Senkrecht oder waagerecht – Column Store oder Row Store?

Eero Mattila
Quest Software GmbH
Köln

Schlüsselworte
Oracle In-Memory Column Store

Einleitung

Mit Oracle 12c wurde die In-Memory Option eingeführt. Zentraler Bestandteil dieser Option ist die spaltenorientierte Vorhaltung von Daten, der sogenannte Column Store. Welche Vorteile bietet Column Store, wann ist es sinnvoll, die In-Memory Option zu verwenden, müssen bestehende Anwendungen für den Einsatz von Column Store angepasst werden? Was unterscheidet die In-Memory Option von der Oracle TimesTen In-Memory Database?

Dieser Vortrag bietet einen Überblick über die Architektur und Anwendungsfälle der In-Memory Option in der Oracle 12c Datenbank.

Begrifflichkeiten

Die Begriffe Row Store, Column Store, zeilen- bzw. spaltenorientiert und In-Memory können für reichlich Verwirrung sorgen. Das englische Wort „Store“ kann einerseits die persistente Speicherung von Daten auf einem Speichermedium bedeuten, andererseits aber auch einen flüchtigen Bereich für die Verarbeitung von Daten im Hauptspeicher. Ebenso kann die Zeilen- oder Spaltenorientierung für beides verwendet werden, also Speicherung oder Verarbeitung von Daten. In-Memory wiederum kann für eine Datenbank bedeuten, dass entweder alle oder nur ein Teil der Daten im Hauptspeicher vorgehalten werden, und zwar entweder zeilen- oder spaltenorientiert, oder aber auch teils teils. Damit sollten, wie es so schön heißt, aller Klarheiten beseitigt sein...

Relationale Datenbanken speichern und verarbeiten Daten „traditionell“ in zeilenorientierten Tabellen. Eine Tabelle besteht meist aus mehreren Spalten, und so enthält ein Datensatz oder eine Zeile unterschiedliche Arten von Informationen:

ID	VORNAME	NACHNAME	STRASSE	ORT	EMAIL
10001	Andi	Arbeit	Hauptstraße	Köln	andi.arbeit@firma.com
10002	Ellen	Lang	Holzweg	Bonn	ellen.lang@laden.org
10003	Lilly	Puth	Sackgasse	Plees	lilly.puth@schuppen.de

Alle Attribute eines Datensatzes werden zusammenhängend gespeichert, was einen sehr schnellen Zugriff auf alle Spalten einer Zeile erlaubt. Diese Speicherung ist immer dann ideal, wenn viele

Spalten, aber wenige Zeilen benötigt werden. Schreibende Zugriffe (Insert/Delete/Update) können bei der zeilenorientierten Speicherung extrem effizient realisiert werden, was vor allem für transaktionsorientierte Anwendungen von großer Bedeutung ist.

Eine spaltenorientierte Datenbank dagegen speichert die Informationen jeder Spalte in einer eigenen Struktur:

VORNAME
Andi
Ellen
Lilli

NACHNAME
Arbeit
Lang
Puth

STRASSE
Haupstraße
Holzweg
Sackgasse

ORT
Köln
Bonn
Plees

EMAIL
andi.arbeit@firma.com
ellen.lang@laden.org
lilli.puth@schuppen.de

Diese Art der Speicherung ist optimal, wenn eine Anwendung wenige Spalten, aber dafür große Mengen an Zeilen lesen muss. In aller Regel kommt dies in analytischen Anwendungen von, also bei Data Warehousing, Data Marts, Data Mining und dergleichen mehr, wenn riesige Datenmengen gelesen werden müssen. Da die einzelnen Spalten bei dieser Art der Speicherung zudem stark komprimiert werden können, lassen sich bei Selects sehr große Performance-Vorteile erzielen. Da jedoch ein Insert, Update oder Delete hier alle Spaltenstrukturen einzeln anfassen muss, leidet die Effizienz bei schreibenden Zugriffen.

Spaltenorientierte Datenbanken sind nichts Neues – erste Implementierungen gab es bereits in den 1970-er Jahren, z.B. RAPID der kanadischen Bundesbehörde für Statistiken. Erstes kommerzielles Produkt war wohl Sybase IQ (heute SAP IQ) um 1995. Mittlerweile gibt es eine Reihe von sowohl proprietären als auch Open Source Lösungen.

Auch In-Memory Datenbanken sind seit langem auf dem Markt, und auch Oracle hat mit TimesTen In-Memory Database bereits seit geraumer Zeit eine eigene Variante, und zwar eine *zeilenorientierte* Datenbank, die komplett im Hauptspeicher liegt und für die Persistenz entweder eine „normale“ Oracle Datenbank oder andere, proprietäre Mechanismen verwendet. Oracle TimesTen zielt auf hochaktive OLTP-Anwendungen, die auf extrem schnelle Antwortzeiten angewiesen sind. Der In-Memory Column Store dagegen ist etwas ganz anderes.

Oracle In-Memory Column Store – Architektur und grundsätzliche Handhabung

Oracle hat mit In-Memory Column Store einen anderen Ansatz gewählt. Die persistente Speicherung der Daten auf Festplatte bleibt Zeilenorientiert, ebenfalls der effiziente Buffer Cache. Der In-Memory Column Store ist eine zusätzliche Struktur im Hauptspeicher, der den analytischen Abfragen zur Laufzeit zur Verfügung steht. Nicht alle Tabellen müssen im Column Store vorgehalten werden, sondern es sollten üblicherweise nur die performancekritischsten dafür vorgesehen werden.

Das heißt also, dass der Column Store einerseits keinen zusätzlichen Plattenplatz belegt (oder nicht unbedingt, aber dazu später mehr), dafür aber zusätzliche Ressourcen im Hauptspeicher benötigt. Die System Global Area (SGA) enthält nun einen neuen Bereich, genannt In-Memory Area. Dieser Bereich wird nicht automatisch aktiviert (schließlich ist die In-Memory Option kostenpflichtig und nur für die Enterprise Edition lizensierbar), sondern muss zunächst mittels Initialisierungsparameter konfiguriert werden.

```
alter system set inmemory_size = 4G scope=spfile;
```

Mit diesem Befehl wird die Größe der In-Memory Area gesetzt – und beim ersten Mal zieht dies unweigerlich einen Neustart der Instanz nach sich. Ab Oracle12c Release 2 kann die Größe nachträglich im laufenden Betrieb erhöht (jedoch nicht reduziert) werden. Zu beachten ist unbedingt, dass der In-Memory Bereich Teil der SGA ist, folglich muss dort Kapazität vorhanden sein – wenn nicht, lässt sich ein Neustart wieder nicht vermeiden.

Ein Blick auf die SGA zeigt nun Folgendes:

```
SQL> select name, value from v$sga;

NAME                         VALUE
-------------------- ----------
Fixed Size                8761232
Variable Size           838860912
Database Buffers       1291845632
Redo Buffers              8015872
In-Memory Area         4294967296 <-----
```

Auch wenn das Automatic Memory Management (mit memory_target) oder Automatic Shared Memory Management (mit sga_target) verwendet wird, so wird die Größe der In-Memory Area dadurch nicht verwaltet. Objekte in diesem Bereich unterliegen auch nicht dem „least recently used“ (LRU) Algorithmus und können daher nicht dynamisch ausgelagert werden, wenn sie länger nicht genutzt wurden, wie z.B. beim Buffer Cache. Ist in der In-Memory Area kein Platz für ein Objekt, wird es nicht (oder nicht komplett) in den Column Store geladen. In diesem Fall wird eine Fehlermeldung im Alert-Log vermerkt.

Nachdem die In-Memory Area aktiviert wurde, können Objekte (Tabellen, Partitionen, Subpartitionen und Materialized Views) in den Column Store geladen werden.

```
create table <tabelle> […] INMEMORY;
```

oder

```
alter table <tabelle> INMEMORY;
```

Von dieser einfachsten Syntax ist jedoch insofern abzuraten, als sie dazu führt, dass Oracle selbst entscheidet, wann und in welcher Reihenfolge die Objekte tatsächlich in den Column Store geladen werden. Ohne weitere Parameter zum INMEMORY-Attribut wird ein Objekt nämlich erst dann in den in die In-Memory Area geladen, wenn es zum ersten Mal verwendet wird, sei es durch ein SELECT- oder ein DML-Statement. Es empfiehlt sich daher, den Zusatzparameter PRIORITY gezielt einzusetzen.

```
alter table <tabelle> INMEMORY PRIORITY CRITICAL;
alter table <tabelle> INMEMORY PRIORITY HIGH;
alter table <tabelle> INMEMORY PRIORITY MEDIUM;
alter table <tabelle> INMEMORY PRIORITY LOW;
alter table <tabelle> INMEMORY PRIORITY NONE; <--- Standardwert!
```

Die Objekte werden in entsprechender Reihenfolge geladen, zunächst also alle mit CRITICAL, dann HIGH und so weiter. Alle Tabellen mit mindestens Priorität LOW werden auch beim Neustart der

Instanz direkt in den Column Store geladen. Sinnvollerweise versieht man daher die wichtigsten Tabellen mit CRITICAL und die dazugehörigen Objekte mit HIGH. Zusätzlich zur Priorität können mit weiteren Parametern einerseits die Komprimierung der Objekte im Hauptspeicher und das Verhalten des Column Stores bei RAC und Active Data Guard beeinflusst werden. RAC-Anwender sollten die Parameter DUPLICATE und DISTRIBUTE sehr genau studieren und testen, um Performance-Nachteile zu vermeiden. Nicht unerheblich ist die Tatsache, dass der Parameter DUPLICATE, der für die Fehlertoleranz im RAC-Umfeld sehr wichtig ist, nur für Engineered Systems zur Verfügung steht. Abhilfe kann hier der nicht dokumentierte Parameter _inmemory_auto verschaffen – aber der ist eben nicht dokumentiert...

Das Attribut INMEMORY kann auch auf Tablespace-Ebene definiert werden. Alle neuen Objekte, die danach in einem solchen Tablespace erstellt werden, erben das INMEMORY-Attribut.

Standardmäßig wird eine Tabelle, Partition oder Materialized View komplett in den Column Store geladen. Es ist aber auch möglich, nur eine Untermenge der Spalten eines Objects für In-Memory zu definieren. Da der Column Store eine feste Größe hat und niemals von sich aus Objekte auslagert, sollte die Befüllung gut überlegt sein. Bei großen Tabellen sollten also nur die Spalten in den Column Store geladen werden, die wirklich davon profitieren können. Nichts ist frustrierender, als festzustellen, dass der Column Store zur Neige geht, ehe die benötigten Objekte geladen werden konnten...

Zunächst ist die In-Memory Area also eine reine Hauptspeicherstruktur. Wenn eine Instanz neu gestartet wird, ist der Column Store zunächst leer und muss befüllt werden. Dies kann sehr CPU-intensiv sein und recht viel Zeit in Anspruch nehmen. Wie oben erwähnt, bedeutet dies aber auch keinen zusätzlichen Bedarf an Plattenplatz. Wer jedoch den Neustart der Instanz beschleunigen will und bereit ist, dafür mehr Massenspeicher zu opfern, kann den FastStart-Mechanismus ab Release 12.2 verwenden. Damit wird in der Datenbank ein Tablespace zur FastStart Area erklärt, und die Inhalte des Column Store werden in diesem Tablespace als Secure File Lobs hinterlegt. Bei einem Neustart wird der Column Store wieder aus diesem Tablespace befüllt, was unter Umständen wesentlich schneller vonstatten geht als der vollständige Neuaufbau, da die Umformatierung aus Zeilenformat in Spaltenformat entfällt.

Selbstverständlich bringt die In-Memory Option eine Reihe von V$-Views mit sich. Welche Objekte in den Column Store geladen wurden und ob der Ladevorgang fertig ist, kann man z.B. mit der folgenden Abfrage herausfinden:

```
select v.segment_name name,
       v.populate_status pop_status,
       v.bytes/1024/1024/1024 orig_GB,
       v.inmemory_size/1024/1024/1024 in_mem_GB,
       v.bytes / v.inmemory_size comp_ratio,
       v.bytes_not_populated missing_bytes
from v$im_user_segments v
order by 1;
```

NAME	POP_STATUS	ORIG_GB	IN_MEM_GB	COMP_RATIO	MISSING_BYTES
ADRESSEN	COMPLETED	0,01953125	0,008056640625	2,42424242424242	0
AUFTRAEGE	COMPLETED	0,2392578125	0,1629638671875	1,46816479400749	0
PERSONEN	COMPLETED	0,0087890625	0,004150390625	2,11764705882353	0
POSITIONEN	COMPLETED	1,5	0,45611572265625	3,28863910076275	0

Erste Ergebnisse

Bereits in einer überschaubaren Umgebung lässt sich die Performancesteigerung feststellen. Als Beispiel sollen hier Abfragen gegen ein kleines Auftragsverwaltungssystem dienen. Die „größte“ Tabelle, POSITIONEN, enthält knapp 26 Millionen Zeilen und belegt ca. 1,5 GB Plattenplatz.

Nehmen wir als Beispiel ein einfaches Statement, sicherlich weder elegant noch beispielhaft für eine tiefgehende Analyse, es geht hier ums Prinzip. Dieses Statement soll den Gesamtumsatz für Badehosen und –anzüge für die letzten zwei Jahre ausgeben.

```
  SELECT  produktname, SUM (auftragssumme) Umsatz
    FROM (  SELECT a.aufid, produktname, menge * einzelpreis auftragssumme
              FROM doag.positionen pos,
                   doag.auftraege a,
                   doag.produkte pro,
                   doag.status  s
             WHERE     a.aufstatus = s.statusid
                   AND (    (    (RTRIM (TO_CHAR (aufdatum, 'Month')) IN
                                      ('Juli', 'August', 'September'))
                             AND (TO_CHAR (aufdatum, 'yyyy') IN ('2016', '2017')))
                    AND (produktname IN ('Badehose', 'Badeanzug')))
                   AND S.kurzbeschreibung = ('GELIEFERT')
                   AND pos.aufid = a.aufid
                   AND pos.prodid = pro.prodid
          GROUP BY a.aufid, produktname, menge * einzelpreis)
GROUP BY produktname;
```

Mit dem NO_INMEMORY Hint kommt die Antwort nach 7 Sekunden:

Database	Start Time	Stop Time	Execution Time	SQL
DOAG@EM12	20/11/2017 17:27:55	20/11/2017 17:28:03	7 secs	SELECT /*+ NO_INMEMORY */ produktname, SUM (auftragssumme) Umsatz

Da die Tabellen nur für Primär- und Fremdschlüssel indiziert sind, führt die Abfrage zwangsläufig zu FULL TABLE SCANs:

```
-------------------------------------------------------------------------------------------
| Id  | Operation               | Name       | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |            |     2 |    52 |       | 63816   (2)| 00:00:03 |
|   1 |  HASH GROUP BY          |            |     2 |    52 |       | 63816   (2)| 00:00:03 |
|   2 |   VIEW                  | VM_NWVW_0  |  3253 | 84578 |       | 63816   (2)| 00:00:03 |
|   3 |    HASH GROUP BY        |            |  3253 |   231K|   280K| 63816   (2)| 00:00:03 |
|*  4 |     HASH JOIN           |            |  3253 |   231K|       | 63756   (2)| 00:00:03 |
|*  5 |      TABLE ACCESS FULL  | STATUS     |     1 |    11 |       |     3   (0)| 00:00:01 |
|*  6 |      HASH JOIN          |            | 16264 |   984K|  7312K| 63753   (2)| 00:00:03 |
|*  7 |       HASH JOIN         |            |   138K|  5683K|       | 53321   (1)| 00:00:03 |
|*  8 |        TABLE ACCESS FULL| PRODUKTE   |     2 |    36 |       |     3   (0)| 00:00:01 |
|   9 |        TABLE ACCESS FULL| POSITIONEN |    25M|   593M|       | 53253   (1)| 00:00:03 |
|* 10 |       TABLE ACCESS FULL | AUFTRAEGE  |   824K|    15M|       |  8825   (5)| 00:00:01 |
```

Unter Verwendung des In-Memory Column Store bleibt die Antwortzeit deutlich unter einer Sekunde:

Database	Start Time	Stop Time	Execution Time	SQL
DOAG@EM12	20/11/2017 17:26:05	20/11/2017 17:26:05	392 msecs	SELECT produktname, SUM (auftragssumme) Umsatz

Der Ausführungsplan zeigt nun eine neue Zugriffsmethode, TABLE ACCESS INMEMORY FULL.

```
-------------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name       | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |            |     2 |    52 |       |  5778   (16)| 00:00:01 |
|   1 |  HASH GROUP BY                     |            |     2 |    52 |       |  5778   (16)| 00:00:01 |
|   2 |   VIEW                             | VM_NWVW_0  |  3253 | 84578 |       |  5778   (16)| 00:00:01 |
|   3 |    HASH GROUP BY                   |            |  3253 |   231K|   280K|  5778   (16)| 00:00:01 |
|*  4 |     HASH JOIN                      |            |  3253 |   231K|       |  5718   (16)| 00:00:01 |
|*  5 |      TABLE ACCESS FULL             | STATUS     |     1 |    11 |       |     3    (0)| 00:00:01 |
|*  6 |      HASH JOIN                     |            | 16264 |   984K|  7312K|  5715   (16)| 00:00:01 |
|   7 |       JOIN FILTER CREATE           | :BF0000    | 16264 |   984K|       |  5715   (16)| 00:00:01 |
|*  8 |        HASH JOIN                   |            |   138K|  5683K|       |  3364   (15)| 00:00:01 |
|   9 |         JOIN FILTER CREATE         | :BF0001    |     2 |    36 |       |     3    (0)| 00:00:01 |
|* 10 |          TABLE ACCESS INMEMORY FULL| PRODUKTE   |     2 |    36 |       |     3    (0)| 00:00:01 |
|  11 |         JOIN FILTER USE            | :BF0001    |    25M|   593M|       |  3295   (13)| 00:00:01 |
|* 12 |          TABLE ACCESS INMEMORY FULL| POSITIONEN |    25M|   593M|       |  3295   (13)| 00:00:01 |
|  13 |       JOIN FILTER USE              | :BF0000    |   824K|    15M|       |   745   (58)| 00:00:01 |
|* 14 |        TABLE ACCESS INMEMORY FULL  | AUFTRAEGE  |   824K|    15M|       |   745   (58)| 00:00:01 |
-------------------------------------------------------------------------------------------------------------
```

Schon in diesem kleinen Beispiel ist die Antwortzeit um mehr als Faktor 10 kürzer. Berichte über unterschiedlichste Anwendungsfälle mit wesentlich größeren Datenmengen und realistischen Data Mart Anwendungen belegen einhellig die Performance-Steigerung um mindestens Faktor 10 bis x, je nach Art der Abfragen und Charakter der Daten.

In-Memory Methoden zur Query Optimierung

Die tatsächliche Funktionsweise der Query Optimierung soll hier nicht vertieft werden – das Thema ist viel zu umfangreich für diesen Rahmen. Dazu gibt es reichlich Literatur von Oracle in Form von White Papers und Dokumentation sowie zahlreiche Blogs im Netz. Ein paar seien hier kurz erwähnt:

- In-Memory Expressions – häufig verwendete Ausdrücke wie Funktionen können in der In-Memory Area hinterlegt werden, um wiederholte Evaluierung der Ausdrücke zu minimieren.
- In-Memory Virtual Columns – virtuelle Spalten können im Column Store hinterlegt werden, um wiederholte Berechnung der Werte zu minimieren.
- Join Groups – häufig gejointe Spalten können im Column Store als Gruppen definiert werden, sodass Joins schneller ausgewertet werden.
- ADO Support für den Column Store – veraltete oder selten benötigte Daten können automatisch aus dem Column Store gelöscht werden, um Platz für aktuellere Daten zu verschaffen.

Anwendungsfälle für Oracle In-Memory Column Store

Das primäre Ziel der In-Memory Option ist die Optimierung von analytischen Abfragen, die mit sehr großen Datenmengen agieren. Data Marts, Date Warehouses sowie Systeme, die eine Mischung aus OLTP- und Analyseaufgaben erfüllen müssen, sind die besten Kandidaten für den Einsatz.

Die größten Vorteile bietet die In-Memory Option im Bereich Datenzugriff für Analyse und Reporting. Ohne, dass eine bestehende Anwendung geändert werden muss, können enorme Verbesserungen der Antwortzeiten bei Abfragen erzielt werden. Andererseits können ggf. viele Indizes, die bisher ausschließlich aus Gründen der Abfrageperformance gepflegt werden mussten, komplett gelöscht werden, was zur allgemeinen Leistungssteigerung der Datenbank führen kann.

Schreibende Zugriffe werden in aller Regel keinen Vorteil erfahren, aber auch keinen Nachteil spüren. Auch hochselektive Statements, die z.B. über Primary Keys oder einfache Joins über Primary Key / Foreign Key einzelne Zeilen abfragen, werden kaum begünstigt. Für Anwendungen, die von solchen Abfragen geprägt sind, kann die Oracle TimesTen Variante die besser Wahl sein, sofern die Daten vollständing im Hauptspeicher untergebracht werden können.

Welche Abfragen profitieren nun am meisten von der Nutzung des Column Stores? Als Faustregel gilt: Je mehr Daten *gelesen* werden im Verhältnis zu den Daten, die letztendlich *verarbeitet* (bzw. zurückgegeben) werden, umso größer ist der mögliche Gewinn durch die In-Memory-Verarbeitung.

Nehmen wir ein Beispiel mit einer Million Zeilen, die von zwei verschiedenen Abfragen gelesen werden. Die erste Abfrage schließt 990.000 Zeilen durch die Abfragekriterien aus und verarbeitet 10.000 Zeilen. Die zweite Abfrage schließt nur 10.000 Zeilen aus und muss 900.000 Zeilen verarbeiten. Befindet sich die Tabelle im In-Memory Column Store, wird die erste Abfrage einen größeren Nutzen davon haben.

Folgende Merkmale von Abfragen sind von grundlegender Bedeutung für den zu erzielenden Performancegewinn:

1) **Anzahl der zu selektierenden Spalten.** Je mehr Spalten abgefragt werden, umso aufwändiger ist die Verarbeitung und geringer der Gewinn.

   `SELECT *` ist schlecht, `SELECT spalte` ist besser.

   Alle Spalten zu lesen dauert länger als nur eine Spalte.

2) **Anzahl der zurückgegebenen Zeilen.** Je mehr Zeilen zurückgegeben werden, umso aufwändiger ist die Verarbeitung und geringer der Gewinn.

   `SELECT spalte` ist schlecht, `SELECT SUM(spalte)` ist besser.

   Alle Zeilen zurückzugeben dauert länger als nur eine Zeile.

3) **Selektivität der Abfrageprädikate.** Je mehr Zeilen die Abfragekriterien erfüllen, umso aufwändiger ist die Verarbeitung und geringer der Gewinn.

   `SELECT AVG(zahl) ... WHERE zahl > 2` ist schlecht,
   `SELECT AVG(zahl) ... WHERE zahl < 2` ist besser.

   Mehr Zeilen zu berechnen dauert länger als weniger Zeilen.

4) **Anzahl der Tabellen in der Abfrage.** Je komplexer die Abfrage, umso aufwändiger ist die Verarbeitung und geringer der Gewinn.

   `SELECT ... from a,b,c,d,e,f,g WHERE <JOIN-Bedingungen)` ist schlecht,
   `SELECT ... from a,b,c WHERE <JOIN-Bedingungen)` ist besser.

   JOIN-Bedingungen über 6 Tabellen zu verarbeiten dauert länger als über 3 Tabellen.

Weitere Merkmale, wie die Verwendung von PL/SQL-Funktionen oder Prozeduren, können natürlich ebenfalls den Nutzen beeinflussen.

Fazit

Die Antwort auf die Frage aus dem Titel dieses Vortrags lautet: sowohl als auch! Der In-Memory Column Store ändert nichts an der physikalischen Speicherung der Daten in der Oracle Datenbank. Er kann aber die Performance von analytischen Abfragen erheblich erhöhen, ohne dass bestehende Anwendungen angepasst werden müssen. Durch den Einsatz des Column Store kann jedoch unter Umständen auf viele komplexe Indizes verzichtet werden, wodurch die OLTP-Leistung gesteigert werden kann. Entscheidend dafür, wie hoch der Nutzen durch die Verwendung der In-Memory Option ausfällt, ist einerseits die Datenstruktur und andererseits die Natur der Anwendungen. Data Warehouse Systeme und Star Schemas sind die natürlichen Kandidaten für den Einsatz der In-Memory Option. Die Einrichtung und Verwaltung des In-Memory Column Store ist sehr einfach, wobei in RAC-Umgebungen auf die Parametrierung besonderes Augenmerk gerichtet werden muss.

Quellen:

Oracle Dokumentation (Oracle12cR2)
Oracle White Papers
Diverse Blogs wie carajandb.com, blogs.oracle.com u.v.a.m.

Kontaktadresse:
Eero Mattila
Quest Software GmbH
Im Mediapark 4e
DE-50670 Köln

Telefon: +49 (0) 221-5777 4169
Fax: +49 (0) 221-5777 450
E-Mail eero.mattila@quest.com
Internet: http:// https://www.quest.com/de-de/