Compression

Dr. Günter Unbescheid
Database Consult GmbH
Jachenau

Schlüsselworte

Datenbank, Performance, Compression

Abstract

Die Komprimierung von Tabellen und Indizes in Oracle Datenbanken ist nicht neu. Bereits in der Version 9 eingeführt, wurde das Feature beständig – bis hin zur aktuellen Version 12c – ausgebaut. Mittlerweile kann die Komprimierung nicht nur für *direct load* Operationen und im OLTP-Umfeld genutzt werden, sondern steht in der Version 12c für *Automatic Data Optimization* (ADO) Techniken bereit. Mit *Hybrid Columnar Compression* seht darüber hinaus eine Technik zur Verfügung, die in Abhängigkeit vom Storage System, mit neuen Speicherstrukturen aufwartet, die verbesserte Komprimierungsraten erlauben. Schließlich lassen sich auch Data Guard Redo-Daten, Datapump-Dumps, RMAN-Backups und TNS-Pakete komprimiert werden.

Die in der Praxis erzielbaren Kompressionsraten hängen dabei nicht nur von den unterstützten Algorithmen, sondern ganz stark auch von dem Charakter und der Sortierung der zu komprimierenden Daten ab. Alle Vorgaben müssen deshalb vor dem Hintergrund des eigenen Datenbestandes getestet werden.

Kontext und Motivation

Aus den schlichten Anfängen mit dem Release 2 der Version 9 ist mittlerweile eine reichhaltiges, technisches Portfolio entstanden, das viel Raum für kluge Design-Entscheidungen bietet und diese auch erfordert. Nach wie vor liegen die Pro's und Con's von *Compression* – ganz gleich in welchem Kontext die Techniken eingesetzt werden – klar auf der Hand und bestimmen die Motive für den Einsatz. Sie seien eingangs kurz aufgezählt:

- Offensichtlich ist die Reduktion von Speicherplatz bzw. Datenvolumen. Je nach Datentyp, Datencharakteristik, DML-Charakteristik der zugehörigen Applikationen und der eingesetzten Algorithmen, finden wir hier jedoch eine sehr breite Streuung von erzielten Kompressionsraten.
- Indirekt: Die Reduktion von Speicherplatz kann Blockzugriffe oder Übertragungsraten minimieren helfen und damit die Performance von Operationen steigern und Caches entlasten.
- Indirekt: Die Komprimierungs- und De-Komprimierungsoperationen fordern CPU-Ressourcen. Diese werden aber in manchen Fällen durch die reduzierten Block-Allokationen wieder wettgemacht.
- Für manche, aber nicht alle Komprimierungstechniken fallen zusätzliche Lizenzkosten an.

Für alle Anwendungen mit ihren Datenzugriffen ist *Compression* transparent. Mit aktuellem Releasestand, d.h. RDBMS Version 12.1.0.1 kann *Compression* in folgenden Kontexten eingesetzt werden. Die wichtigsten, aber nicht alle der folgenden Kontexte werden im Anschluss detailliert besprochen:

- Basic Table Compression für *direct load* und CTAS-Operationen bei Tabellen und Partitionen
- OLTP Table Compression für konventionelle insert- und update-Operationen
- Index Compression für B-Tree Indizes
- Hybid Columnar Compression (HCC) für direct load, update und insert-Operationen in Abhängigkeit vom Storage Subsystem.
- Automatic Data Optimization (ADO) automatisiert unter 12c die Komprimierung auf der Grundlage von Policies.
- LOBs lassen sich komprimieren, wenn sie als Secure Files gespeichert werden
- Advanced Network Compression komprimiert die Netzpakete
- RMAN komprimiert Backupsets
- Datapump kann sowohl Metadaten als auch Tabellendaten komprimieren
- Data Guard kann die Redo-Daten zum Transfer auf die Snapshot Site komprimieren
- Flashback Data Archive History Tabellen lassen sich ebenfalls komprimieren

Generell gilt: Die Komprimierung von Segmentdaten ist im Umfeld verschlüsselter Tablespaces in den meisten Fällen nahezu wirkungslos, es sei denn die Komprimierung wird – explizit – vor der transparenten Verschlüsselung durchgeführt.

Vorgehen

Bei der Recherche und beim Testen im Umfeld unseres Themas *Compression* können wir uns nicht allein und nicht immer auf die Metadaten des Data Dictionary und den dort angezeigten Speicherverbrauch verlassen. In einigen Fällen – besonders wenn es um die detaillierte Beurteilung von Table- und Index-Compression geht – müssen wir die Metadaten des DD durch Daten aus logischen Blockdumps ergänzen. Diese Dumps lassen sich – immer noch – über folgende Syntax leicht generieren:

```
alter system dump datafile <fno> block <bno>;
```

Zur leichtere Identifizierung der im `user_dump_dest` Verzeichnis gespeicherten Trace-Dateien können wir ein Erkennungszeichen in den Dateinamen einschleusen (`tracefile_identifier`). Wenn ein Dump direkt von der Platte (d.h. nicht aus dem Cache) gewünscht ist, sollten wir vorher den Buffer Cache leeren (`alter system flush buffer_cache`) oder – etwas „umweltverträglicher“ – einen expliziten *checkpoint* setzen (`alter system checkpoint`).

Die für die betreffende Session gültige Trace-Datei kann in SQL*Plus über den Befehl `oradebug` angezeigt werden:

```
oradebug SETMYPID;
oradebug TRACEFILE_NAME;
```

Basic Table Compression

Hinter dem Terminus “Basic Table Compression“ verbirgt sich die „Komprimierung“ von Tabellendaten, die im *bulk load* Verfahren eingefügt wurden, d.h. die per `insert /*+ append */` oder CTAS geladen wurden. Tabellen dieser Art werden über die Klausel `compress basic` oder nur `compress`.

Diese vermeintliche Komprimierung ist in Wahrheit eine De-Duplizierung der Daten, die auf Blockebene sogenannte *symbol tables* (ST) einführt, welche dann aus den „wirklichen“ Rows heraus referenziert werden. Dabei können ST auch „geclustered“ werden, so dass in jedem Block ST-Hierarchien entstehen können. Auf diese Weise kann ein Eintrag neben einem Spaltenwert Verweise auf andere

symbols enthalten, die wiederum Werte einer oder mehrerer Spalten enthalten können. Die definierte Reihenfolge der Spalten (*column_id*) kann dabei gegenüber den Metadaten vertauscht werden.

Die folgende Abbildung zeigt einen Ausschnitt aus einem logischen Block-Dump, der nachträglich entsprechend kommentiert wurde (kursiv, fett, unterstrichen). Folgende Row einer Tabelle namens `t2comp` ist hier dargestellt:

```
col1='2AAAAA' , col2='2ZZZZZZZZZ', col3='XX1XXXXXXX'
```

Die kursiv gedruckten Teile stellen erläuternde Ergänzungen des Dumps dar.

```
tab 1, row 1, @0x1f1a (zweite "wirkliche" Row)
tl: 16 fb: --H-FL-- lb: 0x0  cc: 3
col  0: [10]  58 58 31 58 58 58 58 58 58 58 (XX1XXXXXXX)
col  1: [ 6]  32 41 41 41 41 41 (2AAAAA)
col  2: [10]  32 5a 5a 5a 5a 5a 5a 5a 5a 5a (2ZZZZZZZZZ)
bindmp: 2c 00 02 03 00 c9 32 5a 5a 5a 5a 5a 5a 5a 5a 5a
```

Der Eintrag *bindmp* versweist u.a. auf die *symbol table* , die als eigene „Tabelle“ in dem Block gespeichert ist – hier markiert als `tab 0 row` 0:

```
tab 0, row 0, @0x1f45
tl: 11 fb: --H-FL-- lb: 0x0  cc: 2
col  0: [10]  58 58 31 58 58 58 58 58 58 58 (XX1XXXXXXX)
col  1: [ 6]  32 41 41 41 41 41 (2AAAAA)
bindmp: 00 6e 02 05 ce 32 41 41 41 41 41
```

Dieser Eintrag repräsentiert seinerseits 2 Spalten (cc: 2), verweist dabei aber – über seinen bindmp Eintrag – auf row 5 der *symbol table*, der schließlich den „physischen“ Wert „`XX1XXXXXXX`“ enthält:

```
tab 0, row 5, @0x1f66
tl: 13 fb: --H-FL-- lb: 0x0  cc: 1
col  0: [10]  58 58 31 58 58 58 58 58 58 58
bindmp: 00 02 d2 58 58 31 58 58 58 58 58 58
```

Ist die Tabelle als *compress basic* angelegt und regelkonform gefüllt worden, kommt es jedoch nur dann zu einer De-Duplizierung der Daten, wenn sowohl

- redundante Datenbereiche innerhalb des betreffenden Blockes vorliegen, als auch
- eine Mindestfüllung des Blockes gewährleistet ist.

Beim Vorliegen des ersten Kriteriums wird die De-Duplizierung mit ansteigendem Füllgrad des Blockes keineswegs automatisch nachgefahren, da Append-Operationen bekanntlich hinter der HWM-Marke aktiv werden und konventionelle Inserts (s.u.) in diesem Modell nicht komprimiert werden.

Letzte Klarheit über Zustand der Komprimierung kann daher nur erlangt werden durch:

- einen logischen Dump
- oder – falls eine identische Tabelle ohne Komprimierung vorliegt – über die Ermittlung des Füllgrades der Blöcke (Einsatz von Package `dbms_rowid`, kombiniert mit `count(*)` gruppiert nach Blocknummer)

Bei diesem Vergleich ist zusätzlich zu beachten, dass komprimierte Tabellen per Default mit einem Freiplatz (*pctfree*) von 0 angelegt werden, und nicht mit den für sonstige Tabellen üblichen 10%.

Werden nun in einer Tabelle mit *basic compression* Rows verändert oder nachträglich konventionell eingefügt, findet für diese Rows keine De-Duplizierung des Inhaltes statt und zwar auch dann nicht, wenn neue, veränderte Inhalte auf bereits vorhanden *symbol table entries* verweisen könnten. Auf

diese Weise können – von außen betrachtet – auch verkürzende Veränderungen zu einem gestiegenen Speicherbedarf führen. Darüber hinaus ist bei Veränderungen die Wahrscheinlichkeit von *row migrations* auf Grund des pctfree-Wertes von 0 sehr hoch – es sein denn, dieser Werte wurde nachträglich und explizit angepasst.

Ebenso ist es von entscheidender Bedeutung, in welcher Sortierreihenfolge die Daten geladen werden. Hier empfiehlt sich eine Sortierung, welche die am wenigsten selektiven Spalten bevorzugt, weil sie die meisten Wiederholgruppen aufweisen. Die Wahrscheinlich von Cluster-Bildungen ist auch abhängig von der gewählten Blockgröße, mit steigender Blockgröße kann möglicherweise die Komprimierungsrate gesteigert werden, wenn durch veränderte Blockgrenzen „Werteinseln" in nachfolgenden Blöcken vermieden werden können.

Advanced Table Compression

Beginnend mit der Version 11g können auch konventionelle OLTP-Operationen für die Komprimierung herangezogen werden, wenn die betreffenden Tabellen über die Klausel `compress for oltp` (11g) oder `row store compress advanced` (12c) angelegt wurden. Tabellen dieser Art werden per Default mit einem auch sost üblichen pctfree-Wert von 10 angelegt, enthalten daher ein wenig Platz für geringfügige Row-Updates.

Ein Blick auf das Speicherformat mit Hilfe von Block-Dumps zeigt, dass hier offensichtlich dieselben Algorithmen und Formate wie bei der *Basic Compression* zum Einsatz kommen, folglich – rein theoretisch(!) – die gleichen „Kompressionsraten" durch De-Duplizierung erzielt werden könnten. Ein detaillierter Blick auf die Datenblöcke und ihren „*Lifecycle*" zeigt jedoch, dass dies in der Regel nicht der Fall ist.

Die Oracle Dokumentation erwähnt den für die Kompression eingesetzten Batch-Modus, der in Aktion tritt, wenn „*data in the block reaches an internally controlled threshold*". Dies ist insofern sinnvoll,, als damit Kompressionsoperationen konzentriert im „Batch-Modus" und asynchron ablaufen können. Es ist jedoch auf der anderen Seite sehr irreführend formuliert, wie sich durch kontrollierte insert- und update-Aktionen mit zwischengeschalteten Block-Dumps leicht verifizieren lässt. Diese Tests ergeben sowohl unter Version 11g als auch 12c folgendes Bild:
Kompressionsoperationen werden nur durch insert-Operationen ausgelöst und auch nur dann, wenn der Block bis zur pctfree-Grenze gefüllt wird. Vorher bleiben die Rows in ihrem nicht komprimierten Zustand. Desgleichen kann kein update die Kompression anstoßen, auch dann nicht, wenn durch die Veränderung die betreffende Row migriert werden muss! Ob die migrierte Row in ihrem neuen Block eine Komprimierung anstößt, hängt ganz von dessen Füllgrad ab. Inserts, die zufälligerweise nach migrierenden Updates durchgeführt werden, landen selbstredend in dem neu eröffneten Block, und werden dort nur dann eine Komprimierung auslösen, wenn – wie bereits gesagt – die pctfree-Grenze erreicht wurde.

Für den praktischen Umgang mit der *Advanced Table Compression* lassen sich daher folgende Besonderheiten festhalten:

- Liegt eine „gesunde" DML-Charakteristik vor, bei der sich delete-, update- und insert-Operationen – mehr oder weniger – ausgleichen, kann es zu einer nachhaltig wirksamen De-Duplizierung kommen, die jedoch darüber hinaus die für den OLTP-Betrieb grundsätzlich gültigen, typischen Merkmale (s.u.) berücksichtigen muss.

- Ist diese DML-Balance nicht gegeben, wird sowohl der Komprimierungsgrad darunter leiden als auch das Row-Chaining/Row-Migration in die Höhe schießen.
- Falls die Komprimierung zufriedenstellend läuft – und daher die Anzahl der Rows pro Block steigt – kann es bei Block-Operationen vermehr zu Concurrency-Effekten (*buffer busy waits*) kommen.
- Wenn durch INSERT-Operationen Kompressionsoperationen angestoßen werden, muss vorher ein komplettes UNDO- sowie REDO-Protokoll des Blockes erstellt werden, d.h. das Redo Volumen wird in diesem Fall erhöht.

Folgende OLTP-Charakteristiken müssen zusätzlich beachtet werden und können auch nicht optimiert werden:

- DELETE-, INSERT- und UPDATE-Operationen sind bei OLTP nicht planbar, und erst recht nicht sortierbar, wenn man einmal von Tabellen-Clustern absieht, die selten im Einsatz sind.
- Freilisten oder Bitmap-Blöcke (ASSM) bestimmen darüber hinaus bei INSERTS den betreffenden Zielblock unabhängig vom Inhalt der Row, sondern ausschließlich auf Grund seiner Speicherlänge und der einfügenden Instanz.

Dies alles führt dazu, dass die Effizienz der De-Duplizierungsalgorithmen, die ja auf Blockebene arbeiten, zum einem gewissen Teil dem Zufall überlassen bleiben und sich damit die erzielten Kompressionsraten noch schwieriger als bei der *basic compression* vorhersagen lassen.

Index Compression

Auch bei der Komprimierung von B-Tree Indizes werden De-Duplizierungstechniken angewandt, indem die Indexeinträge in ein Präfix und ein Suffix unterteilt werden. Präfixe sind als eine vorgegebene Anzahl Tabellenspalten definiert, die Klausel `compress 2` bedeutet demnach, das die führenden zwei Spalten eines zusammengesetzten Index als Präfix angelegt werden sollen. Die Präfixe werden dann pro Block aus den Indexeinträgen heraus referenziert. Der nachfolgende logisch Block-Dump eines *non-unique* Index auf einer Tabellenspalte zeigt zwei Präfixe mit jeweils einem Spaltenwert und die Anzahl ihrer Referenzen (*prc*)

```
prefix row#0[8021] flag: -P----, lock: 0, len=11
col 0; len 8; (8):  54 65 73 74 74 65 78 74
prc 600
prefix row#1[2609] flag: -P----, lock: 0, len=12
col 0; len 9; (9):  61 61 61 62 62 62 63 63 63
prc 49
```

Die eigentlichen Indexeinträge enthalten dann – neben dem Verweis (*psno*) –nur noch die zugehörigen Row-Adressen, die bei *non-unique* Indizes im nicht komprimierten Zustand Bestandteil des eindeutigen Indexeintrages sind:

```
row#0[8012] flag: ------, lock: 0, len=9
col 0; len 6; (6):  02 00 00 cb 01 8b
psno 0
```

Die Für *non-unique* Indizes kann demnach die maximale Präfixlänge gleich der Anzahl der Indexspalten sein. Für *unique* Indizes, deren Satzadressen nicht Bestandteil des Indexeintrages sind, sind maximal die Anzahl Spalten minus 1 möglich.

Präfixe sind darüber hinaus immer an Indexspalten gebunden und unterscheiden sich darin signifikant von den bereits besprochenen *symbol tables* komprimierter Tabellen. Haben also zwei Präfix-Spalten identische Werte, werden sie trotzdem doppelt gespeichert:

```
prefix row#0[8008] flag: -P----, lock: 0, len=24
col 0; len 10; (10):  31 58 58 58 58 58 58 58 58 58
col 1; len 10; (10):  31 58 58 58 58 58 58 58 58 58
```

Des weiteren werden Indexblöcke immer in *real time* komprimiert gehalten und nicht wie bei der OLTP-Kompression im Zuge einer periodischen Blockreorganisation.

Die Komprimierung von IOTs orientiert sich naturgemäß an der B-Tree Compression und kann – da bei IOTs stets ein *unique* Index zugrunde liegen muss – nur für IOTs eingesetzt werden, deren Primärschlüssel mindestens 2 Spalten umfasst. Die Anzeige der Metadaten zeigt eine merkwürdige Mischung. DBA_TABLES:

```
TABLE_NAME COMPRESSION  COMPRESS_FOR
---------- ------------ ------------
TIOT       DISABLED     BASIC
```

Hingegen DBA_INDEXES:

```
TABLE_NAME                     INDEX_TYPE                  COMPRESS
------------------------------ --------------------------- --------
TIOT                           IOT - TOP                   ENABLED
```

LOB-Spalten

Bekanntlich lassen sich LOB-Spalten – im Gegensatz zu Secure File LOBs – nicht komprimieren. Wohl aber können sie Bestandteil einer komprimierten Tabelle sein, mit dem Resultat, dass die übrigen Spalten wie dargestellt über *symbol table* Einträge de-dupliziert werden, LOB Spalten aber standardmäßig nicht komprimiert gespeichert werden.

Zusätzlich kann es hier zu unliebsamen Überraschungen kommen, wenn Multibyte Character Sets in der Datenbank verwendet werden. In einem solchen Fall werden CLOBS bekanntlich nicht im DB-eigenen CS, sondern in einem UCS-2 kompatiblen CS gespeichert. Im folgenden Dump ist sowohl die Spalte col 1 (`varchar2`) als auch col 2 (`clob`) mit dem Zeich „x“ gefüllt, col 1 mit 20 Zeichen, col 2 mit 400 Zeichen. Die Clob-Spalte verdoppelt aber die Speicherlänge wegen des internen LOB-Speicherformats, das kompatible mit dem UCS-2 Unicode Characterset ist. Mit dem obligatorischen Lob Locator ergeben sich hier 836 Bytes! Achtung: die gerne für die Umfangsmessung verwendete Funktion `getlength` des Paketes `dbms_lob` gibt die Anzahl Zeichen und die Anzahl Bytes zurück.

```
tab 1, row 0, @0x1c0
tl: 867 fb: --H-FL-- lb: 0x2  cc: 3
col  0: [ 2]  c1 02
col  1: [20]  78 78 78 78 78 78 78 78 78 78 78 78 78 78 78 78 78 78 78 78
col  2: [836]
 00 54 00 01 02 0c 80 00 00 02 00 00 00 01 00 00 00 10 fe 35 03 30 09 00 00
 00 00 00 03 20 00 00 00 00 00 01 00 78 00 78 00 78 00 78 00 78 00 78 00 78
.  .  .
LOB
Locator:
  Length:        84(836)
  Version:        1
  Byte Length:    2
```

Secure Files

Nach den vorgehend beschriebenen De-Duplizierungen bieten Secure File LOBs tatsächliche Komprimierungsalgorithmen, die auf drei Stufen – `low`, `middle` und `high` – unterschiedliche Komprimierungsraten ermöglichen. Die Raten sind daher unabhängig von Wiederholgruppen innerhalb von Datenblöcken. Je höher die Kompressionsrate, desto CPU-aufwändiger ist der Algorithmus. Die nachfolgend aufgeführten Zahlen wurden aus einem kleinen Test gewonnen, der wie folgt angelegt war:

- Aufbau einer Basistabelle im Stil `BASICFILE`: c1 number, c2 varchar2(20), c3 clob
- Laden einer XML-Datei in die clob-Spalte, mit dem Namen der Datei in c2.
- Mehrfaches kopieren der Tabelle auf sich selbst bis 128 identische Rows vorliegen.
- Dieses der Inhalt dieser Basistabelle wird dann in Securefile Tabelle ohne sowie und mit Kompression eingefügt.

Hinweis: Die Zeile Bytegröße bezieht sich auf jeweils 1 Lob-Objekt (`used bytes` von `dbms_space`). Die restlichen Zeilen geben die Statistiken zum Laden der angegebenen 128 LOBs. Elapsed Time ist in Sekundenbruchteilen angegeben.

	Datei	**Basicfile**	**SecureFile**	**SF low**	**SF medium**	**SF high**
Bytegröße	34018	73728	73728	16384	8192	8192
Bytes komplett		9437184	9510912	2113536	1056768	1056768
CPU		17	22	12	19	29
redo		328248	445304	202936	131432	131428
undo cv		89648	167176	75404	42428	42428
com. bytes		0	0	1180032	8708608	8708608
elapsed t.		01.027605	00.571199	00.258117	00.276451	00.362111

Die Tabelle zeigt erhebliche Speichergewinne, gerade von Stufe *low* zu *medium*, die jedoch mit CPU-Kosten erkauft werden. Zusätzlich zur Komprimierung kann für Secure Files auch die De-Duplizierung aktiviert werden, die – ja nach Datenlage – zusätzliche Speichergewinne ermöglichen kann. In unserem Beispiel – 128 identische XMLs – wären diese dramatisch.

Die Komprimierung von Securefiles funktioniert unabhängig von der Tabellen und Indexkomprimierung. Diese kann ebenfalls noch für die Basistabelle eingerichtet werden.

Hybrid Columnar Compression (HCC)

In analytischen Anwendungen – wie in Decision Support Systemen oder Datawarehouses – werden die Daten in der Hauptsache selektiert, so gut wie nie verändert und in großem Umfang oft über längere Zeiträume gespeichert. Neben den auch hier einsetzbaren Basic- und Advanced Techniken, die sich beide an den Table-Rows orientieren, steht in diesem Umfeld auch die HCC-Technik zu Verfügung, über die sich erhöhte Kompressionsraten erzielen lassen. HCC nutzt für die Komprimierung eine Mischung aus Spalten- und Row-Orientierung. Ursprünglich im Kontext von Exadata-Systemen entwickelt steht die Technik mittlerweile auch für SPARC Super Cluster und zwei weiteren Storage-Systemen von Oracle zur Verfügung: ZFS Storage Appliances (über NFS) und Pillar Axiom Systeme (über Fible Channel).

HCC wird in zwei unterschiedlichen Ausprägungen bereitgestellt: `FOR QUERY` und `FOR ARCHIVE`, die beide jeweils mit den Optionen `LOW` und `HIGH` konfiguriert werden können. Durch mehr oder weniger (*for query*) starke CPU-Auslastung lassen sich unterschiedliche Kompressionsraten realisieren. Das Hauptmerkmal von HCC, das dem Feature auch seinen Namen gab, ist jedoch die Nutzung von *Compression Units (CU)*, die mehrere Oracle-Blöcke umfassen können und ein für Oracle neues Format aufweisen. Innerhalb der CUs werden Rows nach Columns organisiert. Eine Row ist stets in nur einer CU enthalten, jeder Block gehört maximal einer CU. Die Anzahl der Blöcke innerhalb einer CU variiert und hängt von der Datenlage ab. CU‘s von 4 bis 8 Blöcken sind keine Seltenheit. Beim Lesen einer Row müssen alle Blöcke der CU geladen werden, was beim häufigen Zugriff auf einzelne Rows zu einem erhöhten IO-Aufkommen führen kann.

Die HCC-Komprimierung steht nur für direct load und append-Operationen zur Verfügung. Werden Rows nachträglich verändert oder konventionell eingefügt, wandern sie in einen „normalen“ Oracle-Block, der nach dem Muster der OLTP-Compression komprimiert, d.h. de-dupliziert werden kann. Die Komprimierung wird hier – wie bereits dargestellt – beim Erreichen der Füllgrenze (*pctfree*) wirksam. Dem entsprechend steht HCC auch nicht für Index-Blöcke zur Verfügung. Es ist leicht nachvollziehbar, dass Tabellen mit konventionellen Update- und Insert-Aktivitäten durch die auftretenden Row-Migrationen schnell ihre hohen Kompressionsraten einbüßen können.

ADO: Compression und Policies

Unter der aktuellen Version 12c stehen Features zur Verfügung, welche die automatisierte Klassifizierung von Daten und in diesem Kontext die über Policies gesteuerte Komprimierung auf der Basis von Zugriffsstatistiken (*Heat-Maps*) ermöglichen. Auf diese Weise lässt sich das „Lifecycle Management“ von Daten verstärkt automatisiert umsetzen und die bereits in älteren Versionen verfügbaren Features in diesem Segment – wie beispielsweise die Partitionierung – sinnvoll ergänzen. Lizenztechnisch ist hierzu die „Advanced Compression Option“ der Enterprise Edition notwendig. Die Features sind derzeit jedoch noch nicht im Rahmen von Multitenant-Umgebungen (*pluggable database*) verfügbar. Der

Themenkomplex wird auch unter dem Terminus „Automatic Data Optimization“ (ADO) in der Dokumentation geführt.

Basis für ADO sind die sogenannten *Heat-Maps*, die eine Erweiterung der bereits aus älteren Versionen bekannten *table monitoring* Statistiken darstellen und für Tabellen und Indizes – auch auf der Basis von Partitionen und einzelnen Blöcken – Zeitstempel für Lese-, Schreib- und Scan-Operationen festhalten. Das Feature wird über den init.ora Parameter `heat_map` ein- und ausgeschaltet (Default). Die Statistiken sind über diverse Views und das Package `DBMS_HEAT_MAP` abrufbar. Um nun in den Genuss von ADO zu kommen, müssen entsprechende *policies* über `alter` oder `create table` Kommandos definiert werden, welche folgende Kriterien für ein Segment oder eine Partition festlegen können:

- Zeitraum in Tagen, Monaten oder Jahren, in denen keine Veränderungen oder Zugriffe oder gelegentliche Zugriffe stattgefunden haben darf. Alternativ kann auch die Zeitspanne nach dem Anlegen des Segmentes oder eine PL/SQL-Funktion mit booleschen Return-Wert angegeben werden.
- Kontext der Regel: das gesamte Segment, Datenblock oder der Tablespace
- Komprimierungsart, die bei Vorliegen der Kriterien durchgeführt werden soll: OLTP oder eine der HCC-Arten

Neben den beschriebenen Kompressions-Automatismen können Policies auch zum Storage Tiering eingesetzt werden. Policies lassen sich ein- und ausschalten und können natürlich auch gelöscht werden. Derzeit existieren noch einige funktionale Einschränkungen.

In-Database Archiving

Dieses Feature ist ebenfalls neu in der aktuellen Version 12c. Hierbei können Rows eines Segmentes als *inactive* markiert und dann komprimiert werden. Die Markierung erfolgt über einfache Update-Operationen einer – automatisch verfügbaren – Spalte ORA_ARCHIVE_STATE. Die Rows der betreffenden Tabelle sind nach der Markierung nach wie vor in der Datenbank gespeichert, für Applikationen jedoch nicht mehr sichtbar, wenn der Parameter ROW ARCHIVAL VISIBILITY auf ACTIVE statt ALL gesetzt wurde. Dieses Feature lässt sich mit ADO Policies auf Segment- nicht aber Row-Ebene kombinieren.

Network Compression

„Advanced Network Compression“ (ANC) ist im Rahmen der Advanced Compression Option ab der Version 12c verfügbar. ANC hilft dabei, das über Netzwerkverbindungen übertragene Datenvolumen zu reduzieren. Die Komprimierung wird pauschal über das Netzprofil (`sqlnet.ora`) oder einzelne Aliasnamen (tnsnames.ora) konfiguriert, kann über die Einstellungen high oder low unterschiedliche Wirkungsgrade zeigen sowie in Abhängigkeit von der zu übertragenen Datengröße (`threshold`) genutzt werden.

RMAN Compression

Backupsets, die mit Hilfe von RMAN erstellt werden, können binär komprimiert werden. Die Dekomprimierung findet dann transparent zum Zeitpunkt des Recovery statt. Beginnend mit der Version 11 schließlich lassen sich – vergleichbar mit den bereits besprochenen Heap-Segmenten – unterschiedliche Komprimierungstypen bzw. Algorithmen auswählen: *Basic*, *Low Medium* und *High*. Für die letz-

teren drei Varianten ist die *Advanced Compression Option* erforderlich. Mit steigenden Kompressionsraten erhöhen sich auch hier die CPU-Zyklen und die Backup-Zeiten.

Erhöhte Komprimierungsraten lassen sich zusätzlich über das sogenannte *precompression processing* erzielen. Hierbei wird der Freiplatz innerhalb einzelner Datenblöcke vor der eigentlichen Kompression optimiert (Ersetzung durch *binary zeroes*). Die CPU-Last und die Backup-Zeiten werden hierdurch nochmals gesteigert. Treffenderweise wird dieses Features durch die Klausel `OPTIMIZE FOR LOAD FALSE` aktiviert. Die Voreinstellung liegt bei `TRUE`.

Compression Advisor

Über das Paket DBMS_COMPRESSION steht ein „Compression Advisor“ zur Verfügung mit dem – recht präzise – zu erwartende Kompressionsraten errechnet werden können. Es ist zu beachten, dass für die Kalkulation ausreichend Speicherplatz für tempräre Objekte zur Verfügng stehen muss. Ein Vorteil des Advisors ist, dass über ihn auch HCC-Kompressionraten errechnet werden können, wenn keine hierzu lizensierte Storage Schicht zur Verfügung steht.

Data Guard Redo Transport Compression

Wie der Name bereits vermuten lässt, können Redo-Daten beim Übertrag auf die Standby-Site komprimiert werden. Die Kompression wird über das COMPRESSION-Attribut des Parameters LOG_ARCHIVE_DEST eingestellt. Es können keine unterschiedlichen Kompressionraten eingestellt werden. Bei synchronen Redo-Übertragungen ist darauf zu achten, dass die Network-Compression (sqlnet.ora) ausgeschaltet ist.

Kontaktadresse:

Dr. Günter Unbescheid

Database Consult GmbH

Laich 9

D-83676 Jachenau

Telefon:	+49 (0) 8043 1010
Fax:	+49 (0) 8043 1011
E-Mail	g.unbescheid@database-consult.de
Internet:	www.database-consult.de