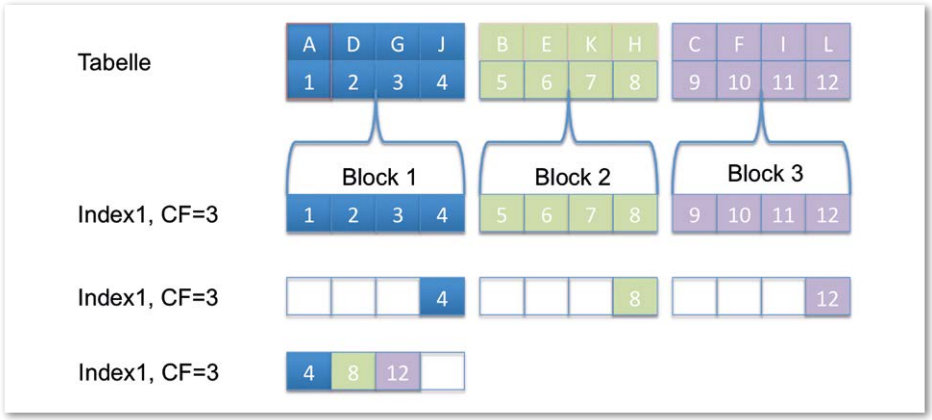

Abbildung 4: Der Clustering-Faktor

Dies ist ein Mythos, der sich aber auch aus alten Support-Dokumenten von Oracle speist *(siehe „Support Doc ID 122008.1")*. Denn dort wurde der Index-Rebuild tatsächlich als Optimierungsmaßnahme für den Clustering-Faktor beschrieben. Das ist indes schon lange nicht mehr so.

Der Clustering-Faktor beschreibt nichts anderes als die Ordnung zwischen der Tabelle und dem Index.

Dieser ist jedoch immer geordnet. Der Index ändert seine Ordnung weder vor noch nach oder zwischen einem Rebuild. Das bedeutet dann allerdings auch, dass sich der Clustering-Faktor nicht ändern kann.

Zur Verdeutlichung dient das Schaubild in *Abbildung 4*, anhand dessen die grundlegende Berechnung des Clustering-Faktors aufgezeigt wird. Oben senkrecht sind die Zeileninhalte einer Tabelle, wie (A,1), dargestellt.

Die Blöcke einer Tabelle wurden farbig unterschiedlich markiert. Die Inhalte des Index1 sind mit der korrespondierenden Blockfarbe der jeweiligen Tabelle eingefärbt.

Vereinfacht: Der Clustering-Faktor wird durch den Wechsel der Blöcke der Tabelle bestimmt. Werden nun alle Zeilen außer denen mit den Werten „4" ,8" und „12" entfernt, ändert sich der Clustering-Faktor weder dabei noch nach einem Rebuild, da weiterhin der Blockwechsel bestehen bleibt.

Schluss mit den Mythen!

Diese kleine Auswahl an Mythen aus dem Oracle-Umfeld ist nicht neu, sondern gut dokumentiert. Dennoch spuken sie in den Köpfen vieler Anwender herum. Es gibt zwei Sätze, die wie Saatkörner zur Entstehung eines Mythos beitragen: „Das haben wir immer schon so gemacht" und „Das ist technisch nicht möglich".

Warum wurde etwas schon immer so gemacht? Warum wird es weiterhin so gemacht? Warum kann etwas technisch nicht sein?

Fazit

Die Entstehung von Mythen kann nur verhindert werden, wenn man das (eigene) Wissen infrage stellt. Nur weil einmal etwas in einer offiziellen Dokumentation geschrieben wurde, muss das nicht den Tatsachen entsprechen oder entsprochen haben. In diesem Sinne: Neugierig bleiben. Infrage stellen. Experimentieren.

Robert Barić
robert.baric@itgain.de