Die neue In-Memory-Option der Datenbank 12c

Herbert Rossgoderer und Matthias Fuchs, ISE Information Systems Engineering GmbH

Dieser Artikel stellt neben dem neuen In-Memory Column Store auch In-Memory Cache und JSON vor, stellt anhand von kleinen Beispielen die Funktionsweise vor und reflektiert erste Erfahrungen aus dem Betatest.

Im Rahmen des ersten Patchsets 12.1.0.2 für die 12*c* Datenbank führt Oracle viele neue Funktionen und Features ein. Die bekanntesten und wichtigsten sind:

- JSON
- In-Memory Caching
- In-Memory Column Store

JSON steht für Java Script Object Notation. Darunter versteht man halb- beziehungsweise unstrukturierte Datenobjekte, die in vielen NoSQL-Datenbanken als Basis verwendet werden, beispielsweise auch in der MongoDB. JSON-Objekte können als „VARCHAR2", „CLOB", „BLOB" etc. in einer Tabelle abgespeichert sein. Der Zugriff erfolgt mit SQL-Abfragen direkt auf die Attribute. Arrays, also Mehrwert-Attribute können als Row Source abgefragt werden. Ebenso sind Indizes auf Attribute innerhalb der JSON-Objekte möglich.

In-Memory Caching

Bei den bisherigen Datenbank-Versionen war es aufwändig, größere Datenmengen beziehungsweise Blöcke im Buffer Cache dauerhaft vorzuhalten. Um zu gewährleisten, dass auch größere Tabellen im Speicher gehalten werden, mussten die Tabellen mit dem Storage-Attribut „KEEP" gekennzeichnet sein. Full Database Caching ermöglicht nun, die gesamte Datenbank im Memory abzulegen. Voraussetzung dafür ist, dass der Buffer-Cache größer ist als die Summe aller Datafiles, abzüglich „SYSAUX" und „TEMP"-Tablespace. Die Initialisierung erfolgt mit „ALTER DATABASEFORCE FULL DATABASE CACHING;". Beim ersten Zugriff werden die Daten in den Speicher geladen.

Falls nur Teile der Datenbank geladen werden sollen, kann Automatic Big Table Caching verwendet werden, um ganze Objekte auf Basis von Zugriffshäufigkeiten im Buffer-Cache zu halten. Somit sind dort nur die sehr häufig zugegriffenen Objekte vorhanden. Sobald der Bereich voll ist, werden die weniger genutzten Objekte wieder komplett herausgenommen um Platz für Neue zu schaffen. Im Gegensatz dazu erfolgt bei der traditionellen blockweisen Speicherung im Buffer-Cache die Vorhaltung im Memory auf Basis der Blockzugriffe ohne Objektbezug. Das Big Table Caching wird mit dem Parameter „DB_BIG_TABLE_CACHE_PERCENT_TARGET" aktiviert, indem ein Prozentwert relativ zum Buffer Cache angegeben wird.

In-Memory Column Store

Der In-Memory Column Store ist ein neuer Pool in der SGA. Die Tabellen werden im In-Memory Column Store in spaltenorientierter Weise abgelegt. Inhaltlich sind die Strukturen im In-Memory Column Store zu den Strukturen im Buffer-Cache konsistent. Es ergeben sich somit zwei Möglichkeiten, um Daten abzulegen. Einmal im zeilenweisen („row format") oder im dualen Format. Im dualen Format sind sowohl die zeilenbasierten als auch die spaltenorientierten Formate („columnar format") vorhanden *(siehe Abbildung 1)*. Der Optimizer entscheidet, auf welchem Pool zugegriffen werden sollen.

Das zeilenbasierte Format hat sich bei OnLine-Transaction-Processing-Zugriffen (OLTP) bewährt. Da bei OLTP-Systemen meistens die gleichen Blöcke gelesen und geschrieben werden, kann bereits mit einer geringen Anzahl von In-Memory-Blöcken (Buffer-Cache) eine deutliche Steigerung der Performance erreicht werden. Wenn zum Beispiel zehn Prozent der vorhandenen Daten im Buffer Cache (Memory) stehen und 95 Prozent der Abfragen dort ablaufen, ergibt sich eine Gesamtbeschleunigung um den Faktor zwanzig (100/5). Im Gegensatz dazu werden bei analytischen Abfragen in sogenannten „Decision Support System"-Umgebungen (DSS) große Teile der Daten gelesen und

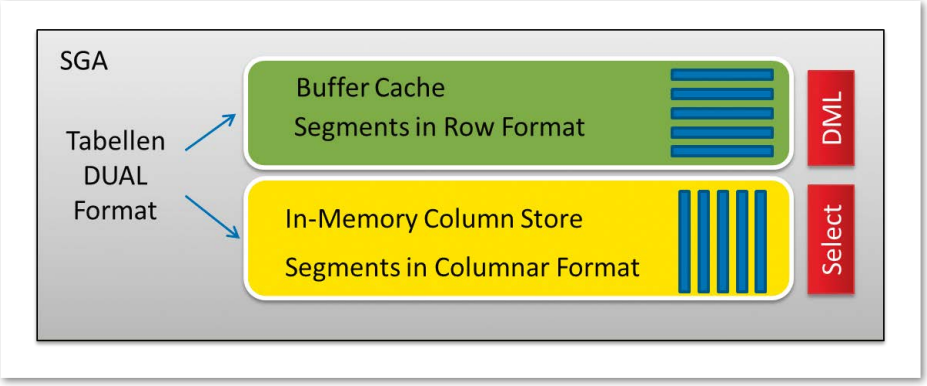

Abbildung 1: In-Memory Dual Format

aggregiert oder anderweitig weiterverarbeitet. Wenn hier nur zehn Prozent der Daten im Memory liegen, aber 90 Prozent der Daten nicht aus dem Buffer-Cache gelesen werden können, sondern zum Beispiel von Disk, erreicht man lediglich noch einen Faktor von 1,1 (100/90). In Kombination mit Komprimierung und prozessoroptimierter Verarbeitung (Single Instruction, Multiple Data, SIMD) können weitere Performance-Steigerungen erreicht werden. Die Ressourcen (RAM) werden dadurch geschont.

Ziele des In-Memory Column Store

Der In-Memory-Ansatz soll die Abfrage-Performance, vor allem in analytischen Umgebungen, steigern. Dies wird durch den direkten Zugriff aller Daten im Memory erreicht. Gerade bei Verwendung von analytischen Funktionen können Berechnungen beschleunigt werden (SIMD), was zu deutlichen Performance-Steigerungen führt.

Um die Performance traditionell in einer Data-Warehouse-Umgebung zu steigern, sind Indizes erforderlich. Da im Rahmen eines Aggregat-Laufs große Teile einer Tabelle gelesen werden, müssen ebenfalls Indizes über die meisten Spalten hinzugefügt werden. Somit kann in einem Data Warehouse das Verhältnis „Indizes zu Tabellengröße“ von 2 zu 3 und mehr erreichen. Die Pflege der zusätzlichen Struktur ist aufwändig und erfordert viele Anpassungen beim Aufbau. Die Verwendung des In-Memory Stores dagegen ist eine Anweisung („ALTER TABLE INMEMORY“) und es gibt nur wenige Variationen. Die Konfiguration ist somit schnell erledigt.

Ein Nebeneffekt aus der Verwendung von In-Memory Stores anstelle von Indizes ist eine Beschleunigung bei Update und Insert Statements (DML), da keine Indizes parallel gepflegt werden müssen. Die Pflege des In-Memory Stores bedeutet einen deutlich geringeren Aufwand.

Die Daten vorbereiten

Der Befehl „alter table lineitem inmemory;“ lädt eine Tabelle in den Hauptspeicher (populate). Beim ersten Aufruf wird der Columnar In-Memory Store aufgebaut. Durch ein einfaches „SELECT“ werden die Daten transferiert „select count(*) from lineitem;“. Die Umsetzung dauert je nach Größe und Geschwindigkeit des Storage bis zu einigen Minuten. Der Status lässt sich über die System-View „v$im_segments“ abfragen (siehe *Listing 1* und *Abbildung 2*).

```
SELECT v.owner,
  v.segment_name name,
  v.populate_status status,
  v.bytes_not_populated
FROM v$im_segments v
ORDER BY bytes_not_populated DESC;
```

Listing 1

TPCH	LINEITEM	STARTED	55725686784
TPCH	LINEITEM	STARTED	54393135104
TPCH	ORDERS	STARTED	11784044544
TPCH	ORDERS	STARTED	6970802176
TPCH	PARTSUPP	STARTED	5811478528
TPCH	PARTSUPP	STARTED	2967216128
TPCH	PART	STARTED	1339424768
TPCH	CUSTOMER	STARTED	1004298240
TPCH	CUSTOMER	STARTED	801906688
TPCH	PART	STARTED	667910144

Abbildung 2: „v$im_segments“ während des Aufbaus

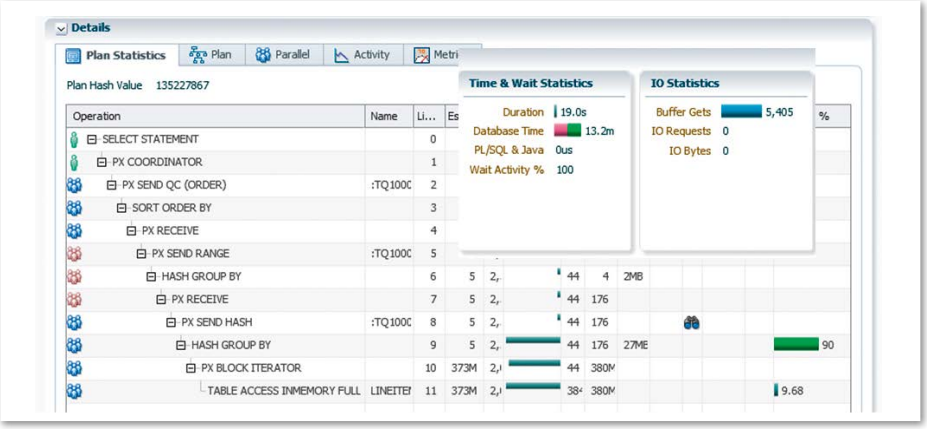

Abbildung 3: In-Memory Plan

Der Session-Parameter „inmemory_query“ gibt an, ob der In-Memory Store benutzt werden soll. Er kann mit „ENABLE/DISABLE“ gesetzt werden. In *Abbildung 3* sieht man einen Ausführungsplan unter Verwendung des In-Memory Stores. An den I/O-Statistiken kann man klar erkennen, dass kein I/O entsteht.

Im Vergleich dazu sieht man in *Abbildung 4* den Unterschied ohne In-Memory. Es werden 54 GB von Platte beziehungsweise aus dem Buffer-Cache gelesen. An dem Verhältnis der Buffer-Gets von In-Memory und ohne sieht man die unterschiedlichen Arten der Ablage („columnar“ vs. „rows“).

Vergleich zwischen In-Memory und traditionellem Buffer-Cache

Um die Unterschiede des Zugriffs zu zeigen, wurden drei Statements aus dem TPC-H-Benchmark herausgegriffen und gegen verschiedene Systeme laufen gelassen. Das TPC-H-Schema ist in Version 2.17 aufgesetzt, mit Indizes, Primary und Foreign Keys. Die verwendeten Systeme waren virtuelle Installationen auf einer Oracle Virtual Machine (OVM) mit 22 Cores und 100 GB SGA (130GB RAM) und eine Installation auf Hardware mit 256 GB SGA (1 TB RAM) und 40 Prozessor Cores. Als Storage wurde in beiden Fällen ein Oracle Sun ZFS 7320 verwendet, das mit Direct NFS (dNFS) angesprochen wurde. *Abbildung 5* zeigt die Werte für ausgewählte Abfragen.

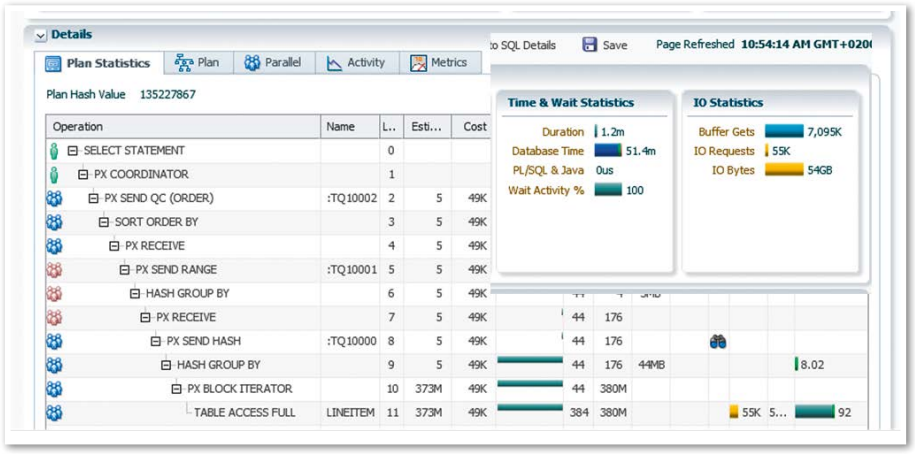

Abbildung 4: Standard Plan ohne In-Memory

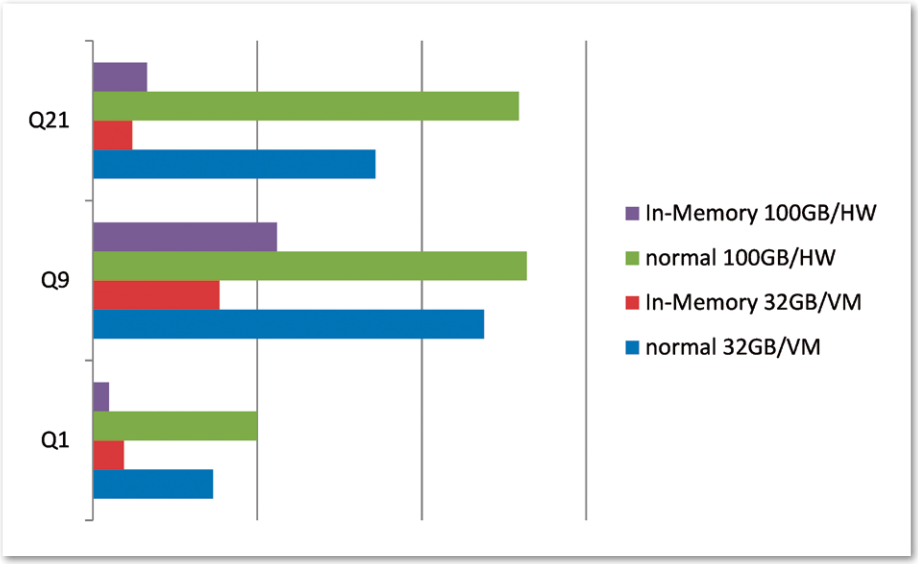

Abbildung 5: Abfragezeiten im Vergleich

Die Performance-Steigerungen durch Verwendung des In-Memory Stores sind deutlich zu erkennen, jedoch vom Statement abhängig. Im ersten Statement werden viele analytische Funktionen verwendet, darunter auch „min()“, die sehr günstig für den direkten In-Memory Zugriff sind. Insgesamt wird eine deutliche Performance-Verbesserung erreicht.

RAC mit zwei Knoten

Die Verteilung des In-Memory Column Store kann entweder gespiegelt oder verteilt auf die einzelnen Cluster-Knoten erfolgen und lässt sich automatisch oder basierend auf RowIds, Partitions oder Subpartitions definieren. Die Definition erfolgt zum Beispiel mit „ALTER TABLE "DM"."POS_ALL" inmemory priority medium distribute by partition;“. Dies bedeutet, dass der In-Memory Store mit mittlerer Priorität angelegt wird. Beim Setzen von „priority“ wird der Store nach dem Start oder nach der Konfiguration automatisch geladen. Zuerst wird „priority“ mit „high“, dann „medium“ und zum Schluss „low“ erstellt. Ein Aktivieren durch den ersten Zugriff ist nicht notwendig. Die Verteilung („distributed“) soll auf Basis von Partitionen erfolgen.

Listing 2 zeigt eine Abfrage, um die Verteilung einzusehen. In *Abbildung 6* erkennt man, dass die Partitionen abwechselnd auf „inst_id 1“ und „2“ bestückt wurden. Die Komprimierung beträgt immer mindestens Faktor „2“.

Der Optimizer entscheidet, ob der In-Memory Store verwendet wird. Im Ausführungsplan erscheint dann der Eintrag „TABLE ACCESS INMEMORY FULL“ *(siehe Abbildung 7)*. Es wurde das Statement „select /*+ MONITOR */ count(*) from "DM"."POS_ALL" t;“ mit einem Parallelisierungsgrad von „8“ verwendet. Der Hint sichert nur, dass das Statement im SQL-Monitor erscheint.

In der Tabelle sind 138.576.922 Zeilen. Eine Abfrage dauert bei einer Parallelität von „8“ etwa zehn Sekunden mit In-Memory Store – bei ausgeschaltetem Store rund vier Minuten.

Es ist zu bedenken, dass der In-Memory ColumnStore über zwei Knoten verteilt ist. Der RAC besteht aus zwei virtuellen Maschinen mit 22 virtuellen Cores und einer Oracle Sun ZFS 7320 Appliance, der über dNFS angebunden ist. Die Abfrage über einen Index auf die Anzahl der Zeilen ist etwas kürzer als zehn Sekunden („index fast full scan“: sechs Sekunden).

Die Tabelle hat aktuell eine Größe von 30GB und 387 Spalten. Mit dem In-Memory Column Store ist es möglich, auf allen Spalten Abfragen zu gestalten, egal ob mit oder ohne Index.

Das Statement „select min(t.fpos_eccs_cumulated_val_por) from "DM"."POS_ALL" t;“ läuft ebenfalls in acht Sekunden. Ohne In-Memory dauert die Ausführung mehr als vier Minuten – ein Index ist auf dieser Spalte natürlich keiner vorhanden. Somit erreicht man über alle 387 Spalten diese Performancesteigerung. Gerade bei vielen Spalten muss beachtet werden, dass, wenn ein großer Teil davon gelesen wird und die Anzahl der verwendeten Zeilen sinkt, der zeilenbasierte Zugriff schneller ist. Genau diesen Punkt herauszufinden ist die Aufgabe des Optimizers.

Fazit

Die In-Memory-Funktionalitäten können sehr schnell und einfach eingesetzt werden. Änderungen an bisherigen Anwendungen sind nicht notwendig, um die Optimierungen umzusetzen. Bei Verwendung von analytischen Abfragen, also Abfragen, bei denen große Teile einer Tabelle gelesen werden, sind deutliche Performance-Steigerungen zu erwarten. Bei gezielter Verwendung von Indizes könnten spezielle Abfragen schneller

```
SELECT inst_id,
Partition_name PARTITION,
  owner,
  bytes/1024/1024/1024 DISK_GB,
  round(inmemory_size/1024/1024/1024,2) IM_GB,
  inmemory_compression Comperssion,
  round(bytes/inmemory_size,2) comp_ratio
FROM gV$IM_SEGMENTS order by partition_name;
```

Listing 2

	INST_ID	PARTITION	OWNER	DISK_GB	IM_GB	COMPERSSION	COMP_RATIO
1	1	008D0229	DATA	0,1328125	0,06	FOR QUERY LOW	2,22
2	1	008D0430	DATA	0,1796875	0,08	FOR QUERY LOW	2,19
3	2	008D0531	DATA	0,1875	0,09	FOR QUERY LOW	2,2
4	1	008D0630	DATA	0,25	0,11	FOR QUERY LOW	2,28
5	2	008D0731	DATA	0,2109375	0,1	FOR QUERY LOW	2,2
6	2	008D0930	DATA	0,265625	0,12	FOR QUERY LOW	2,16
7	2	008D1130	DATA	0,2265625	0,11	FOR QUERY LOW	2,15
8	2	009D0131	DATA	0,0390625	0,09	FOR QUERY LOW	0,42
9	1	009D0430	DATA	0,04296875	0,1	FOR QUERY LOW	0,42
10	1	009D0630	DATA	0,0703125	0,17	FOR QUERY LOW	0,43
11	2	009D0930	DATA	0,0703125	0,18	FOR QUERY LOW	0,4

Abbildung 6: Verteilung im RAC

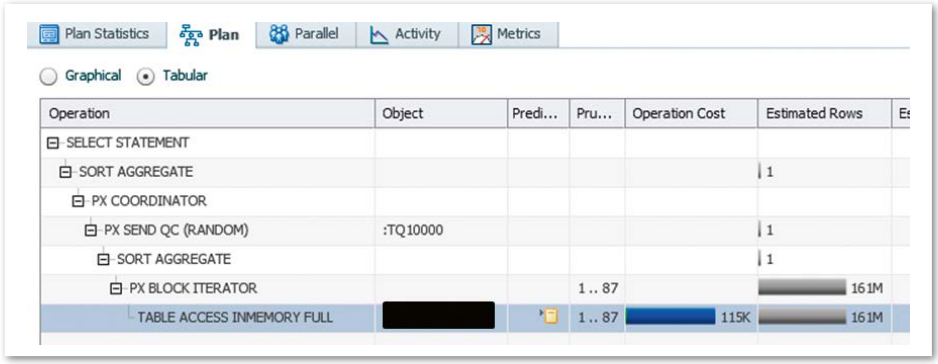

Abbildung 7: In-Memory Execution Plan

sein. Dies wird aber mit mehr Storage Verbrauch und langsameren DMLs erkauft. Eine Performance-Einbuße bei Änderungen in den Daten aufgrund von der zusätzlichen, parallelen Pflege des In-Memory Stores konnte nicht festgestellt werden.

Beim Aufbau des In-Memory Column Stores sollte man beachten, dass CPU- und I/O-Verbrauch sehr stark ansteigen. Dies kann man durch Reduzierung der Prozesse, die zum Laden des Stores verwendet werden, verringern. Der Aufbau kann dann allerdings deutlich länger dauern. Bei jedem Restart der „Instanz/DB" geht der In-Memory Column Store verloren und muss neu geladen werden. In einer RAC-Umgebung kommen weitere administrative Herausforderungen hinzu. Es ist zu entscheiden, ob verteilte oder duplizierte Stores erstellt werden. Zudem lässt sich die Verteilung automatisch oder manuell angeben.

Herbert Rossgoderer
herbert.rossgoderer@ise-informatik.de

Matthias Fuchs
matthias.fuchs@ise-informatik.de

Wir begrüßen unsere neuen Mitglieder

Persönliche Mitglieder

Till Brügelmann
Fabian Gaußling
Thomas Geisel
Wolfgang Parzinger
Christina Veit
Sabrina Schönthaler
Sven Buchholz
Stefan Hoeller
Johann Lodina
Daniel Jansen
Reinhard Vielhaber
Britta Wolf
Patrick Bär

Firmenmitglieder

Forensis Finance & Controlling
inxire GmbH