# Die Effizienz von SmartScan für die Datenbank einschätzen

Franck Pachot, dbi-services

Der Artikel in der letzten Ausgabe hat gezeigt, dass man zuerst „direct-path read“ einführen muss, damit SmartScan wieder ins Spiel kommt. Mit dem Simulationsmodus lässt sich die Effizienz von SmartScan grob abschätzen. Dieser Beitrag geht nun ins Detail, um zu verstehen, warum das betreffende Volume gefiltert werden kann oder nicht. Jenseits von Theorie und Marketing wird gezeigt, ob die Aktivität der Datenbank mit den Anwendungen, Benutzern und Daten beim Umstieg auf Exadata an Leistung gewinnen wird.

SmartScan wird nicht deswegen etwas bringen, weil man viele „direct path read“ durchführt und man überprüft hat, dass ein Großteil der „physical read bytes“ durch „cell simulated physical IO bytes eligible for predicate offload“ abgedeckt sind. Seine Effizienz hängt davon ab, was gefiltert werden soll.

## Projection Offloading

Die Projektion ist die Auswahl der Spalten, die für die SELECT-Clause notwendig ist. Man geht beispielsweise mit dem oben genannten Parameter in den Simulationsmodus und startet einen „SELECT *“-Befehl ohne WHERE-Clause. *Listing 1* zeigt den Ausführungsplan (Format: „allstats last +predicates“).Mit „+projection“ werden alle Spalten des SELECT-Befehls angezeigt. *Tabelle 1* zeigt die wesentlichen Statistiken bei der Ausführung.

Alles wurde dem SmartScan unterworfen, ohne dass etwas gefiltert worden wäre. Man hat damit also nichts gewonnen. Jetzt folgt nochmal das Gleiche, aber nur mit wenigen Spalten („SELECT ID“, „SEQ FROM DEMO2“, siehe *Listing 2*).

Die Spalten sind in der Projektion des Vorgangs „STORAGE“ sichtbar und diese Projektion wurde somit – via iDB – mit einem E/A-Abruf übermittelt. *Tabelle 2* zeigt das Ergebnis. Auf dem betreffenden Volume wurden von SmartScan nur 23 Prozent zurückgegeben. Andere Spalten als „ID“ und „SEQ“ wurden als Reaktion auf den E/A-Abruf von den an die Datenbank zurückgegebenen Blöcken entfernt. Es ist zu beachten, dass die Statistiken der Simulation zwar „predicate offload“ heißen, tatsächlich aber alle „offload“ Predicate und Projection enthalten. Das hier ist der Beweis.

## Predicate Offloading

Bei der nächsten Abfrage kommt ein Prädikat „WHERE MOD(ID,2)=0“ hinzu, um die Hälfte der Zeilen zu beseitigen *(siehe Listing 3)*. Man sieht im Ausführungsplan, dass das Prädikat durch den Speichervorgang verarbeitet werden kann. Tatsächlich beträgt das von SmartScan zurückgegebene Volumen nur die Hälfte von zuvor *(siehe Tabelle 3)*.

| Statistic | Total |
|---|---|
| cell simulated physical IO bytes eligible for predicate offload | 2,340,585,472 |
| cell simulated physical IO bytes returned by predicate offload | 2,121,558,080 |
| physical reads bytes | 2,346,206,208 |
| physical reads direct | 285,716 |

*Tabelle 1*

```
-------------------------------------------------------------------------------------
| Id  | Operation                 | Name  | Starts | E-Rows | A-Rows | Buffers | Reads  |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |       |      1 |   2000K |  2000K|     285K|    285K|
|   1 |  TABLE ACCESS STORAGE FULL| DEMO2 |      1 |   2000K|   2000K|     285K|    285K|
-------------------------------------------------------------------------------------

Column Projection Information (identified by operation id):
-----------------------------------------------------------
1 - "ID"[NUMBER,22], "DEMO2"."SEQ"[NUMBER,22], "DEMO2"."D"[NUMBER,22],
"DEMO2"."E"[NUMBER,22], "DEMO2"."F"[VARCHAR2,1000], "DEMO2"."G"[VARCHAR2,4000]
```

*Listing 1*

```
-------------------------------------------------------------------------------------------
| Id  | Operation                 | Name  | Starts | E-Rows | A-Rows | Buffers | Reads  |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |       |      1 |   2000K |  2000K|     285K|    285K|
|   1 |  TABLE ACCESS STORAGE FULL| DEMO2 |      1 |   2000K|   2000K|     285K|    285K|
-------------------------------------------------------------------------------------------

Column Projection Information (identified by operation id):
-----------------------------------------------------------
1 - "ID"[NUMBER,22], "SEQ"[NUMBER,22]
```

*Listing 2*

```
-------------------------------------------------------------------------------------------
| Id  | Operation                 | Name  | Starts | E-Rows | A-Rows | Buffers | Reads  |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |       |      1 |   1000K |  1000K|     285K|    285K|
|*  1 |  TABLE ACCESS STORAGE FULL| DEMO2 |      1 |   1000K|   1000K|     285K|    285K|
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
1 - storage(MOD("ID",2)=0)
filter(MOD("ID",2)=0)

Column Projection Information (identified by operation id):
-----------------------------------------------------------
 1 - "ID"[NUMBER,22], "SEQ"[NUMBER,22]
```

*Listing 3*

## Join Offloading

Es wird hier nicht weiter ins Detail gegangen, man muss allerdings bedenken, dass die Reaktionszeit einer Abfrage sich nicht aus dem Lesen der Daten ergibt. Bei komplexen Abfragen gibt es Verknüpfungen, Sortierungen und Aggregation. Das alles geschieht nur auf dem Datenbankserver mithilfe von Tempfiles. SmartScan bringt hier nichts.Für die Filterung der Verknüpfungen gilt eine Ausnahme: Bei der Erstellung der Hash-Tabelle (die kleinere der beiden Tabellen eines „HASH JOIN“) wird ein Bloom-Filter erstellt, mit dem beim Lesen der zweiten Tabelle viele Zeilen entfernt werden können, selbst ohne die Entsprechung mit der Hash-Tabelle vorzunehmen. Dieser Bloom-Filter ist wie ein Prädikat: Er wird auf dem Speicherserver ausgeführt, wenn SmartScan läuft. Man kann diesen Filter als große „IN ()“-Clause verstehen, die als erster Grobfilter funktioniert.

*Abbildung 1* zeigt die Funktionsweise: Wenn die Hash-Tabelle erstellt wird, entsteht ein Filter, der diese Tabelle darstellt, aber False Positives enthalten kann. Verwendet SmartScan die zweite Tabelle (was das an die Datenbank zu übertragende Volumen begrenzt), brauchen nur noch die False Positives entfernt zu werden.

| Statistic | Total |
|---|---|
| cell simulated physical IO bytes eligible for predicate offload | 2,340,585,472 |
| cell simulated physical IO bytes returned by predicate offload | 55,272,816 |
| physical reads bytes | 2,341,306,368 |
| physical reads direct | 285,716 |

*Tabelle 2*

| Statistic | Total |
|---|---|
| cell simulated physical IO bytes eligible for predicate offload | 2,340,585,472 |
| cell simulated physical IO bytes returned by predicate offload | 27,967,088 |
| physical reads bytes | 2,343,059,456 |
| physical reads direct | 285,716 |

*Tabelle 3*

Betrachtet man nur die „wait events“ bei Verknüpfungen zwischen zwei großen Tabellen, wird man viele „direct path read temp“ sehen, die keinem SmartScan unterworfen sind. Wenn man allerdings mit der Simulationsbibliothek fortfährt, wird man sehen, dass der Bloom-Filter viel-

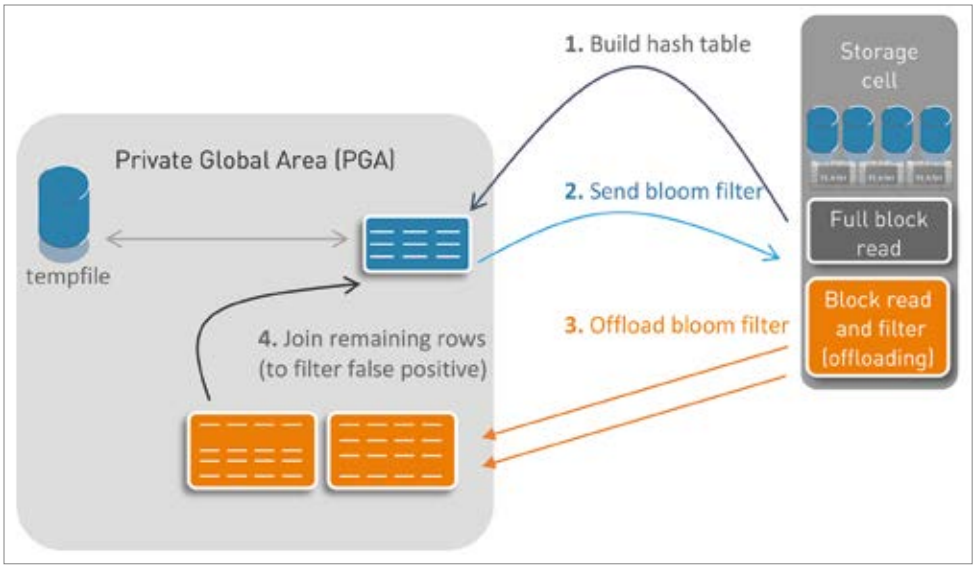


*Abbildung 1: Bloom-Filter für Hash Join Offloading*

| Event | Waits | Time(s) | Avg Wait(ms) | % Total Call Time | Wait Class |
|---|---|---|---|---|---|
| **read by other session** | 46,201,741 | 158,777 | 3 | 50.2 | User I/O |
| **db file sequential read** | 87,834,244 | 102,268 | 1 | 32.3 | User I/O |
| **CPU-Zeit** | | 27,646 | | 8.7 | |
| **db file scattered read** | 6,313,301 | 5,369 | 1 | 1.7 | User I/O |
| **enq: TX - row lock contention** | | | | | |

*Tabelle 4*

| Event | Waits | Time(s) | Avg Wait(ms) | % Total Call Time | Wait Class |
|---|---|---|---|---|---|
| **DB CPU** | | 1,685 | | 49.63 | |
| **direct path read** | 347,955 | 347,955 | 3 | 27.26 | User I/O |
| **db file sequential read** | 599,906 | 271 | 0 | 7.99 | User I/O |
| **enq: TX - row lock contention** | 185 | 89 | 481 | 2.62 | Application |
| **log file sync** | 23,503 | 76 | 3 | 2.24 | Commit |

*Tabelle 5*

| Statistic | Total |
|---|---|
| **cell physical IO interconnect bytes** | 221,656,831,180 |
| **cell simulated physical IO bytes eligible for predicate offload** | 336,000,856,474 |
| **cell simulated physical IO bytes returned by predicate offload** | 187,871,446,630 |
| **physical reads bytes** | 361,291,243,520 |
| **physical write bytes** | 1,785,430,016 |

*Tabelle 6*

leicht einige dieser Lese-/Schreibvorgänge der Tempfiles ersparen wird.

Betrachten wir nochmals die Simulationsstatistiken. Der Bloom-Filter erscheint nur im Parallel Query bis 12.1.0.2, wo er im Serial für In-Memory verwendet wird. Tatsächlich gibt es jedoch weitere Fälle, in denen man ihn im Serial beobachten kann.

Beim Auslesen einiger Spalten einer Tabelle, beim Lesen von mehr als ein paar Zeilen oder beim „HASH JOIN" zwischen großen Tabellen – hier wird klar, bei welcher Art von Datenbanken SmartScan seine Vorteile aufzeigt: für Reporting, Data Warehouse (ETL und Abfragen), analytische Abfragen, DSS etc.

Bei einer ERP-Datenbank, die alle Spalten („SELECT*") einiger Zeilen („WHERE PK=...") auslesen wird, bringt SmartScan keine Vorteile. Der Autor kennt eine Banking-Anwendung, die alles als CLOB (XML) speichert und die nur über den Primärschlüssel zugreift. Dafür kann man Exadata vergessen, ebenso In-Memory. Beides ist nicht dafür gemacht. Heute gibt es für diese Datenmodelle NoSQL-Datenbanken, die kurz vor den relationalen SQL-Datenbanken entworfen wurden. Bei ERP- oder anderen Datenbanken, die die Parameter „optimizer_index_caching,optimizer_index_cost_adj" definieren, um den Zugriff über den Index zu erzwingen, wird SmartScan ebenso wenig etwas bringen.

## Den SmartScan-Vorteil bewerten

Man nimmt einen AWR-Bericht und wirft einen Blick auf den Abschnitt „Top Events". Die Angabe „%DB Time" gegenüber „direct path read" zeigt den maximalen Vorteil, den man erwarten kann. Dazu einige Beispiele.

*Tabelle 4* zeigt die „Top 5 Timed Events". Diese Basis hat einen eingeschränkten Puffer-Cache, führt aber meist Single-Block-Lesevorgänge durch. SmartScan bringt hier nichts. Man muss zuerst diese Lesevorgänge verstehen. Ein fehlerhafter Ausführungsplan wird möglicherweise immer wieder die gleichen Zeilen lesen. Auch der Kauf eines neuen Rechners wird die Feinabstimmung nicht vermeiden.

In *Tabelle 5* sind die „Top 5 Timed Foreground Events" dargestellt. Hier kann SmartScan 27 Prozent der Reaktionszeit eindämmen. Es wäre also inter-

essant weiterzugehen, um den Gewinn einzuschätzen. Dazu wird die Simulation mit „_rdbms_internal_fplib_enabled“ aktiviert. Man vergleicht das von SmartScan zurückgegebene Volumen mit den „physical read bytes“, um die Effizienz des Filtervorgangs bewerten zu können.

*Tabelle 6* zeigt die interessanten Statistiken. Es mussten 336 GB gelesen werden („physical reads bytes“) und dennoch wurden nur 206 GB zwischen der Speicher- (hier simuliert durch „fplib“) und der Datenbank-Schicht übertragen. Genauer gesagt: Diejenigen Schreibvorgänge müssen entfernt werden, die in „cell physical IO interconnect bytes“ ebenfalls enthalten sind. Das vorgestellte Beispiel hat keine ASM-Redundanz, sodass das durch Schreibvorgänge übertragene Volumen den „physical write bytes“ entspricht. Bei normaler Redundanz hätte man es mit „zwei“ multiplizieren müssen.

Man musste also 336 GB auslesen, von denen 313 GB durch SmartScan verarbeitet werden. 175 GB wurden von SmartScan zurückgegeben; 206 - 1x1,6 - 175=29 GB wurden zurückgegeben, ohne von SmartScan bearbeitet worden zu sein. Durch Übernahme der Gesamtzahl der Lese-/Schreibvorgänge braucht man schließlich nur 206/(336+1,6) = 61 Prozent des notwendigen Volumens zu übertragen.

Angesichts der Tatsache, dass die E/A 27,26 und 7,99 Prozent der „DB Time“ ausmachen, ist ein Gesamtgewinn von (0,2726 + 0,0799) x 0,61 = 21,5 Prozent der „DB Time“ mit Exadata zu erwarten. Das ist gut, man sollte allerdings gegenüber seinen Benutzern keine Verpflichtung über diesen Wert hinaus eingehen.

## Laufende Änderungen

Die Simulation dient nur zur Bewertung des gefilterten Datenvolumens. Die Effizienz von SmartScan hängt auch von den gleichzeitigen Änderungen ab. Zuerst war zu erkennen, dass der Checkpoint im Falle einer Vielzahl veränderter Blöcke zu aufwändig wäre und SmartScan nicht durchgeführt wird. Es gibt jedoch auch das Problem der Konsistenz: Man kann weder die nicht durchgeführten Änderungen noch die nach Beginn der Abfrage durchgeführten Änderungen sehen. So kann es plötzlich sein, dass SmartScan den vollständigen Block zu einem bestimmten Zeitpunkt ohne jede Filterung zurückgeben muss, weil nur der Code der Datenbank ein konsistentes Image rekonstruieren kann. *Tabelle 7* zeigt mit „Top 10 Foreground Events by Total Wait Time“ einen AWR-Auszug dieses Falls.

| Event | Waits | Time(s) | Avg Wait(ms) | % Total Call Time | Wait Class |
|---|---|---|---|---|---|
| **DB CPU** | | 2.2 | | 54.3 | |
| **enq: KO - fast object checkpoint** | 1 | 1.7 | 1690.24 | 42.7 | Application |
| **direct path read** | 1,130 | .3 | 0.28 | 8.0 | User I/O |
| **db file sequential read** | 535 | .2 | 0.30 | 4.0 | User I/O |
| **SQL*Net more data to client** | 402 | .1 | 0.12 | 1.3 | Network |
| **enq: RO - fast object reuse** | 2 | 0 | 1.56 | .1 | Application |
| **SQL*Net message to client** | 213 | 0 | 0.01 | .0 | Network |
| **reliable message** | 3 | 0 | 0.43 | .0 | Other |
| **control file sequential read** | 137 | 0 | 0.01 | .0 | System I/O |
| **log file sync** | 1 | 0 | 0.73 | .0 | Commit |

*Tabelle 7*

Man sieht, dass nur 8 Prozent der Zeit auf den SmartScan („direct path read“) entfallen. Man hat aber 42 Prozent der Zeit für die Einrichtung dieses Checkpoints aufgewendet, um diese „direct path read“ durchführen zu können. Es geht allerdings noch schlechter. In *Tabelle 8* sind alle Statistiken zu „cell“ in den Simulationsmodus eingebracht.

Die Statistiken „cell blocks processed by ...“ geben das von jeder der verschiedenen SmartScan-Schichten verarbeitete Volumen an. Es wurden 142.870 Blöcke im Direktmodus gelesen. Alle diese Blöcke wurden durch die „cache layer“ verarbeitet, aber ausgehend von „transaction layer“ ergeben sich weniger Blöcke.

SmartScan hat Blöcke gesehen, deren SCN außerhalb der SCN der Abfrage liegt. Diese Blöcke müssen ein „undo“ anwenden, um ein konsistentes Image zu rekonstruieren. Die „storage cell“ kann dies nicht machen: Sie hat keinen Zugriff auf die Festplatten der anderen und kann daher nicht das gesamte „undo“ einsehen.

SmartScan war hier also deutlich weniger effektiv als erwartet. Beim Betrachten der Zahlen versteht man, warum weiter oben komplizierte Berechnungen durchgeführt wurden, um die Effizienz zu messen. Einige Tools (einschließlich Oracle) erstellen einfach das Verhältnis „eligible“ zu „returned“, vergessen in diesem Fall aber alle betroffenen Blöcke, die nicht von SmartScan verarbeitet wurden.

Das Verhältnis erscheint gut: Die Hälfte des Volumens wird von SmartScan zurückgegeben. Tatsächlich entspricht aber das ausgetauschte Volumen („cell physical IO interconnect bytes“) selbst nach Entfernen der Schreibvorgänge („physical write bytes“) dem gesamten betreffenden Volumen. In diesem Beispiel gibt es also keinen Vorteil. Nur einen Zeitverlust durch den Checkpoint.

SmartScan ist nicht für die Optimierung einer OLTP geschaffen, bei der die Änderungen genauso wichtig sind wie die Lesevorgänge. Deswegen müssen den Blöcken die Funktionen zugewiesen werden, die auf Ebene der Speicherserver nicht verfügbar sind.

## Fazit

Man muss verstehen, wo SmartScan angewendet werden kann und wo nicht. Denn dies ist der wichtigste Faktor der Leistungsverbesserung mit Exadata im Vergleich zu jeder anderen Plattform. Wie man also sieht, ist Exadata eher für Data Warehouse gemacht:

- Full Table Scan oder Index Fast Full Scan
- Parallel Query oder Serial Direct Read

| Statistic | Total | per Second | per Trans |
|---|---|---|---|
| **cell IO uncompressed bytes** | 1,170,391,040 | 213,965,455.21 | 97,532,586.67 |
| **cell blocks helped by minscn optimization** | 10 | 1.83 | 0.83 |
| **cell blocks processed by cache layer** | 142,870 | 26,118.83 | 11,905.83 |
| **cell blocks processed by data layer** | 78,128 | 14,283.00 | 6,510.67 |
| **cell blocks processed by txn layer** | 78,128 | 14,283.00 | 6,510.67 |
| **cell commit cache queries** | 36,182 | 6,614.63 | 3,015.17 |
| **cell physical IO interconnect bytes** | 1,215,807,488 | 222,268,279.34 | 101,317,290.67 |
| **cell scans** | 5 | 0.91 | 0.42 |
| **cell simulated physical IO bytes eligible for predicate offload** | 1,170,391,040 | 213,965,455.21 | 97,532,586.67 |
| **cell simulated physical IO bytes returned by predicate offload** | 539,262,144 | 98,585,401.10 | 44,938,512.00 |
| **physical read bytes** | 1,174,036,480 | 214,631,897.62 | 97,836,373.33 |
| **physical reads** | 143,315 | 26,200.18 | 11,942.92 |
| **physical reads cache** | 445 | 81.35 | 37.08 |
| **physical reads direct** | 142,870 | 26,118.83 | 11,905.83 |
| **physical write bytes** | 23,306,240 | 4,260,738.57 | 1,942,186.67 |

*Tabelle 8*

- Wenige gleichzeitige Aktualisierungen
- Kompression

Man wird eher zu Exadata raten, wenn man bereits alles getan hat, was für die vorhandene Infrastruktur möglich war. Es gilt zunächst, Parallel Query zu beherrschen, um alle lizenzierten CPU zu verwenden. Eventuell bietet sich auch RAC an, um die Leistung mehrerer Server einzusetzen. Wenn das optimal läuft und es keine weitere Möglichkeit mehr als die E/A-Bandbreite auf den „direct path read" gibt, kann man wirklich auf die schnellere Geschwindigkeit mit Exadata umsteigen. Der Autor hat einen Kunden, dem genau das passiert ist, als es um die Erfassung von Leistungsproblemen ging. Bei einer OLTP-Last wird SmartScan keine Vorteile bringen, da es folgende Punkte nicht verbessert:

- „SELECT*"-Anfragen
- Zugriffe auf wenige Zeilen über den Index (etwa durch PK)
- Optimierungen in „FIRST_ROWS"
- Gleichzeitige Aktualisierungen mit dem Puffer-Cache und Undo, beides sehr wirksamen Mechanismen. Dank derer hat Oracle den Markt der transaktionalen Datenbanken beherrscht (außer den Mainframes).

Natürlich kann man Exadata wählen, um sein Data Warehouse zu beschleunigen, und mit der Absicht einer Konsolidierung auch seine OLTP-Datenbanken integrieren. Selbst wenn SmartScan nicht angewendet wird, kann man dennoch immer von einer leistungsstarken Maschine profitieren. Bei OLTP-Lasten muss man mit Exadata jedoch drei Dinge beachten:

- Nicht vergessen, dass die HCC-Komprimierung für Tabellen erfolgt, die in erster Linie schreibgeschützt sind. Deutlich ausgedrückt: Bis heute gibt es noch keine Kompression, die effizient auf Änderungen reagiert.
- Der Wechsel auf RAC bedeutet nicht, dass alle Dienste auf allen Knoten geöffnet werden müssen. Die Arbeit an den gleichen Blöcken mit verschiedenen Instanzen wird alle vorhandenen Beschränkungen verstärken.
- Nicht alle Index-Dateien löschen. Unter OLTP dienen die Index-Dateien auch dazu, Einschränkungen bei der Eindeutigkeit zu validieren, ein Sperren der Tabellen zu vermeiden etc.

Vor einer Entscheidung über eine neue Infrastruktur gilt es zunächst, die aktuellen Datenbanken zu prüfen und zu versuchen, den möglichen Gewinn einzuschätzen. Man kann durch Einblick in einen AWR-Bericht eine erste Vorstellung gewinnen und mehr erfahren, wenn man SmartScan simuliert sowie einige Berechnungen mit den entsprechenden Statistiken durchführt.

Der Artikel soll kein negatives Bild von Exadata vermitteln, bei der es sich um eine außergewöhnliche Maschine handelt. Natürlich kann man auch seine OLTP-Datenbanken einbringen, damit wird man allerdings keine herausragenden Leistungen erzielen. Ein vereinfachender Vergleich: Man kann mit dem Sportwagen auch durch die Stadt fahren, aber dafür hat man ihn nicht erworben.

Franck Pachot
franck.pachot@dbi-services.com