Exadata: Real World Performance

Randolf Geist
Unabhängiger Berater
Mannheim, Deutschland

Schlüsselworte
Exadata, Performance, SQL Processing, Offloading, Analyse

Einleitung

Die Exadata Plattform liefert überragende Performance, wenn die Exadata-spezifischen Features entsprechend eingesetzt werden können. Insbesondere die sogenannten Exadata „Smart Scans“ können Full Segment Scans extrem beschleunigen und optimieren.

Der Clou der Plattform liegt darin, dass Teile der Datenverarbeitung in die sogenannten „Storage Cells“ verlagert werden können. Wir haben es hier also mit „intelligentem“ Storage zu tun, der versteht, was in den abgelegten Daten abgespeichert ist und bestimmte Operationen bereits im Storage Layer durchführen kann.

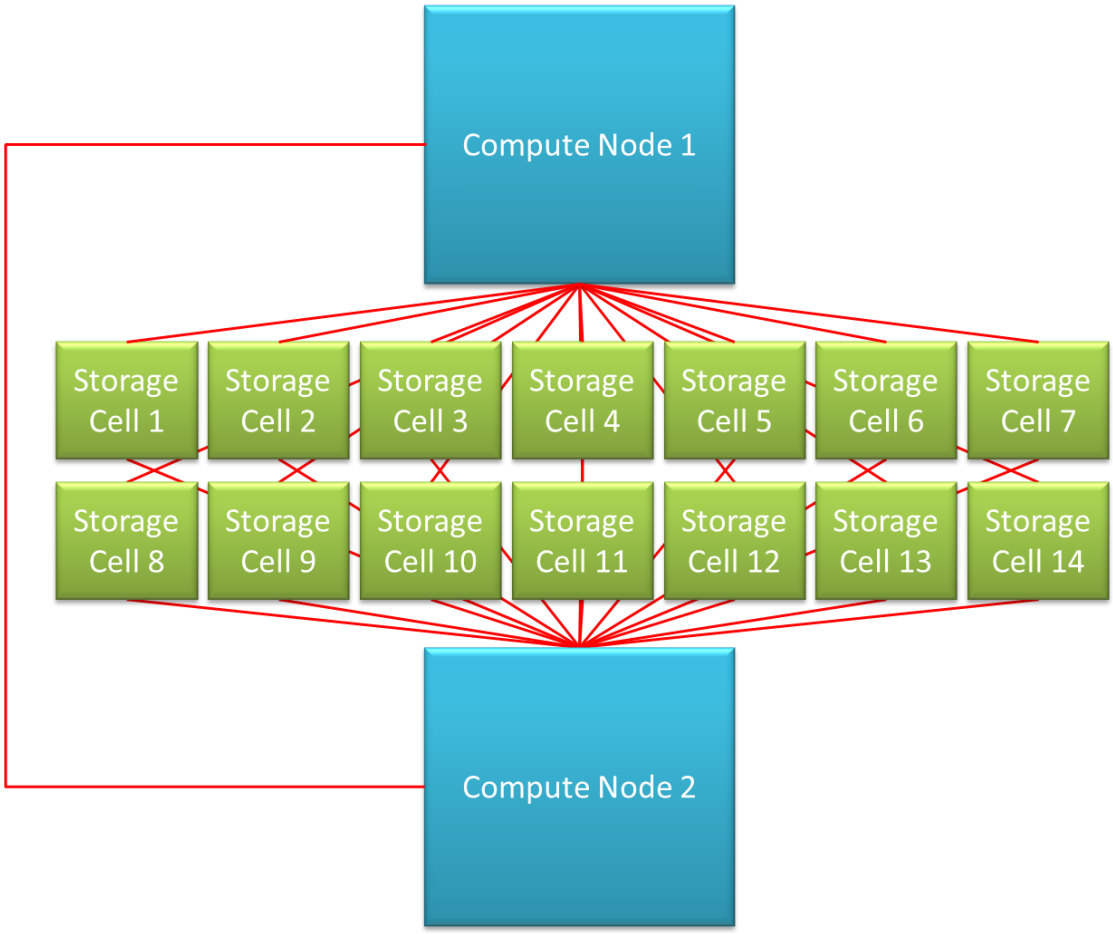


Abbildung 1: Exadata X3-8 Architektur

Dies hat mehrere Vorteile: Zum Einen können diese Operationen bereits auf Storage Layer-Ebene parallelisiert und optimiert werden, zum Anderen kann durch die Aufteilung der Operationen zwischen Storage Layer und Compute Node Layer die Menge der Daten, die zwischen den beiden Layern ausgetauscht werden muss, reduziert werden.

Die Verbindung zwischen den Compute Nodes und den Storage Cells verwendet Infiniband und ein dafür optimiertes Protokoll. Auch die Compute Nodes sind über Infiniband verbunden. Allerdings findet keine Kommunikation zwischen den Storage Cells statt – auf Storage Cell Ebene ist Exadata also eine „Shared Nothing"-Konfiguration, was einige interessante Implikationen hat.

Wie man den offiziellen Datenblättern entnehmen kann, können Smart Scans in einem Exadata Full Rack (oder den Xn-8 (X3-8 / X4-8) Modellen) bei den aktuellen Modellen in Verbindung mit Flash Cache bis zu 100GB pro Sekunde (unkomprimiert) erreichen.

Oracle spricht hier offiziell von „SQL Processing Speed" oder auch „Maximum SQL bandwidth", was suggeriert, dass komplexere SQL Operationen in den „Storage Cells" verarbeitet werden können.

Eine interessante Frage ist, wie sich diese Performance verändert bei zunehmend komplexerem SQL. Zu diesem Zweck überprüfen wir zuerst, ob wir die versprochene Performance tatsächlich erreichen können, und können dabei eine detailliertere Analyse kennenlernen, die uns erlaubt, einen genaueren Einblick in verwendeten Exadata-spezifischen Features zu erhalten.

Der untersuchte Fall

Ausgangspunkt der gesamten Analyse war eine bestimmte, extrem zeitkritische Abfrage eines Kunden, die es zu optimieren galt. Es handelte sich dabei um eine SQL-Abfrage, die 20 Tabellen mittels Outer Join verknüpfte. Die zu verarbeitende Gesamtdatenmenge war ca. 210GB, wobei durch die Struktur der Abfrage und Daten nur Full Table Scans der Tabellen in Frage kam. Dabei kam den Exadata spezifischen Features entgegen, dass nur ca. 1/12 der Daten (ca. 18GB) tatsächlich für die Joins benötigt wurde (ungefähr 1/4 durch Selektion und davon 1/3 durch Projektion ausgewählt), und außerdem ein gewisse physische Ordnung der Daten vorlag, was wiederum den Einsatz von Storage Indizes zur Vermeidung von physischem I/O ermöglichen kann. Allerdings erlaubt das hier beschriebene Szenario nicht den Einsatz von sogenannten Bloom Filtern, eine Technologie, die es erlaubt, Joins, die Daten filtern, beim Einsatz von Smart Scans zu optimieren, da der Bloom Filter in den Storage Cells verarbeitet und somit die an die Compute Nodes zu sendende Datenmenge minimiert werden kann.

Um festzustellen, ob die versprochene Performance von 100GB pro Sekunde grundsätzlich erreicht werden kann, wird zuerst eine simple SELECT COUNT(*)-Abfrage auf ein UNION ALL der 20 Tabellen gemacht und eine detaillierte Analyse der entsprechenden Session Statistiken durchgeführt.

Danach wird schrittweise die Komplexität erhöht. Komplexität heisst in diesem Falle vor allem, dass in den „Compute Nodes" mehr mit den von den „Storage Cells" übertragenen Daten gemacht werden muss im Sinne von SQL Processing, da eben dieser Teil des SQL Processing nicht in den „Storage Cells" durchgeführt werden kann.

Analyse Exadata Features

Über entsprechende Statistiken kann ausgewertet werden, welche Exadata-spezifischen Features zu welchem Grad eingesetzt werden. Dies ist wichtig für ein genaueres Verständnis, wie eine bestimmte Performance im Exadata-Umfeld zustande kommt.

Unter anderem folgende Features können darüber analysiert werden:

- Smart Scans: Einsatz von Smart Scans, und zu welchem Grad Smart Scans verwendet wurden, da je nach Szenario (zum Beispiel Read Consistency, Chained Rows, oder hohe CPU-Auslastung der „Storage Cells“) auch bei Smart Scans die Menge der in den Zellen verarbeiteten Daten stark variieren kann

- Offloading Percentage: Wie viele Daten wurden in den „Storage Cells“ verarbeitet, im Verhältnis zu der Menge an Daten, die zwischen den Zellen und den „Compute Nodes“ ausgetauscht wurden. Hier sind auch durchaus negative Prozente möglich, da bei Einsatz von Kompression oder Spiegelung von Schreibvorgängen (ASM Mirroring) mehr Daten zwischen „Storage Cells“ und „Compute Nodes“ ausgetauscht werden können als in den Storage Cells verarbeitet werden

- Flash Cache: Einsatz von Flash Cache und Menge an I/O Requests / Daten verarbeitet mittels Flash Cache

- Storage Indexes: Vermeidung von I/O mittels Storage Indexes

Zur Auswertung der entsprechend relevanten Statistiken bietet sich Tanel Poder’s „ExaSnapper“ Tool an, dass ein Delta der entsprechenden Statistiken automatisch erzeugen und graphisch darstellen kann.

Testumgebungen

Folgende Umgebungen kamen für die Tests zum Einsatz:

Exadata X2-8 Single Compute Node
64 Cores, 128 CPUs, 1TB RAM, 8 Infiniband-Adapter a 40Gbit/sec (theoretische Bandbreite 40GB pro Sekunde zwischen Compute Node und Storage Cells)

14 Storage Cells, 12 CPU Cores pro Storage Cell, insgesamt 168 CPU Cores, 5,3TB Flash Cache, High Capacity Disks

Exadata Database Machine X2-8 Full Rack with High Capacity SAS Disks
Up to 14 GB/second of uncompressed raw disk bandwidth
Up to 50 GB/second of uncompressed Flash data bandwidth
Up to 25,000 Disk IOPS
Up to 1,000,000 Flash IOPS
336 TB of raw disk data capacity
Up to 100 TB of uncompressed user data*
Data Load Rate: Up to 5 TB/hour
Up to 14 GB/second of uncompressed raw disk bandwidth

Laut offiziellem Datenblatt maximal 50GB pro Sekunde „uncompressed Flash data bandwidth“ bei Einsatz von Flash Cache

Exadata X3-8 Two Node RAC
80 Cores, 160 CPUs, 2 TB RAM, 8 Infiniband-Adapter a 40Gbit/sec pro Node (theoretische Bandbreite 40GB pro Sekunde zwischen Compute Node und Storage Cells)

14 Storage Cells, 12 CPU Cores pro Storage Cell, insgesamt 168 CPU Cores, 44TB Flash Cache, High Capacity Disks

Exadata Database Machine X3-8 Key Capacity and Performance Metrics		
Maximum SQL flash bandwidth [1]	100 GB/s	
Maximum SQL flash read IOPS [3]	1,500,000	
Maximum SQL flash write IOPS [4]	1,000,000	
Flash data capacity (raw) [5]	44.8 TB	
Effective Flash cache capacity [7]	Up to 448 TB	
	HC[1] Disks	HP[1] Disks
Maximum SQL disk bandwidth [2]	10 GB/s	24 GB/s
Maximum SQL disk IOPS [3]	32,000	50,000
Disk data capacity (raw) [5]	672 TB	200 TB
Disk data capacity (usable) [6]	300 TB	90 TB
Maximum data load rate [8]	16 TB/hour	

Laut offiziellem Datenblatt maximal 100GB pro Sekunde „Maximum SQL flash bandwidth“ bei Einsatz von Flash Cache

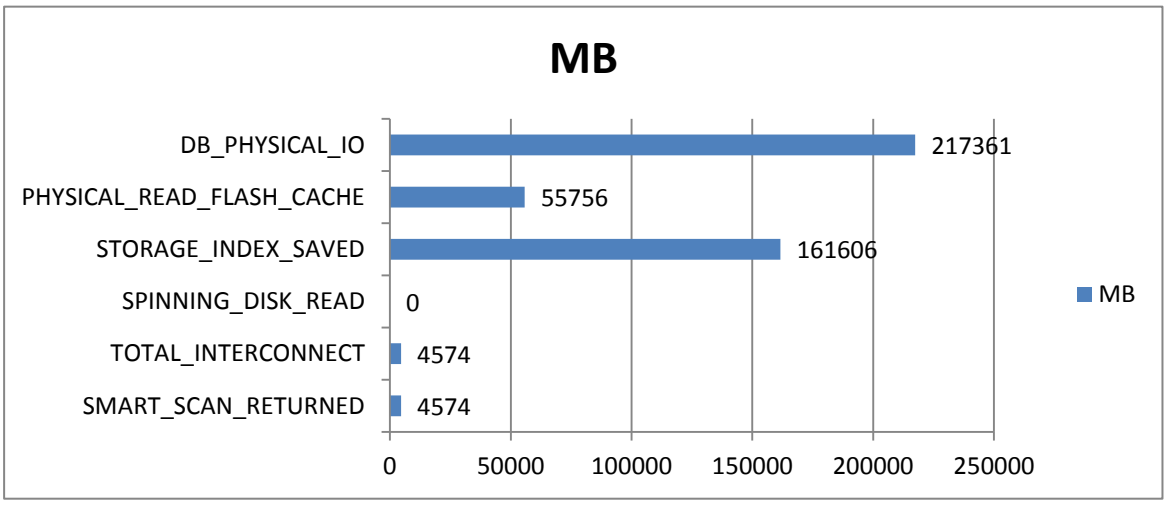
Einfache SELECT COUNT(*)-Abfrage

Folgendes Bild ergab sich für die SELECT COUNT(*)-Abfrage bei einem Parallel DOP von 64 auf der X2-8 und einem DOP von 128 auf der X3-8 und Verwendung von Flash Cache für die Full Table Scans (Auf X3-8 mit aktueller Firmware automatisch gecacht, auf X2-8 mittels CELL_FLASH_CACHE KEEP Storage-Klausel explizit eingeschaltet):

X2-8: 2,54 Sekunden, im Durchschnitt ca. 85GB pro Sekunde gelesen, Peak 90GB pro Sekunde

X3-8: 2 Sekunden, im Durchschnitt ca. 105GB pro Sekunde gelesen, Peak 105GB pro Sekunde

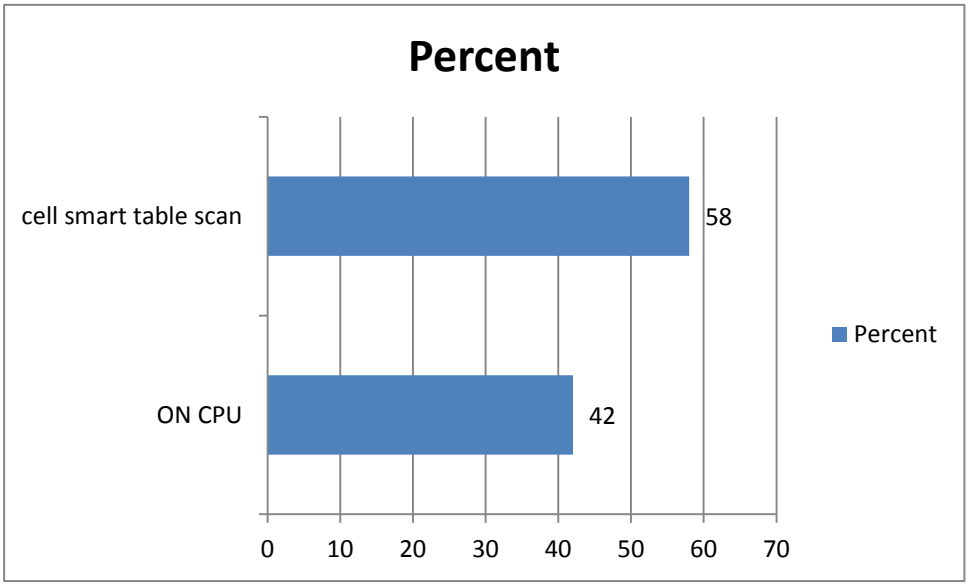
Erstaunlicherweise werden bei der einfachsten Form der Abfrage also (deutlich) höhere Durchsätze erzielt als gemäß Datenblatt spezifiziert. Wie ist das möglich? Handelt es sich um Messfehler oder veraltete Angaben in den Datenblättern? Hier ist der Blick in die oben genannten Statistiken sehr hilfreich für das Verständnis:



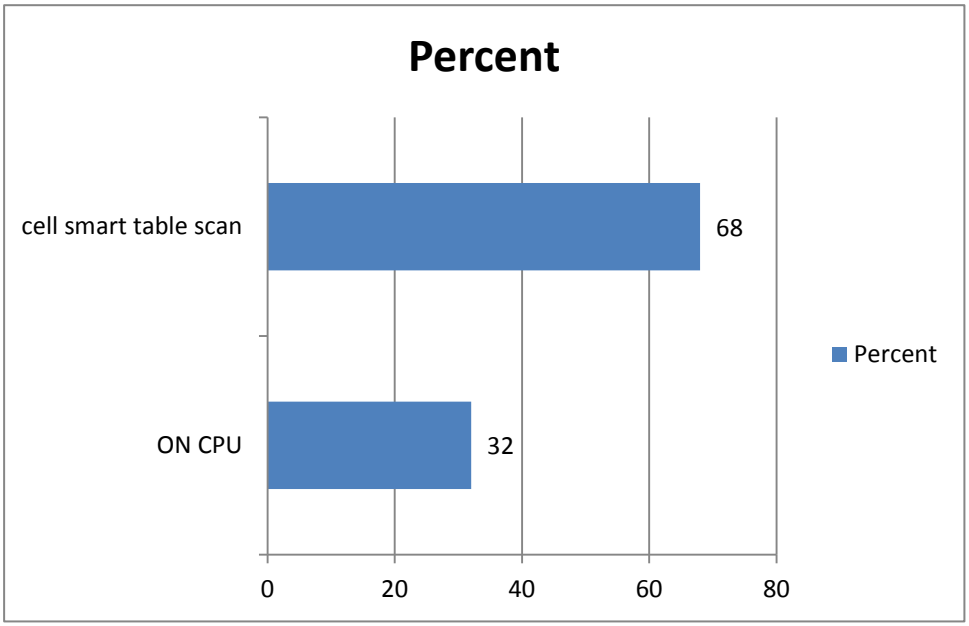
Der Grund für die über dem jeweils angegebenen Maximum liegende Lesegeschwindigkeit ist also offensichtlich die Tatsache, dass nur ca. ein viertel der Daten tatsächlich gelesen wurde, und drei viertel der Daten per Storage Indexes überhaupt nicht gelesen werden mussten.

Desweiteren wird offensichtlich, dass durch den „Smart Scan“ nur ein Bruchteil der tatsächlich in den „Storage Cells“ verarbeiteten Daten an die „Compute Nodes“ zurückgeliefert wurde, hier also ca. 4,5GB der 55 bzw. 217GB, also auch deutlich weniger als die oben genannten 18GB für die eigentliche Abfrage.

Auf der X2-8 Testumgebung ergibt sich folgendes Bild für die Aktivitätsverteilung:

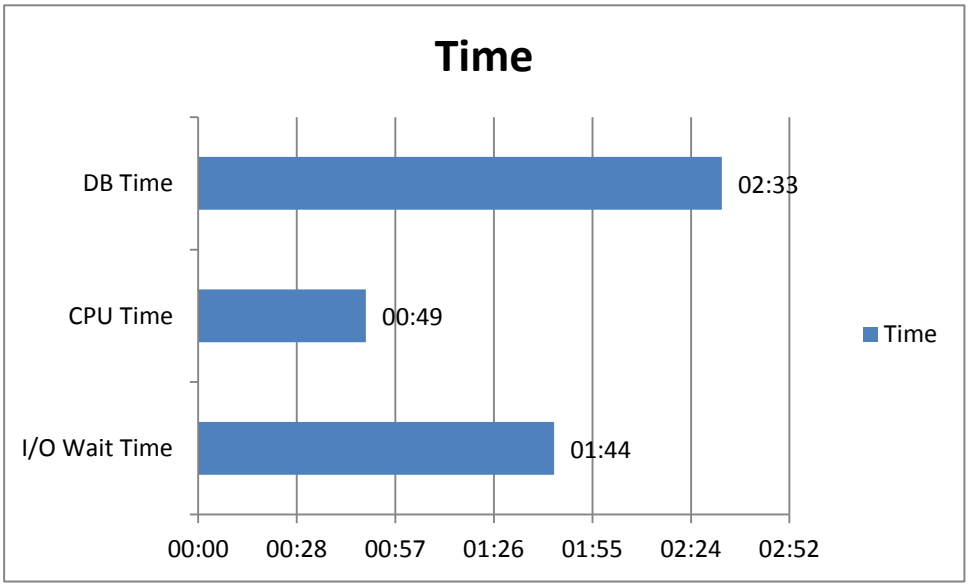


Das Aktivitätsprofil der Abfrage sieht auf der X3-8 Testumgebung folgendermaßen aus:

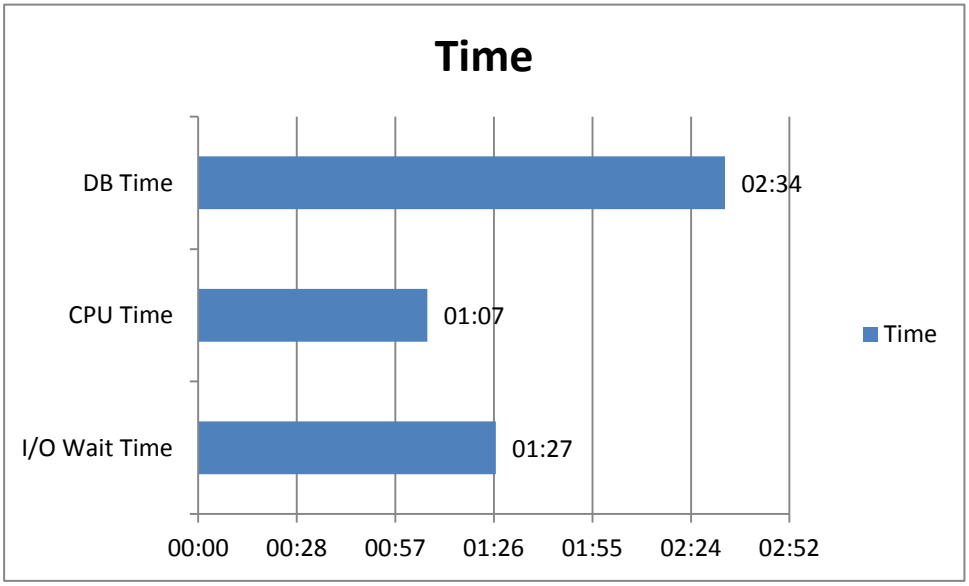


Und folgendes Bild für die Datenbankzeit:

X2-8:



X3-8:



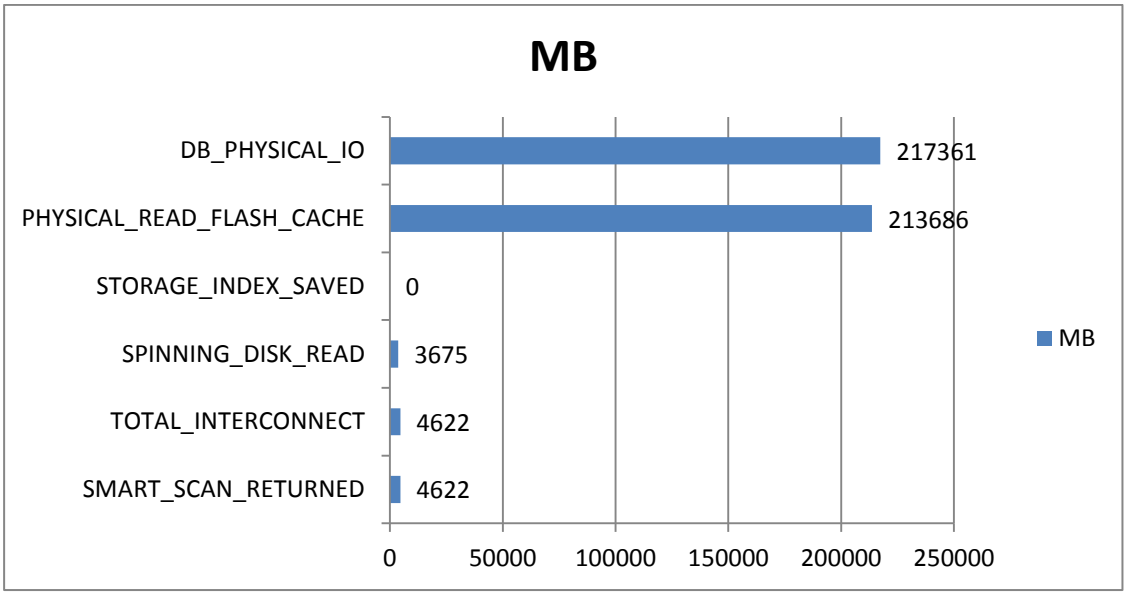
Wird der Einsatz von Storage Indizes verhindert, ergibt sich folgende Performance für die gleiche Art der Abfrage:

X2-8: 5,1 Sekunden, im Durchschnitt ca. 43GB pro Sekunde gelesen, Peak 45GB pro Sekunde

X3-8: 3,1 Sekunden, im Durchschnitt ca. 70GB pro Sekunde gelesen, Peak 75GB pro Sekunde

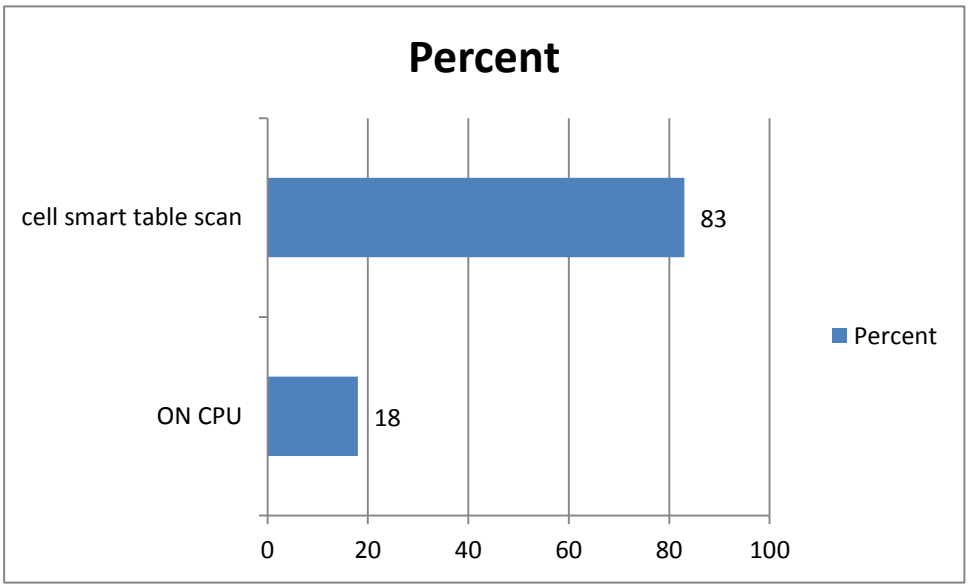
Hier werden also die laut Datenblatt erreichbaren Scan-Geschwindigkeiten nicht mehr ganz erreicht.

Die Unwirksamkeit der Storage Indizes in diesem Fall kann bestätigt werden, wenn wir wiederum auf die Session Statistiken schauen:

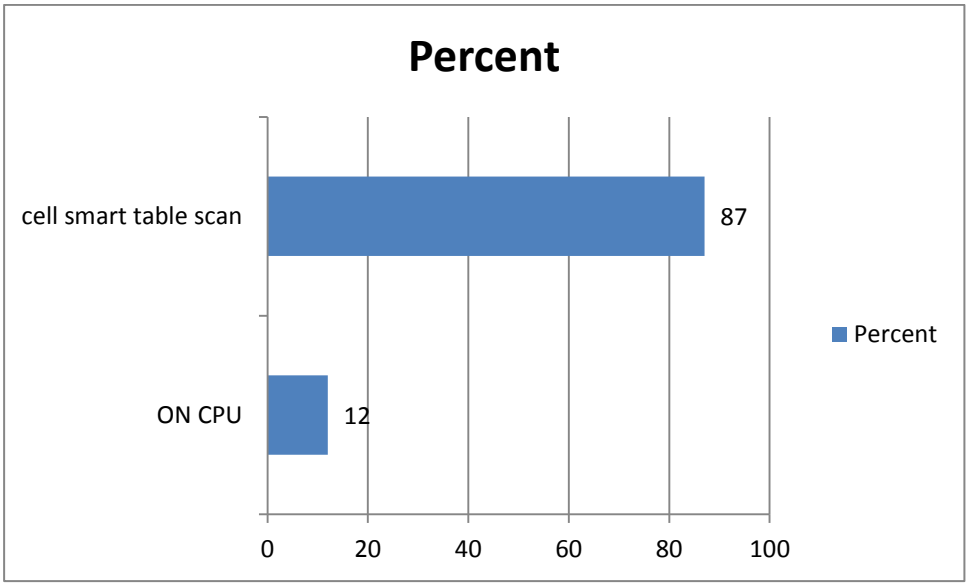


Im Gegensatz zum vorherigen Beispiel müssen jetzt also tatsächlich alle 217GB (in den Zellen) gelesen werden. Auch das Aktivitätsprofil verändert sich entsprechend:

X2-8:

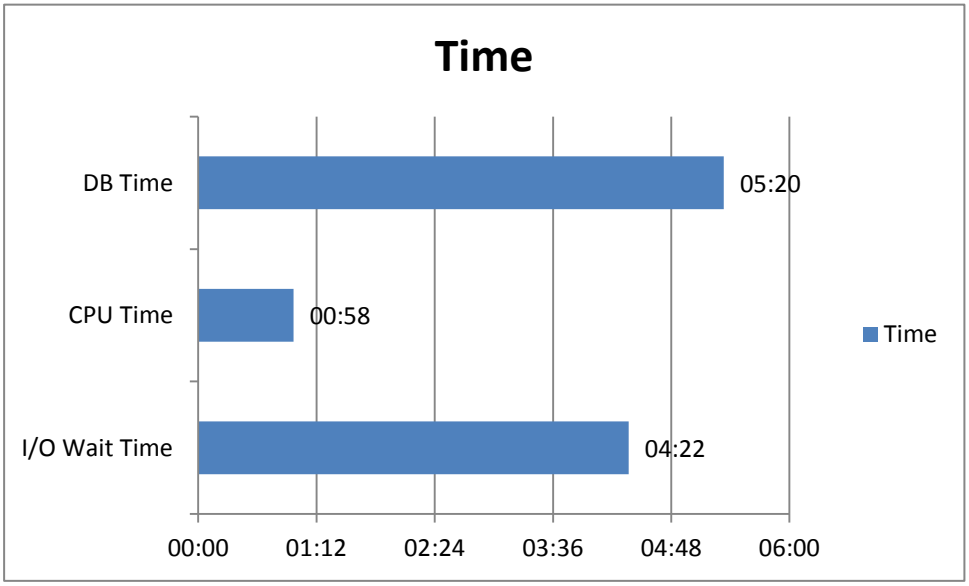


X3-8:

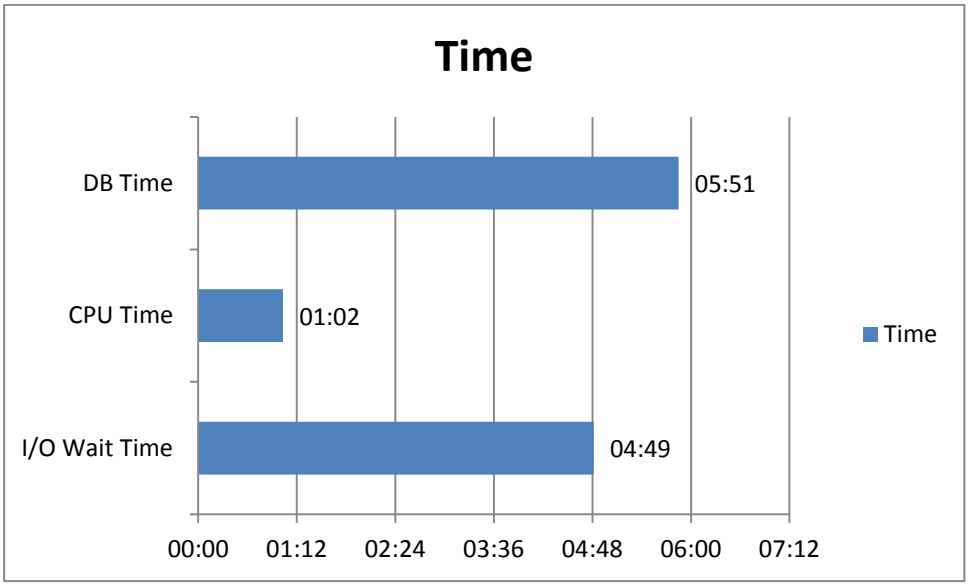


Und eine entsprechende Verschiebung / Erhöhung der Datenbankzeit kann ebenso gesehen werden:

X2-8:



X3-8:



Es wird jetzt also deutlich mehr auf I/O gewartet als bei der Verwendung von Storage Indizes. Die Gesamtdatenbankzeit erhöht sich entsprechend.

Einfache SELECT MAX()-Abfrage

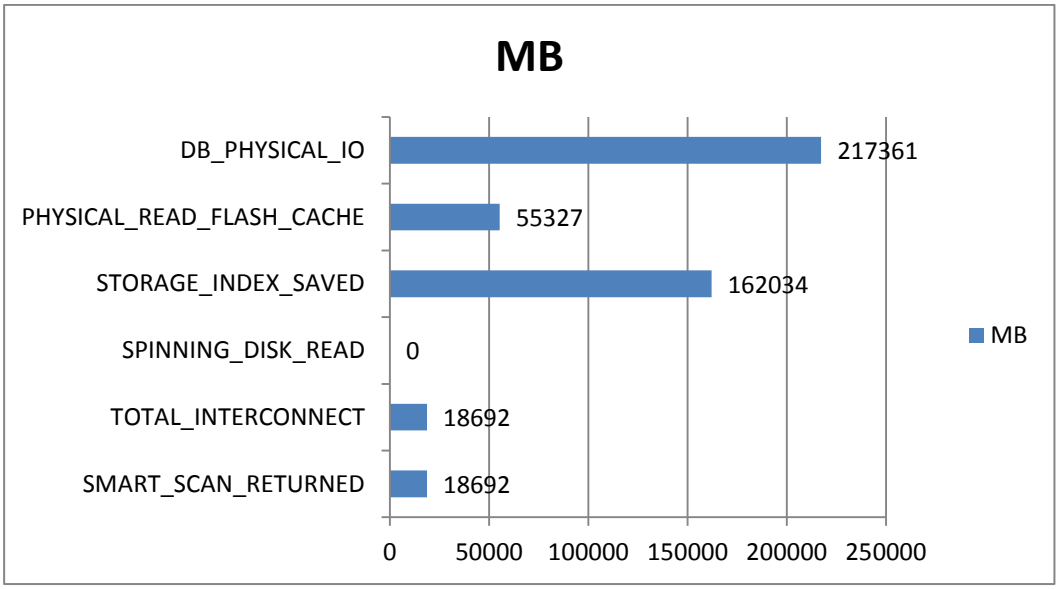
Als nächste Komplexitätsstufe wird die COUNT(*)-Abfrage in eine SELECT MAX()-Abfrage verändert. Dabei werden alle Spalten mit MAX(COLUMN) verwendet, die auch bei der Join-Abfrage sowohl bei Selektion, als auch Projektion zur Verwendung kommen. Dabei ergaben sich folgende Laufzeiten bei Verwendung von Storage Indizes, also zu vergleichen mit dem ersten SELECT COUNT(*)-Beispiel:

X2-8: 7,0 Sekunden, im Durchschnitt ca. 31GB pro Sekunde gelesen, Peak 31GB pro Sekunde

X3-8: 3,7 Sekunden, im Durchschnitt ca. 58GB pro Sekunde gelesen, Peak 60GB pro Sekunde

Im Vergleich zu der einfachen SELECT COUNT(*)-Abfrage unter Verwendung von Storage Indizes ist hier eine deutliche Verlängerung der Laufzeit zu beobachten, als auch eine verringerte Lesegeschwindigkeit auf I/O-Seite.

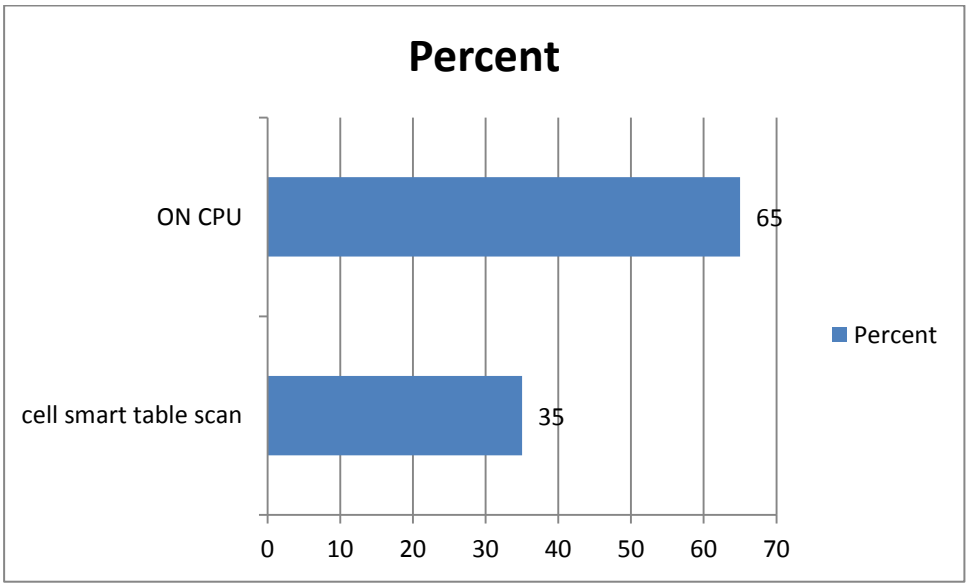
Die Analyse der Session Statistiken zeigt folgendes Bild:



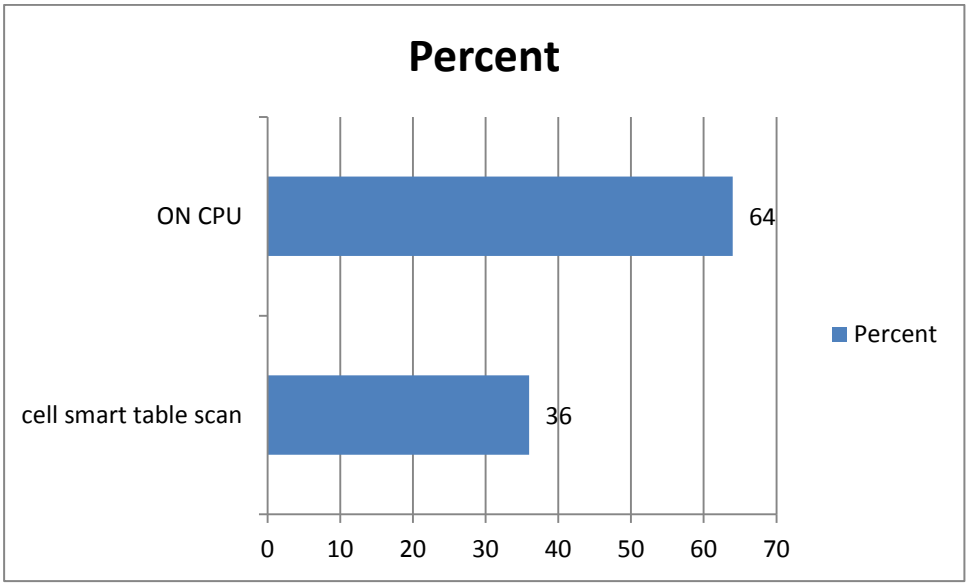
Wir können also sehen, dass die Storage Indizes tatsächlich erfolgreich verwendet wurden. Im Unterschied zur SELECT COUNT(*)-Abfrage wurden aber anstatt 4,5GB jetzt 18GB vom Smart Scan an die Compute Nodes geliefert, da für die Auswertung der MAX()-Funktionen die tatsächlichen Spaltenwerte benötigt werden, im Gegensatz zur Auswertung einer COUNT(*)-Funktion.

Die erhöhte Komplexität der Aktivität in den Compute Nodes kann auch im Aktivitätsprofil gesehen werden:

X2-8:



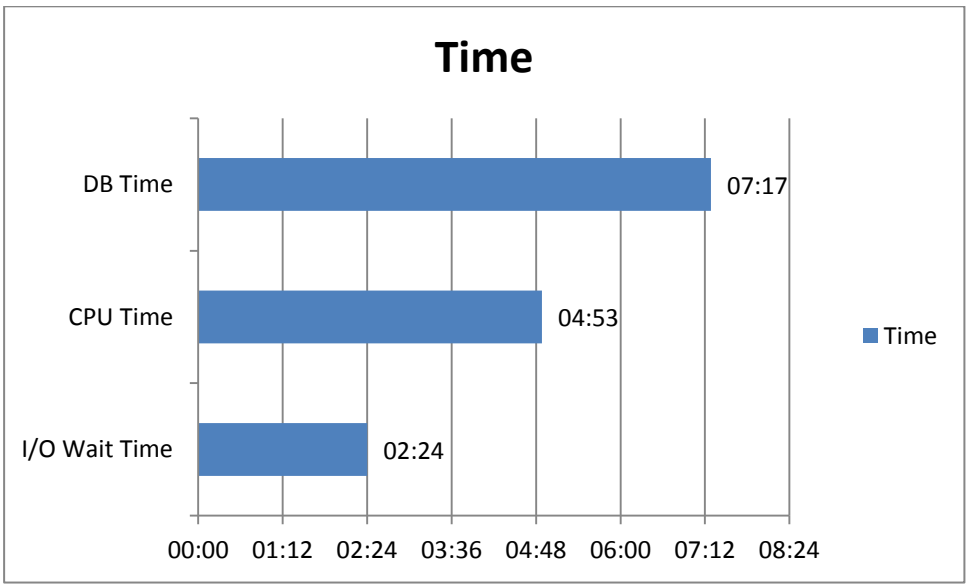
X3-8:



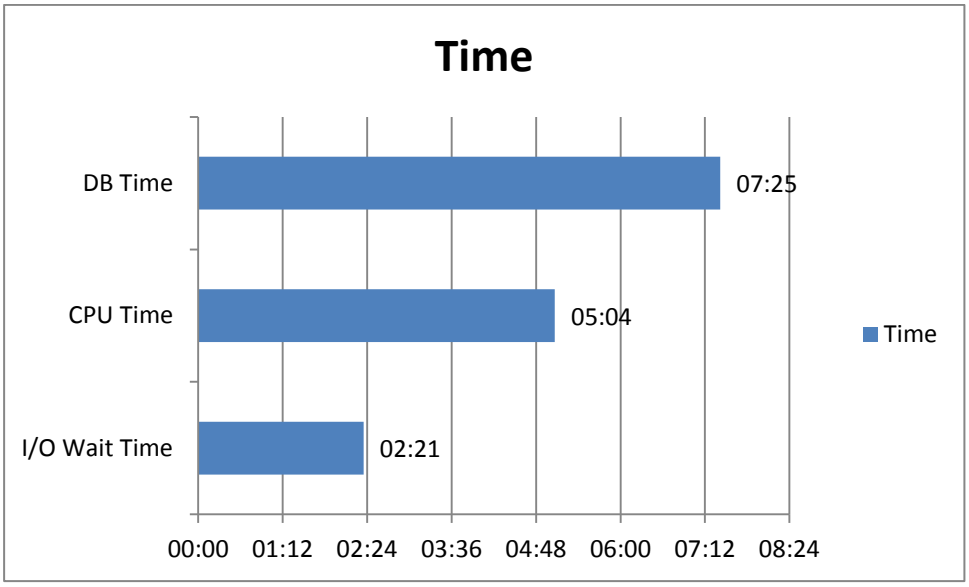
Es wird jetzt also prozentual deutlich mehr Zeit auf CPU verbracht als beim SELECT COUNT(*).

Die entsprechende Verlängerung und Verschiebung der Datenbankzeit sieht folgendermaßen aus:

X2-8:



X3-8:



Interessanterweise erhöht sich nicht nur die CPU-Zeit, sondern auch die I/O-Wait Zeit, allerdings nicht so signifikant wie die CPU-Zeit.

Die eigentliche Abfrage

Als letzte Stufe der Komplexitätserhöhung wird jetzt anstelle eines UNION ALL ein (Outer) Join der 20 Tabellen durchgeführt, und auf das Ergebnis des Joins eine SELECT MAX()-Abfrage der Projektion durchgeführt. Im Gegensatz zu der UNION ALL-Abfrage hat der Join nicht mehr 400 Millionen Zeilen als Ergebnis, sondern nur noch 20 Millionen. Allerdings muss immer noch die gleiche Datenmenge (20 mal 20 Millionen, ca. 18GB netto) verarbeitet werden. Es sind jetzt in der Projektion 20 mal so viele Spalten wie bei der UNION ALL-Abfrage.

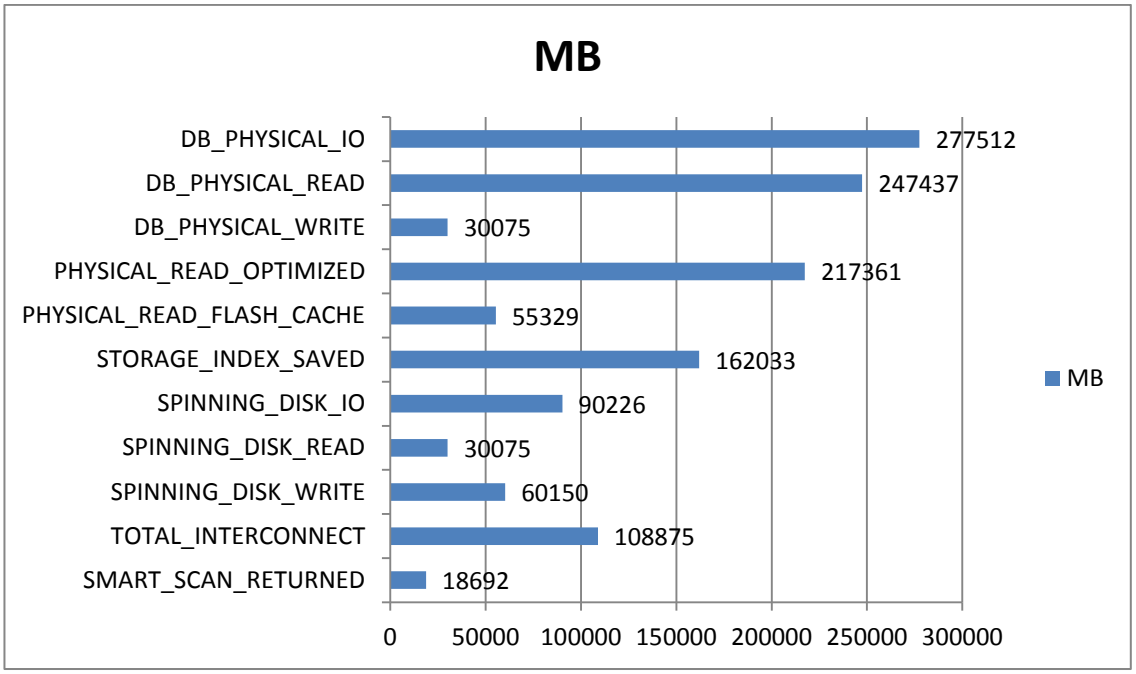
Dabei ergeben sich folgende Laufzeiten:

X2-8: 39,3 Sekunden, im Durchschnitt ca. 5,5GB (7GB) pro Sekunde gelesen, Peak 28GB pro Sekunde

X3-8: 23,7 Sekunden, im Durchschnitt ca. 9GB (12GB) pro Sekunde gelesen, Peak 27GB pro Sekunde

Das ist erst mal ein extremer Unterschied zu den vorherigen Zahlen – insbesondere sind auch die Lesegeschwindigkeiten weit davon entfernt, was die Plattform grundsätzlich in der Lage ist zu liefern. Tatsächlich sind die Lesegeschwindigkeiten in Bereichen, die von den Festplatten geliefert werden können, wir also möglicherweise hier nicht mehr wirklich vom Einsatz des Flash Caches profitieren.

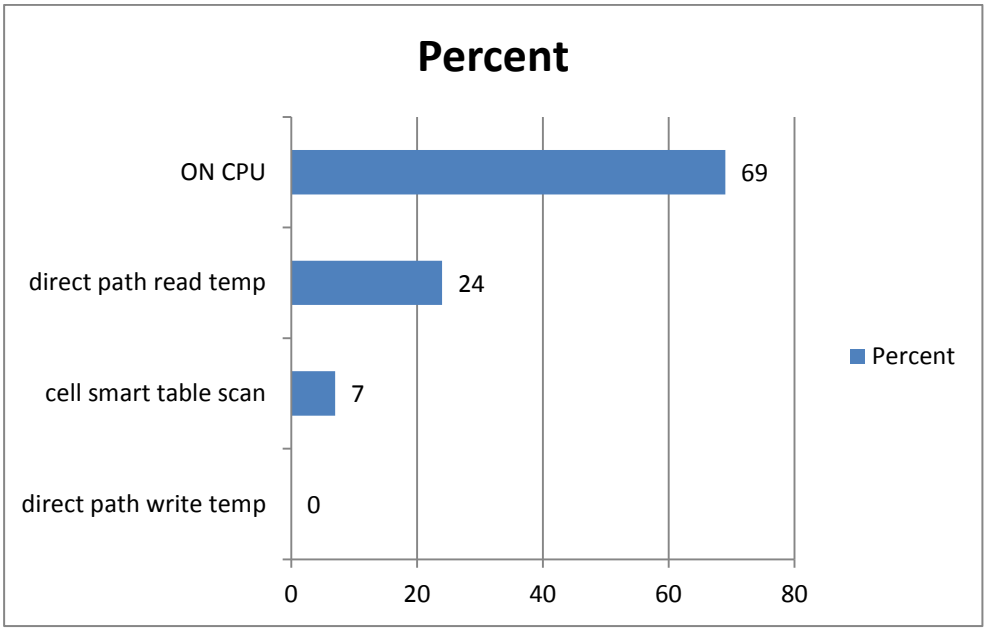
Die Laufzeiten und die Zahlen in Klammern bezüglich der Lesegeschwindigkeit oben werden klarer, wenn wir auf die entsprechenden Session-Statistiken schauen:



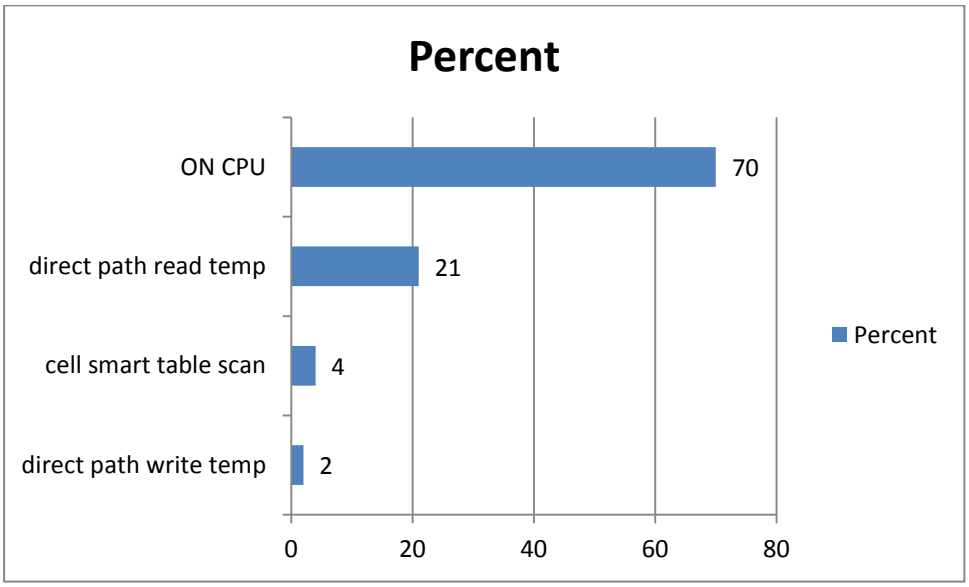
Hier wird also deutlich, dass deutlich mehr I/O erzeugt wurde als in den vorherigen Beispielen. Insbesondere mussten 30GB geschrieben werden, was durch die ASM-Spiegelung zu 60GB verdoppelt wird (Normal Redundancy), und diese 30GB werden dann auch wieder gelesen. Das bedeutet auch, dass jetzt 90GB mehr zwischen den Compute Nodes und den Storage Cells ausgetauscht werden müssen – ein Anstieg von 18GB auf 108GB. Trotzdem haben wir im Grunde das gleiche Profil wie vorher – wir sparen 162GB an I/O über Storage Indizes und lesen 55GB mittels Flash Cache per Smart Scan, der 18GB an die Compute Nodes zurückliefert. Diese Optimierung wird aber durch die zusätzliche Schreib- und Leseaktivität deutlich beeinflusst / verschlechtert.

Diese Statistiken lassen erst mal vermuten, dass das zusätzliche I/O durch TEMP-Aktivität erzeugt wird. Die entsprechenden Aktivitätsprofile bestätigen dies und sehen so aus:

X2-8:

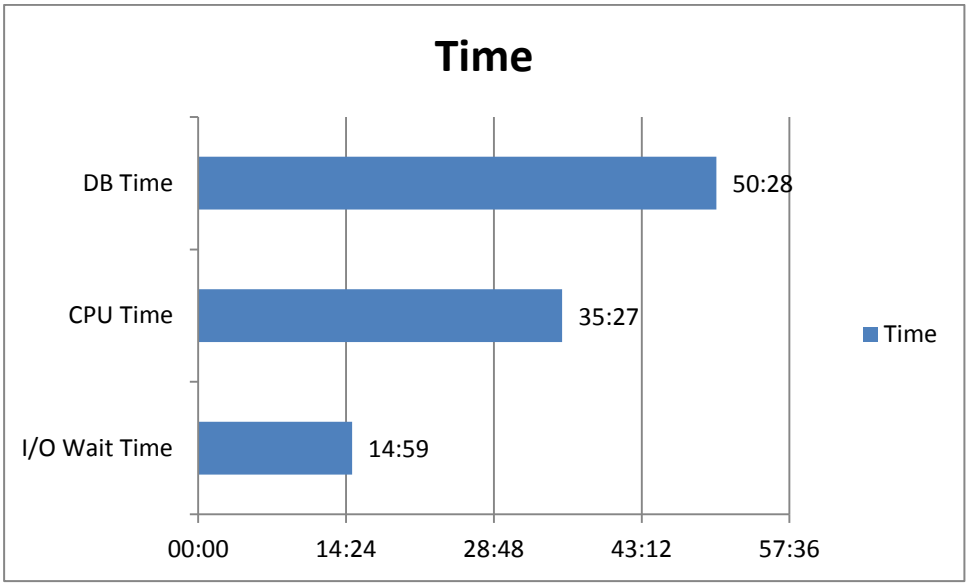


X3-8:

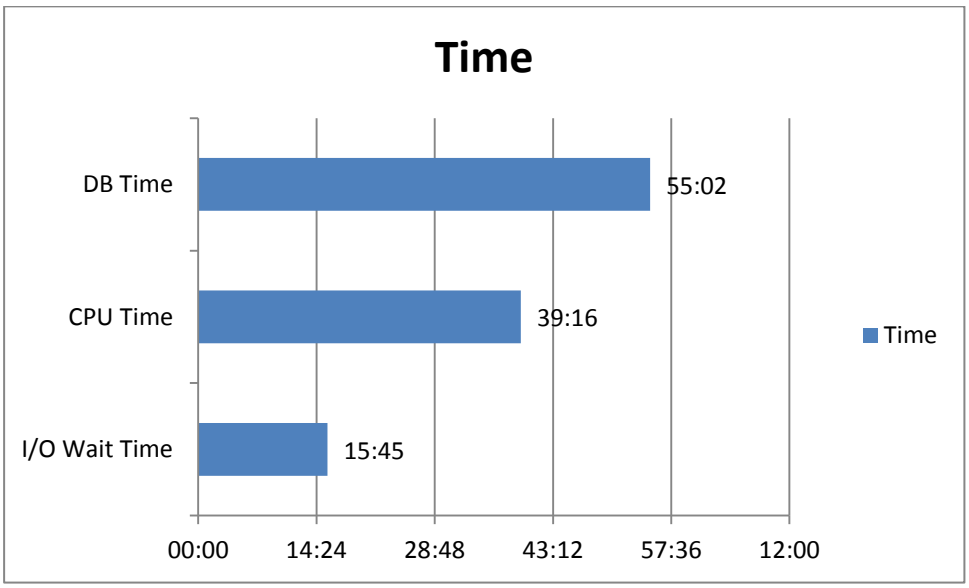


Die massive Erhöhung der getätigten Arbeit zeigt sich auch in den Auswertungen der Datenbankzeit:

X2-8:



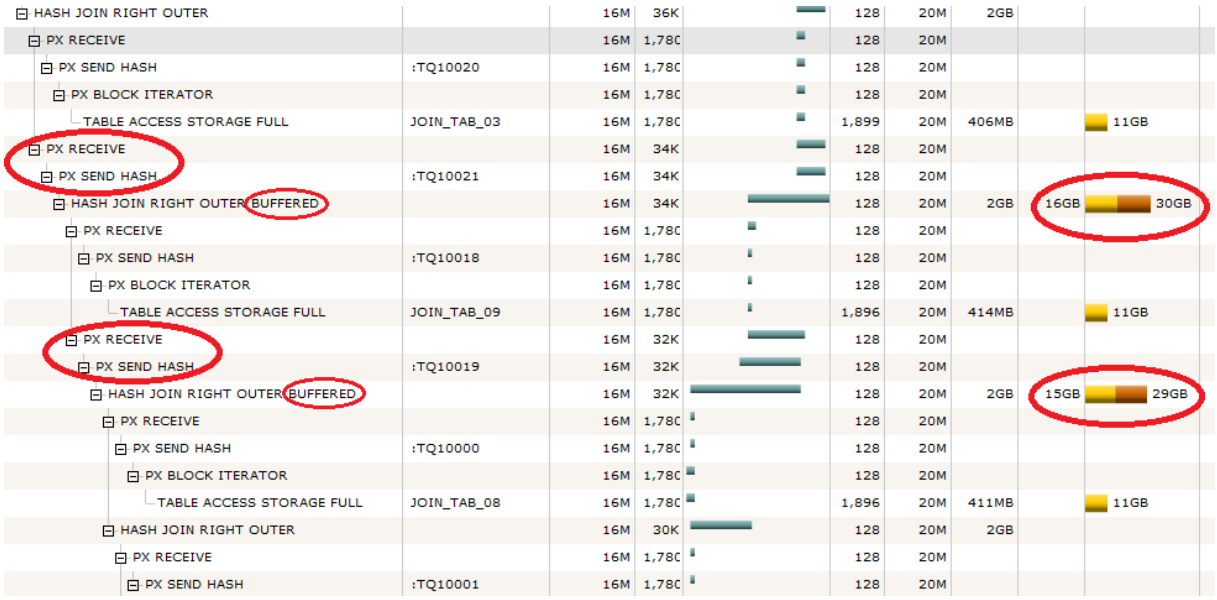
X3-8:



Wir reden hier also zum Beispiel bei der X3-8 von einer Erhöhung von 7 auf 55 Minuten!

Die Frage ist hier jetzt natürlich, was genau verursacht das zusätzliche TEMP-I/O und die Erhöhung der Datenbankzeit?

Ein Blick in den entsprechenden Real-Time SQL Monitoring Report zeigt folgendes:



Im Vergleich zu den HASH JOINs darüber und darunter benötigen die beiden HASH JOIN BUFFERED jeweils 15GB an TEMP, was gelesen und geschrieben wird und die zusätzliche I/O-Menge erklärt.

Was ist also das Besondere an diesen „BUFFERED“ HASH JOINs? Dazu muss man wissen, dass Oracle für die Implementierung von Parallel Execution das sogenannte Producer-Consumer Modell einsetzt. Hier sind die doppelte Menge an Parallel Execution Servern im Einsatz – bei einem DOP von 128 also 256 Stück – und diese agieren in zwei Sets mit jeweils 128 PX Servern. Wenn eine Redistribution der Daten laut Ausführungsplan notwendig ist (PX SEND / PX RECEIVE mittels einer sogenannten Table Queue (:TQxxxx im Plan)), schickt das eine PX Server Set die Daten an das andere PX Server Set, das die Daten empfängt.

Nun kann es Situationen laut Ausführungsplan geben, in denen mehrere dieser PX SEND / PX RECEIVE-Operationen gleichzeitig aktiv wären, also mehrere Redistributionen der Daten zur gleichen Zeit notwendig wären – und dies ist von Oracle nicht implementiert / unterstützt. Es kann also nur eine Redistribution der Daten gleichzeitig aktiv sein, und das ist genau der Grund und Zweck der HASH JOIN BUFFERED-Operationen: Ein non-BUFFERED HASH JOIN agiert so, dass er zuerst die erste Datenquelle einliest und das Hash Table gemäß der Join Keys erzeugt (idealerweise passt das Hash Table in den PGA-Speicher, wenn nicht, dann muss auf TEMP auslagert werden und der Hash Join wird zu einem One-Pass oder Multi-Pass Hash Join). Nachdem das Hash Table gebaut ist, kann die zweite Datenquelle verarbeitet werden. Dabei wird jeweils ein Lookup im Hash Table durchgeführt, und das Ergebnis kann sofort an die folgende Operation weitergereicht werden.

Der BUFFERED Hash Join ändert dieses Verhalten so ab, dass nach dem Verarbeiten der ersten Datenquelle, also nach dem Erzeugen des Hash Tables, die zweite Datenquelle gelesen und zwischengepuffert wird, ohne den tatsächlichen Join auszuführen. Dieser Schritt kann insofern optimiert werden, dass Oracle ein Bitmap aus der Verarbeitung der ersten Datenquelle vorliegt, die anzeigt, ob ein Hash Bucket Daten beinhaltet oder nicht. Sollten keine Daten in dem Hash Bucket aus der ersten Datenquelle liegen, können Daten aus der zweiten Datenquelle gleich verworfen werden, die diesem Hash Bucket zugeordnet werden, und müssen nicht zwischengepuffert werden (es gibt Ausnahmen wie den Right Outer Join, bei dem diese Optimierung nicht möglich ist, da alle Zeilen der zweiten Datenquelle laut Outer Join „überleben“).

Dieses „Zwischenpuffern“ der Daten sorgt also dafür, dass die entsprechende PX SEND / RECEIVE-Operation ausgeführt wird, bevor der eigentliche Join stattfindet, also bevor das Ergebnis des Joins an die folgende Operation weitergegeben werden kann. Natürlich bedeutet so ein Zwischenpuffern, dass zusätzlich PGA-Speicher benötigt wird, um die zweite Datenquelle des Hash Joins vorzuhalten. Je nach Datenmenge kann das zu massiver TEMP-Aktivität führen, falls der PGA-Speicher nicht zum Zwischenpuffern ausreicht.

Es ist nicht einfach zwischen der TEMP-Aktivität durch Bauen des Hash Tables und TEMP-Aktivität aufgrund des Zwischenpufferns zu unterscheiden, wobei Oracle hier durchaus Informationen zu den sogenannten „Workareas“ zur Verfügung stellt (Optimal / One-Pass / Multi-Pass, aber nicht im Monitoring-Report angezeigt). Wir können aber aufgrund des obigen Monitor- Reports davon ausgehen, dass die Hash Tables selbst in den PGA-Speicher passen, da sowohl der vorherige, als auch der nachfolgende HASH JOIN (ohne BUFFERED) keine TEMP-Aktivität verursachen und mit jeweils 2GB PGA-Speicher auskommen.

Der Test Case wurde extra so gebaut, dass die Auswirkung der HASH JOIN BUFFERED genauer untersucht werden kann, in dem durch entsprechende Änderung des Joins-Kriteriums die Anzahl der HASH JOIN BUFFERED kontrolliert werden kann, ohne das Ergebnis der Abfrage zu beeinflussen. Dies ist in einem Real-Life Szenario so einfach nicht möglich – wenn entsprechende Redistributionen notwendig sind, werden auch entsprechende HASH JOIN BUFFERED-Operationen vom Optimizer erzeugt werden. Man kann dies also normalerweise nur indirekt beeinflussen, in dem man die Menge der notwendigen Redistributionen beeinflusst, und dies ist in den allermeisten Fällen nur über den Einsatz von entsprechender Partitionierung der Daten möglich. Später dazu mehr.

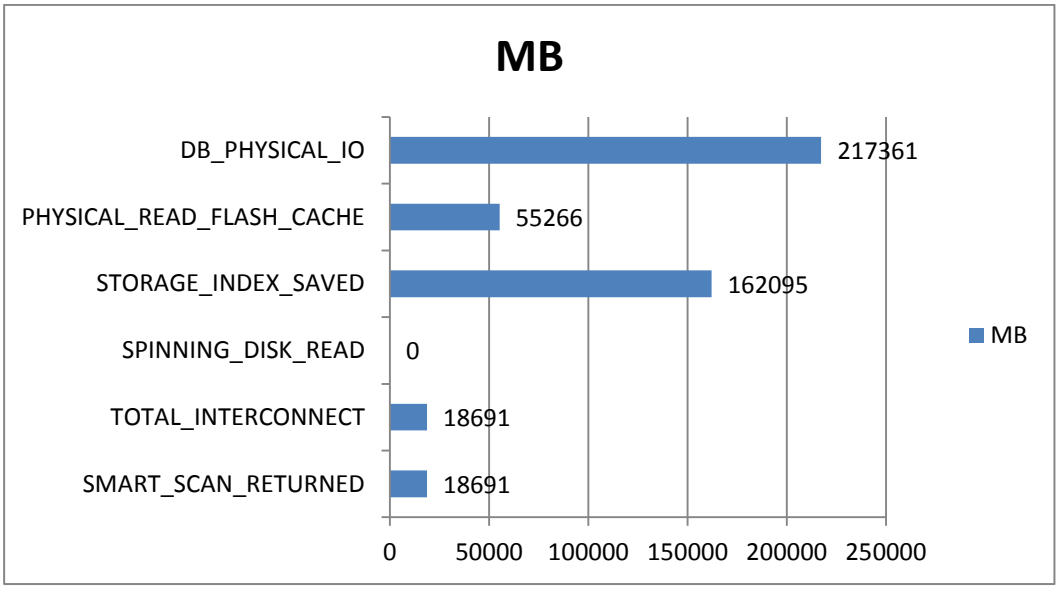
Hier die entsprechenden Ergebnisse eines Best-Case (kein HASH JOIN BUFFERED) und eines Worst-Case (maximale Anzahl an HASH JOIN BUFFERED):

Best-Case:

X2-8: 22,83 Sekunden, im Durchschnitt ca. 10GB pro Sekunde gelesen, Peak 27GB pro Sekunde

X3-8: 12,81 Sekunden, im Durchschnitt ca. 17GB pro Sekunde gelesen, Peak 35GB pro Sekunde

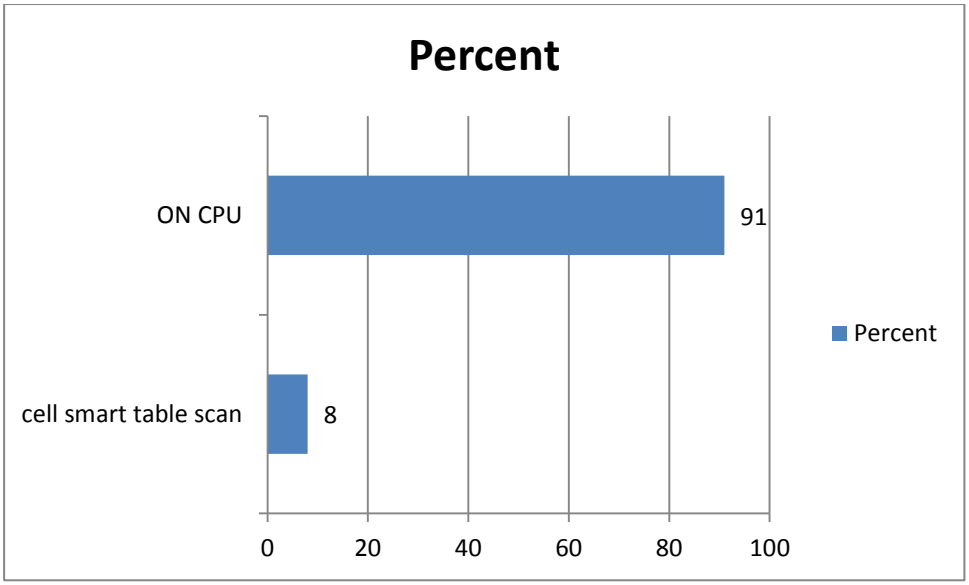
Mit folgenden Session Statistiken:



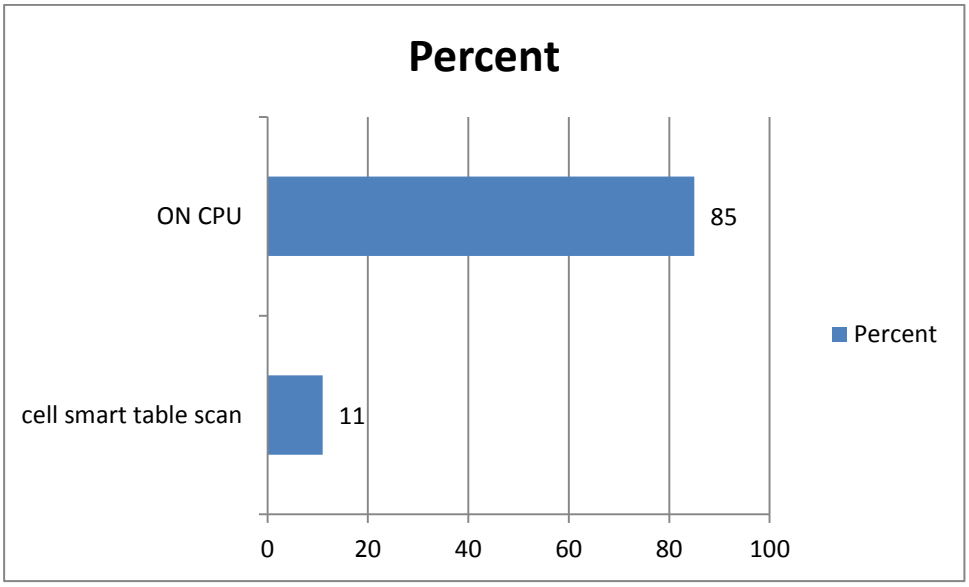
Hier haben wir also ein recht vergleichbares Profil mit der vorherigen einfachen SELECT MAX()-Abfrage, aber immer noch eine deutlich verlängerte Ausführungszeit.

Die Aktivitätsprofile sehen folgendermaßen aus:

X2-8:

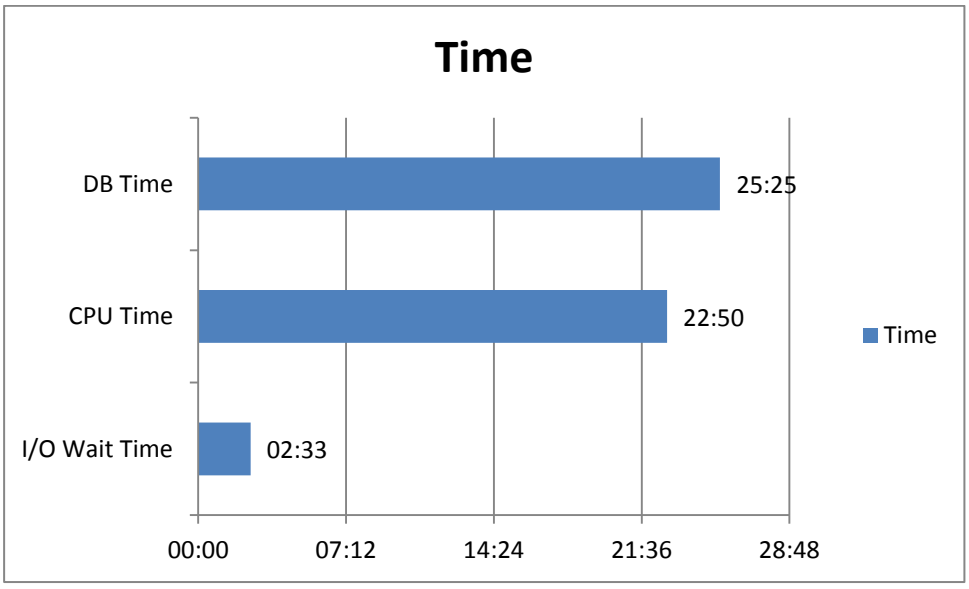


X3-8:

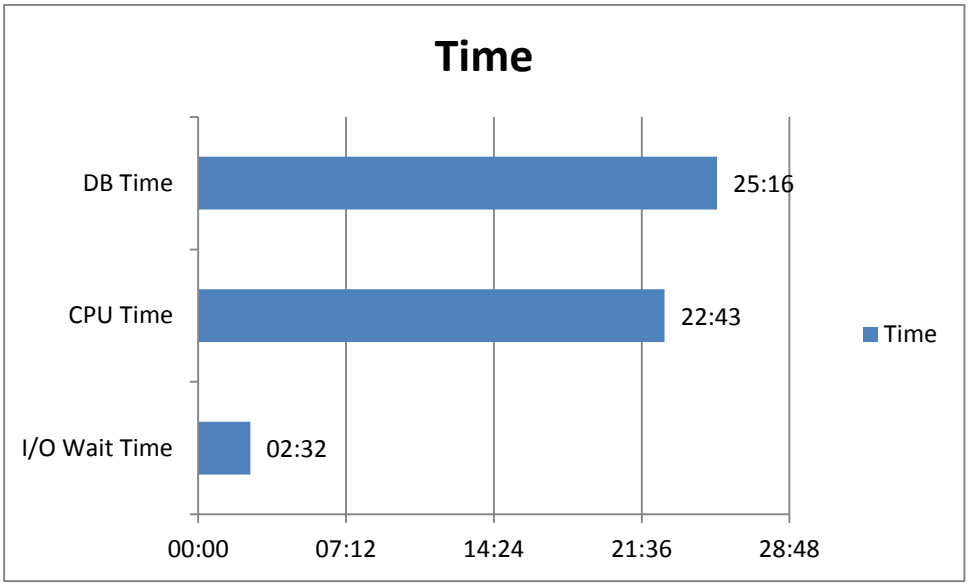


Und die Datenbankzeit folgendermaßen:

X2-8:



X3-8:



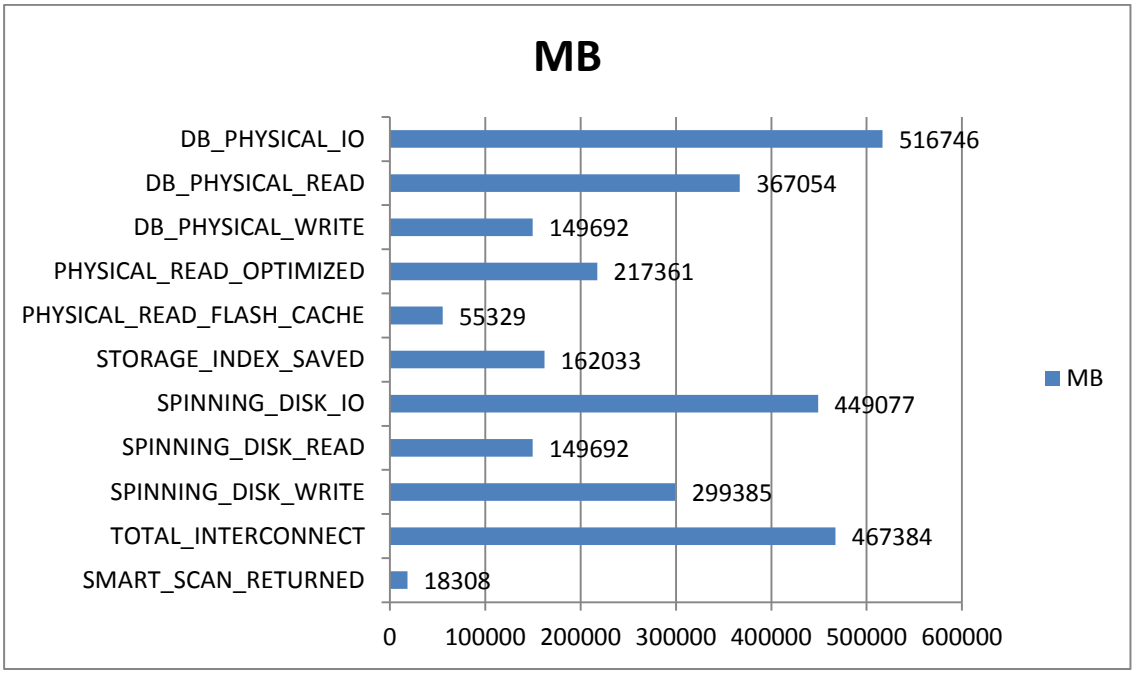
Wir reden hier also fast von einer Halbierung der Laufzeit gegenüber der Original-Abfrage, einzig allein durch den Unterschied, dass zwei HASH JOIN BUFFERED durch non-BUFFERED-Varianten ersetzt wurden, und dadurch kein TEMP-I/O benötigt wird, aber offensichtlich auch deutlich weniger CPU-Zeit verbraucht wird.

Worst-Case:

X2-8: 123,0 Sekunden, im Durchschnitt ca. 1,8GB (4,2GB) pro Sekunde gelesen, Peak 15GB pro Sekunde

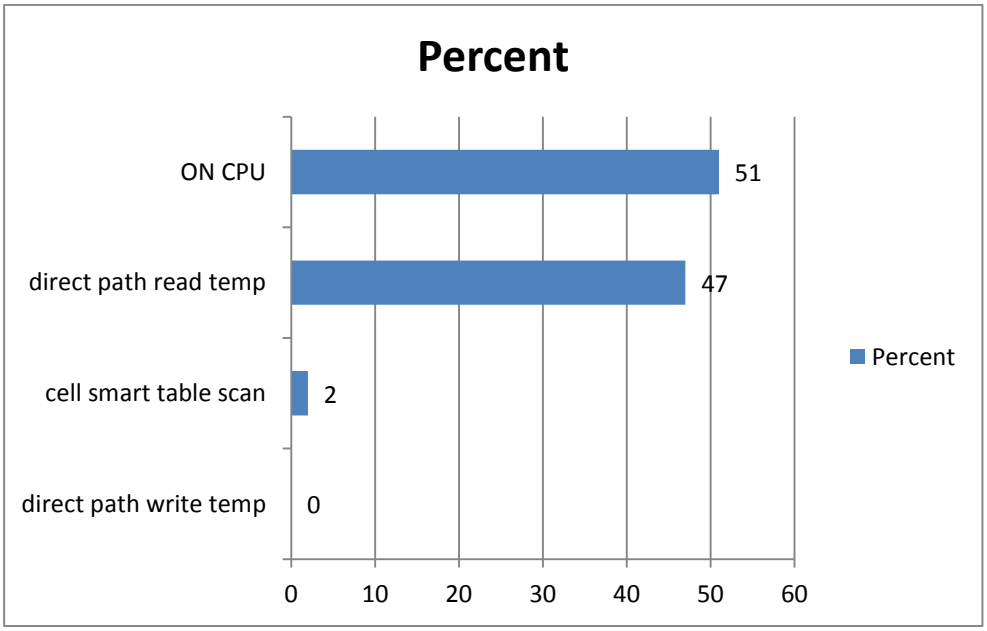
X3-8: 78,9 Sekunden, im Durchschnitt ca. 2,8GB (6,5GB) pro Sekunde gelesen, Peak 17GB pro Sekunde

Die Session Statistiken deuten an, welche Mehrarbeit im Worst-Case Szenario notwendig ist:

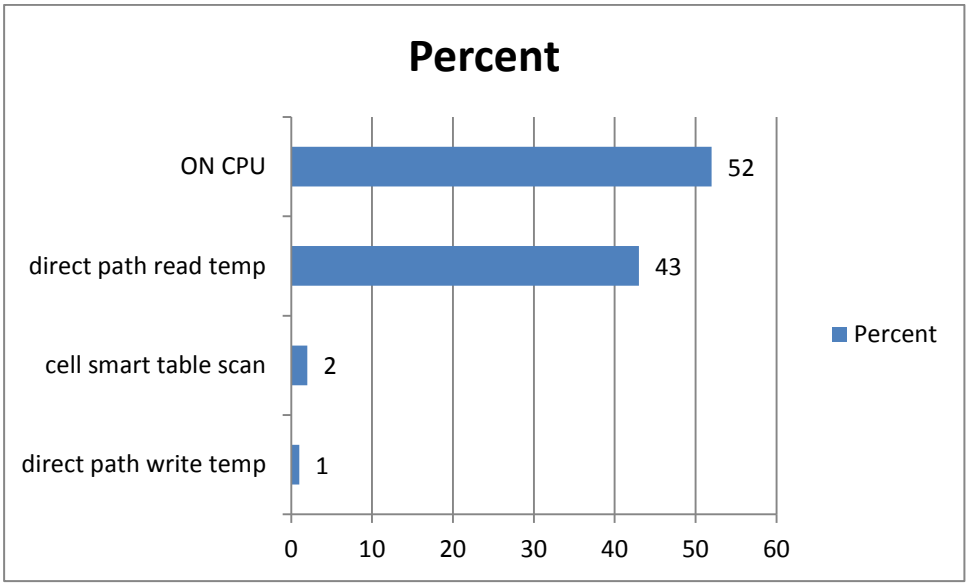


Das Aktivitätsprofil sieht folgendermaßen aus:

X2-8:

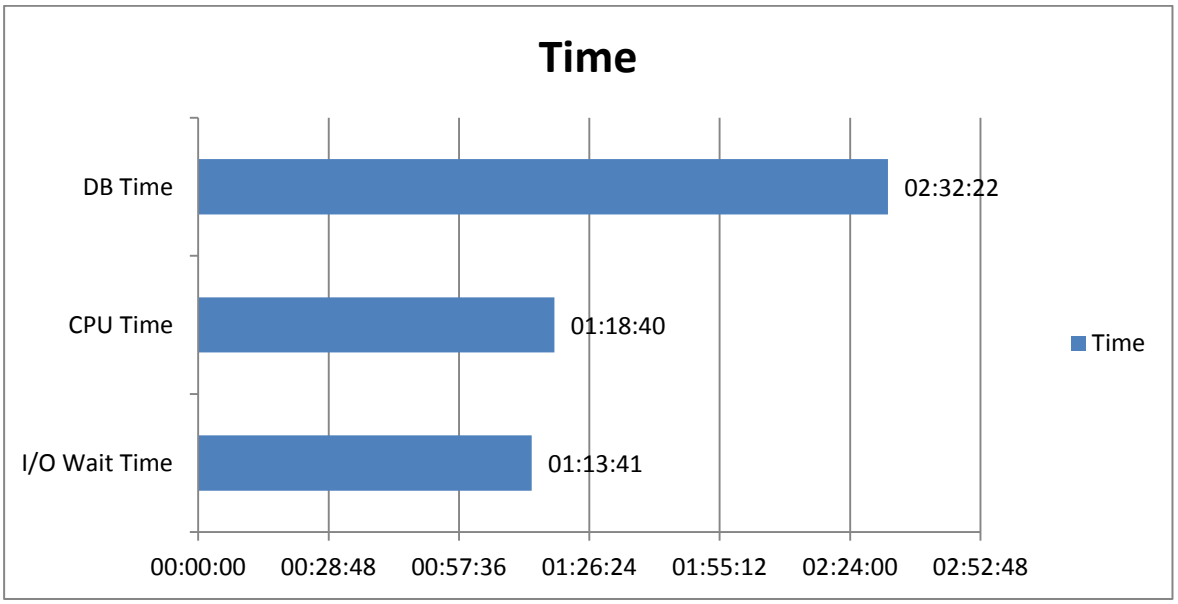


X3-8:

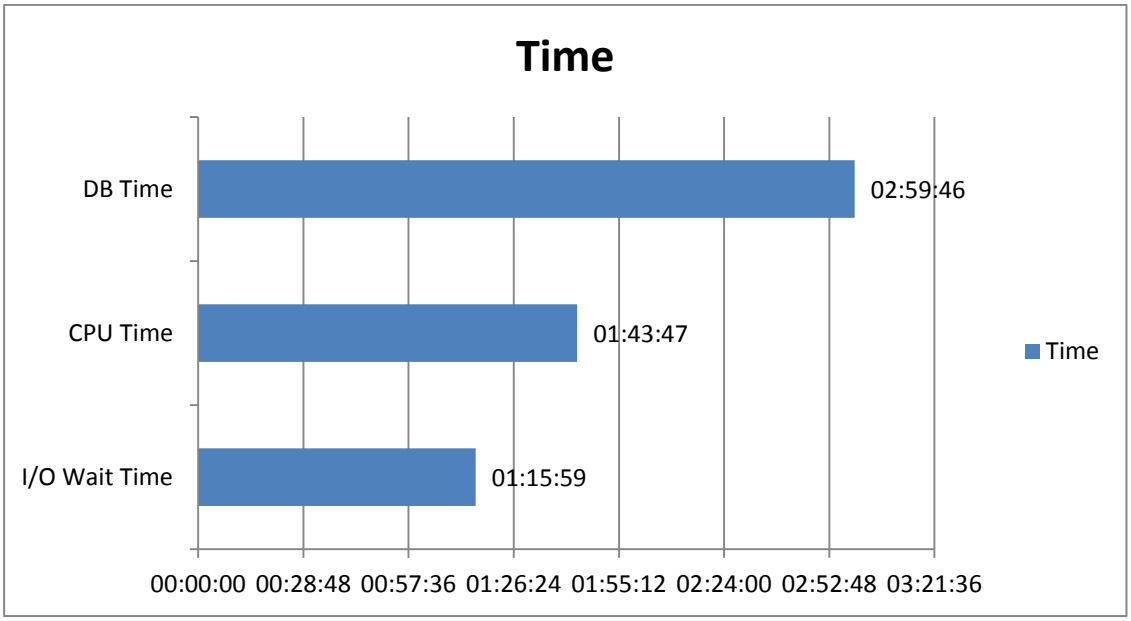


Und die Datenbankzeit sieht folgendermaßen aus:

X2-8:



X3-8:



Bei der Laufzeit zwischen Best-Case und Worst-Case liegen hier auf der X2-8 Umgebung 100 Sekunden, und auf der X3-8 Umgebung über 60 Sekunden. Wir reden hier also auf der X3-8 von einem Unterschied zwischen 25 und 180 Minuten Datenbankzeit zwischen Best-Case und Worst-Case (25 vs. 150 Minuten auf der X2-8), und zwischen 212GB und 516GB an I/O-Menge aus Datenbank-Sicht!

Wichtig ist zu verstehen, dass wir hier bei Best-Case und Worst-Case von der gleichen Grunddatenmenge reden, die zu verarbeiten ist (18GB insgesamt), und die auch als Ergebnis des Joins entsteht, daran ändert sich nichts. Was sich ändert, ist die Join-Bedingung, so dass eine Umverteilung der Daten nach jedem Join notwendig wird. Diese zusätzlichen Umverteilungen benötigen Resourcen (maßgeblich CPU-Zeit, aber auch Speicher und eventuell Netzwerk im Falle von RAC). Darüber hinaus verursachen diese zusätzlichen Umverteilungen aufgrund der beschriebenen Implementierungsbeschränkung mehr BUFFERED-Operationen, was wiederum mehr CPU-Zeit und mehr PGA-Speicher für die Zwischenpufferung der Daten benötigt.

Die Anzahl der HASH JOIN BUFFERED hat also maßgeblich Einfluss auf die Performance. Man kann normalerweise, wie bereits erwähnt, die Erzeugung der BUFFERED Operationen nicht direkt beeinflussen, insofern ist der Testfall hier nicht realistisch. Allerdings kann die Distribution der Daten beeinflusst werden, und damit indirekt die Notwendigkeit für BUFFERED Operationen.

Aber auch das Best-Case Szenario ist immer noch weit entfernt von den Laufzeiten der UNION ALL-Abfragen, obwohl im Grunde die gleiche Datenmenge zu verarbeiten ist. Natürlich ist eine Join-Operation deutlich komplexer als eine UNION ALL-Operation. Es stellt sich die Frage, ob diese Joins noch weiter optimiert werden können.

Wenn wir die Aktivitätsprofile der Best-Case Abfrage von oben anschauen, wird klar, dass die meiste Zeit in den Compute Nodes auf CPU verbracht wird – die HASH JOINs sind also sehr CPU-intensiv.

Bei genauerer Betrachtung der Aktivität auf Ausführungsplanebene wird deutlich, dass ein signifikanter Anteil der CPU-Zeit bei der Umverteilung / Redistribution der Daten verbracht wird –

gemäß Real-Time SQL Monitoring / Active Session History reden wir hier von einem Anteil von 23% (X2-8) bzw. 26% (X3-8) der Samples, die CPU-Aktivität anzeigen.

Es ist also interessant, welche Laufzeiten erreicht werden können, wenn wir die Redistributionen weiter minimieren. Dazu werden die Daten geeignet nach dem Join-Kriterium HASH-partitioniert, so dass Full Partition Wise Joins möglich werden. Diese partitionsweisen Operationen haben mehrere Vorteile: Es wird potentiell deutlich weniger Speicher für die Join-Operationen benötigt, und es ist keine Redistribution der Daten notwendig – es können also möglicherweise die oben genannten ca. 25% CPU-Zeit eingespart werden. Desweiteren sind durch die nicht notwendigen Redistributionen auch keine BUFFERED Operationen notwendig. Dies kann auch für sogenannte „Partial Partition Wise"-Operationen gelten, da hier nur eine der beiden Datenquellen umverteilt werden muss und insofern die Anzahl der Redistributionen minimiert werden kann.

Insofern sollte im Fall von performance-kritischen Joins eine entsprechende Partitionierung zumindest erwogen werden. Natürlich kommt es dabei darauf an, wie die sonstigen Zugriffsmuster aussehen und ob durch die Partitionierung andere Probleme entstehen (zum Beispiel Verwendung von partitionsweisen Operationen nur bei bestimmten DOPs, die zu der Anzahl Partitionen passt, potentielle Ungleichverteilung von Daten in den Partitionen, Statistik-Management, Extent-Management, lokale Indizes, die pro Partition gelesen werden müssen etc.)

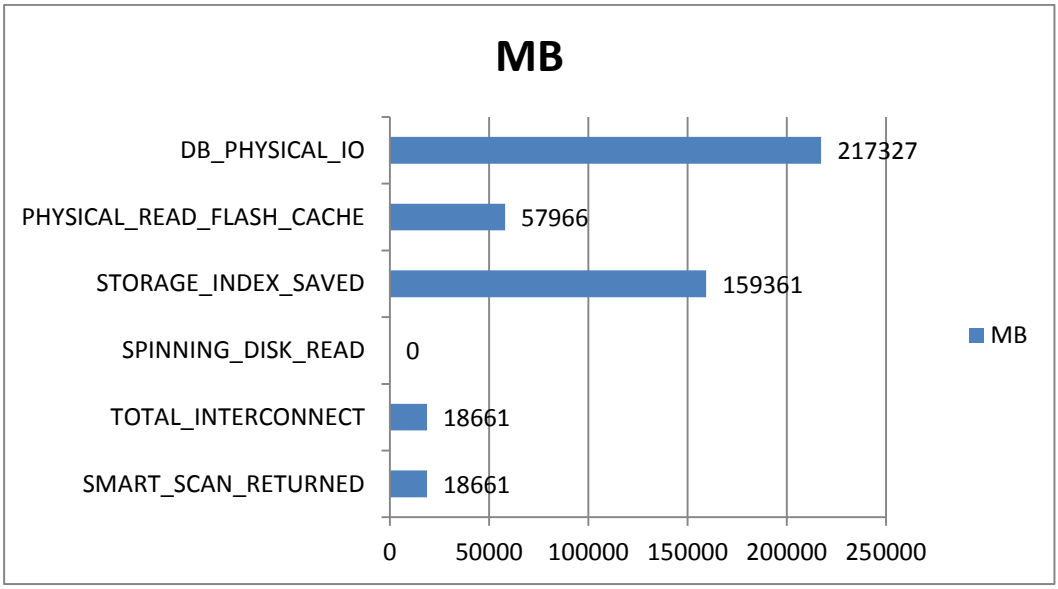
Hier die Ergebnisse des Full-Partition-Wise Joins:

X2-8: 14,8 Sekunden, im Durchschnitt ca. 15GB pro Sekunde gelesen, Peak 57GB pro Sekunde

X3-8: 7,6 Sekunden, im Durchschnitt ca. 29GB pro Sekunde gelesen, Peak 102GB pro Sekunde

Das ist nochmal eine signifikante Steigerung gegenüber den anderen Ergebnissen. In diesem Falle hier gilt diese Laufzeit für alle Variationen (Best-Case / tatsächliche Abfrage / Worst-Case), so dass sich im Vergleich zu den Varianten mit mehreren BUFFERED-Operationen eine extreme Beschleunigung ergibt.

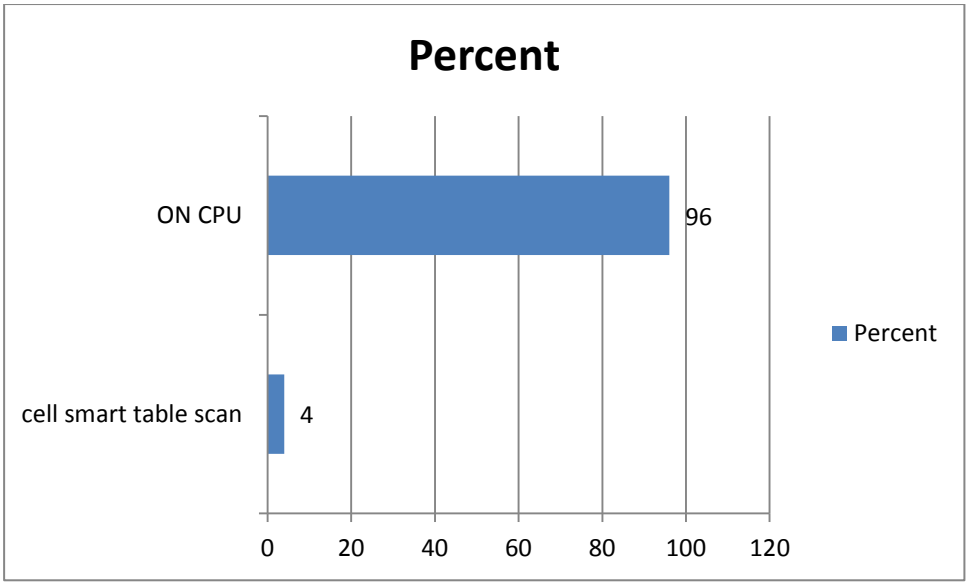
Die Session-Statistiken zeigen für alle Variationen folgendes Bild:



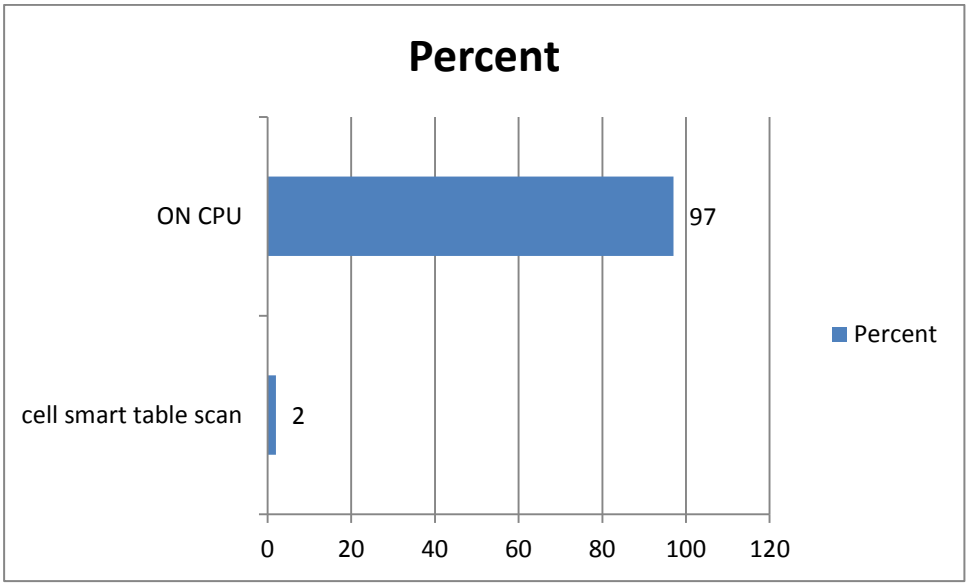
Die höhere Effizienz der Operation spiegelt sich auch in den Aktivitätsprofilen und der Datenbankzeit wider:

Aktivitätsprofil:

X2-8:

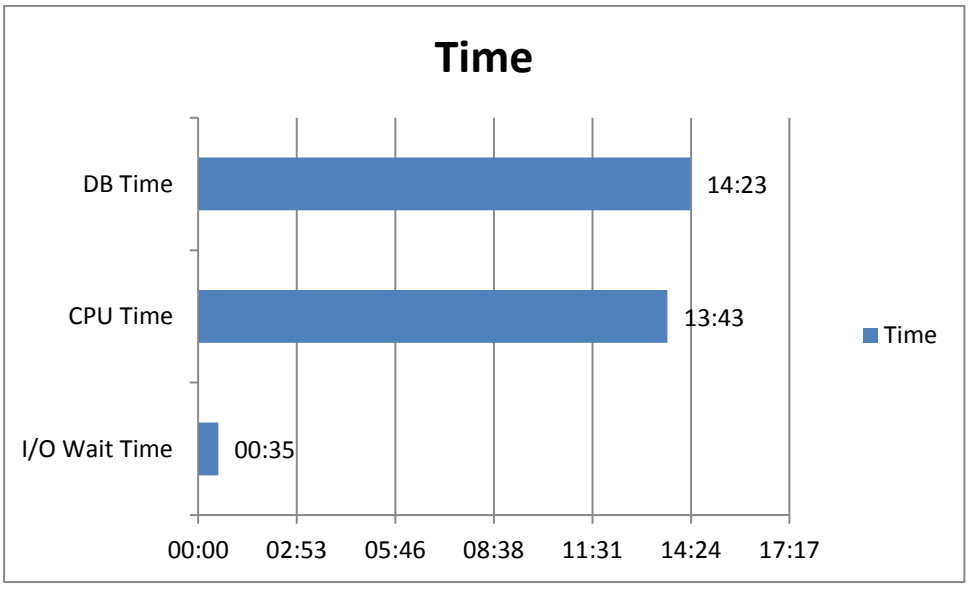


X3-8:

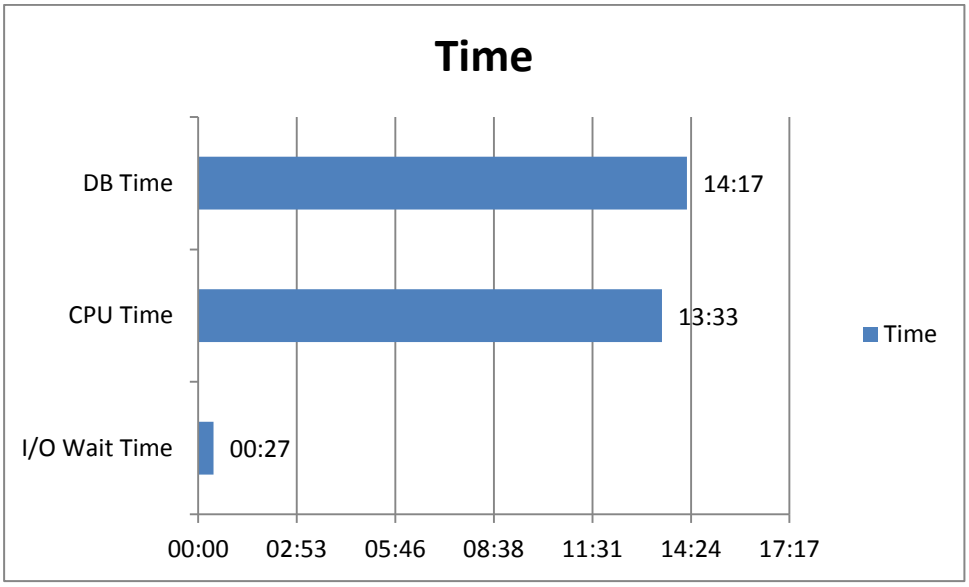


Datenbankzeit:

X2-8:



X3-8:



Gegenüber dem Best-Case ohne entsprechende Partitionierung reden wir also hier von einer Reduktion von 35-40% sowohl der Laufzeit, als auch der Datenbankzeit. Die Peaks der Lesegeschwindigkeiten liegen wieder im erwarteten Bereich (57GB bzw. 102GB), auch wenn die Durchschnitts-Lesegeschwindigkeit immer noch deutlich unter dem theoretisch Möglichen liegt.

Es wird also deutlich, dass die partitionsweise Verarbeitung der Daten eine Reduktion der Laufzeit / Datenbankzeit sogar über dem vorhergesagten ermöglicht (ca. 25% CPU-Zeit in Umverteilung verbracht), und insofern offensichtlich auch beim eigentlichen Join wohl weniger CPU-Zeit benötigt.

Der Unterschied zwischen Partition-Wise-Joins und nicht-partitionierter Verarbeitung wird sogar noch signifikanter, wenn mit einem niedrigeren DOP gearbeitet wird. Hier spielt eine ganz wichtige Eigenschaft von partitionsweisen Operationen eine Rolle, die bisher noch nicht erwähnt wurde: Bei den partitionsweisen Hash Joins werden die Hash-Tables partitionsweise erzeugt, unabhängig davon, wie viele Partitionen/Daten insgesamt verarbeitet werden müssen, da immer genau eine Partition der einen Tabelle zusammen mit der passenden Partition der andere Tabelle (pro Parallel Execution Server) verarbeitet wird.

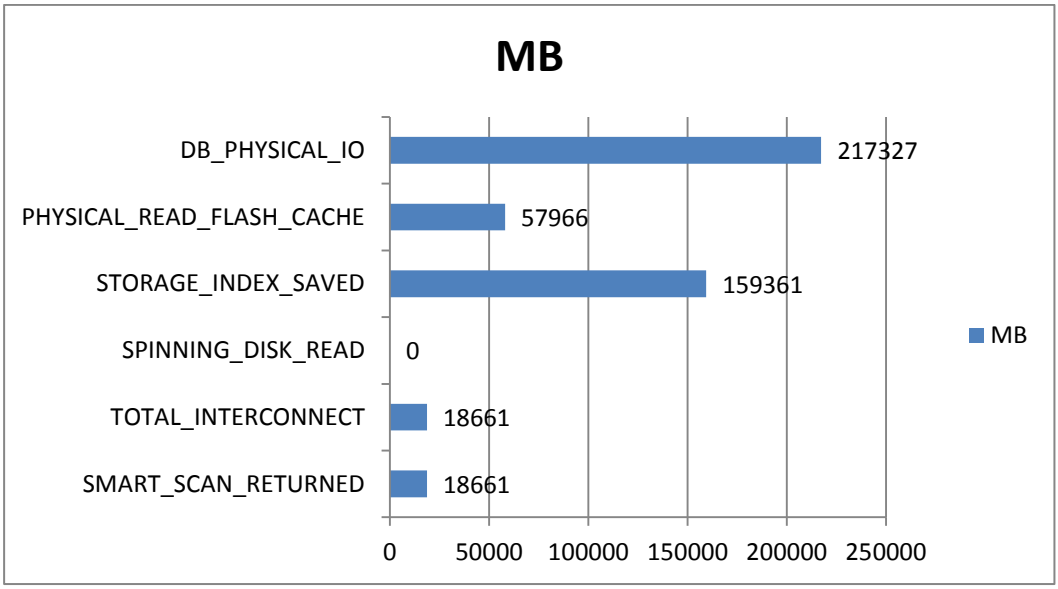
Bei nicht-partitionsweisen Operationen müssen alle Daten für das Erzeugen der Hash Tables auf die Parallel Execution Server verteilt werden, bevor mit der Verarbeitung der zweiten Datenquelle begonnen werden kann. Das heisst, der PGA-Speicher- und CPU-Bedarf liegt hier bei partitionsweiser Verarbeitung potentiell deutlich niedriger.

Hier die Ergebnisse der X2-8, wenn mit einem DOP von 16 gearbeitet wird:

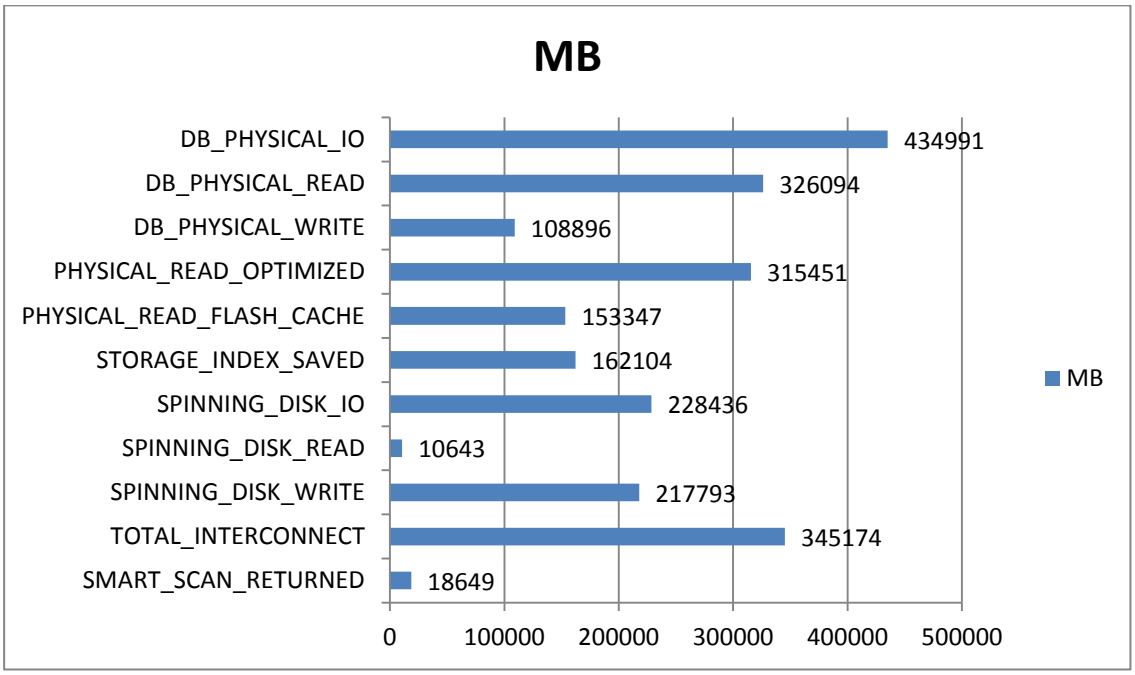
Partitionsweise: 48, 1 Sekunden, im Durchschnitt ca. 4,5GB pro Sekunde gelesen, Peak 18GB pro Sekunde, max. PGA-Speicher 6GB

Abfrage (2 x HASH JOIN BUFFERED): 149,4 Sekunden, im Durchschnitt ca. 1,5GB (2,9GB) pro Sekunde gelesen, Peak 12GB pro Sekunde, max. PGA-Speicher 22GB

Die Session-Statistiken der partitionsweisen Verarbeitung entsprechen genau der vorherigen mit DOP 64 bzw. 128:



Die Session-Statistiken der nicht partitionsweisen Verarbeitung sehen ganz anders aus:



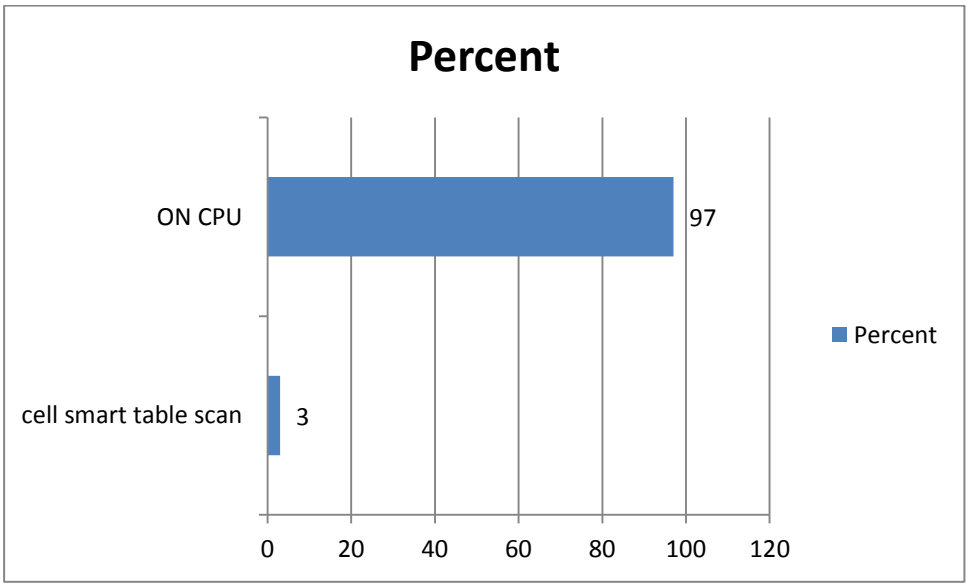
Hier wird also ähnlich wie beim „Worst-Case“ Szenario deutlich mehr I/O generiert. In diesem Fall nicht nur durch die zwei HASH JOIN BUFFERED, sondern auch durch die ganz normalen HASH JOINs, da jetzt mit Auto PGA-Einstellungen nicht mehr genügend PGA-Speicher pro Parallel Execution Server zugeteilt wird, um die jetzt ungefähr vierfache Datenmenge (DOP 64 -> DOP 16) pro Parallel Execution Server noch im PGA-Speicher zum Erzeugen der Hash Tables verarbeiten zu können, das heisst, jeder HASH JOIN lagert jetzt Daten auf TEMP aus.

Wie unterschiedlich effizient die beiden Varianten arbeiten, lässt sich daran sehen, dass die 48 Sekunden der partitionierten Variante bei DOP 16 nicht so viel langsamer sind als die 35 Sekunden für

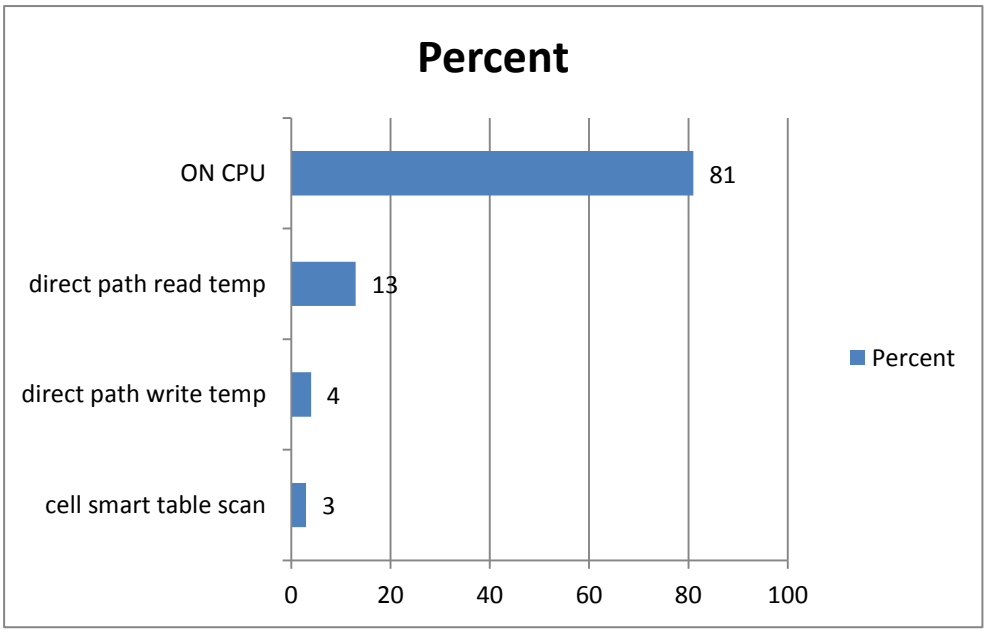
die nicht partitionierte Variante bei DOP 64, und auch weniger als viermal langsamer als die DOP 64-Zeit (ca. 15 Sekunden). Der verwendete PGA-Speicher bei DOP 16 war bei der partitionierten Variante maximal 6GB, während die nicht partitionierte maximal 22GB PGA-Speicher benötigte und trotzdem noch zusätzlich massiv auf TEMP auslagern musste.

Die Aktivitätsprofile sehen entsprechend unterschiedlich aus:

Partitioniert:

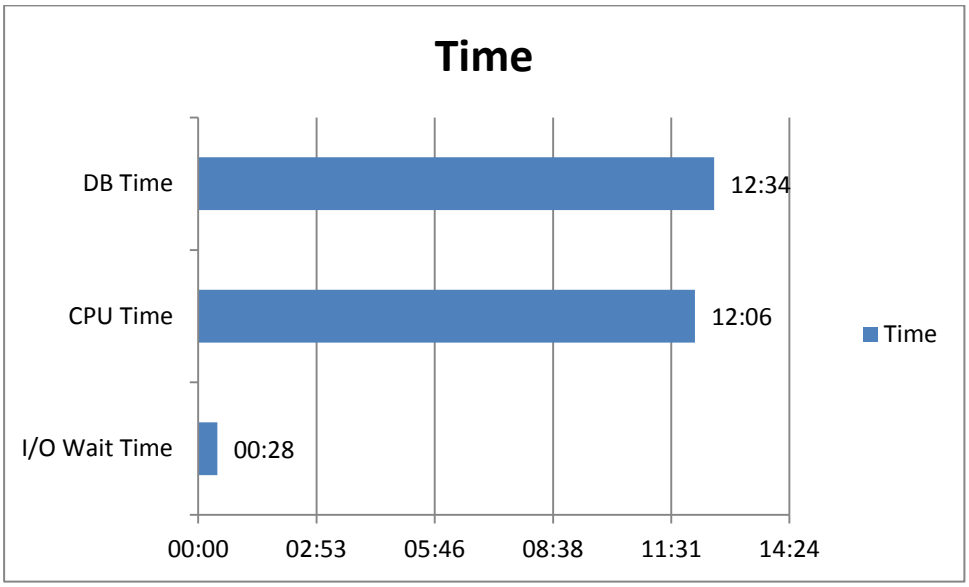


Nicht partitioniert:

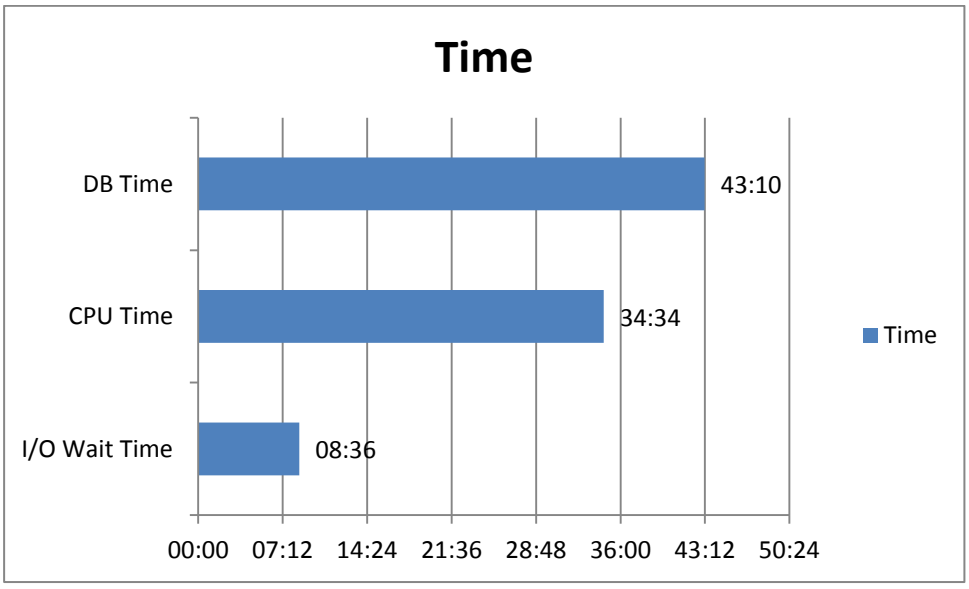


Und entsprechend auch die Datenbankzeit:

Partitioniert:



Nicht partitioniert:

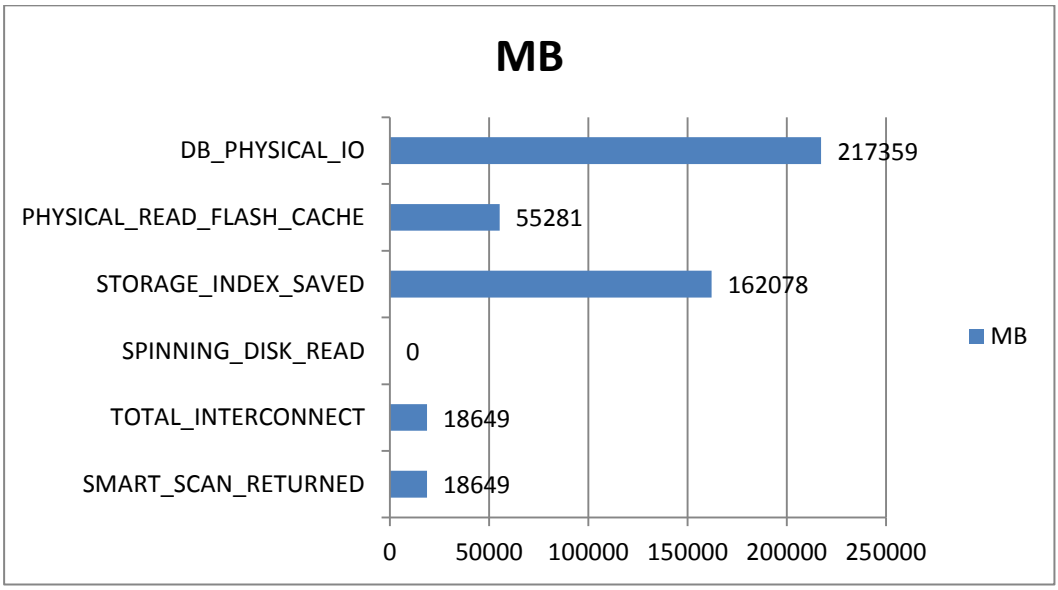


Erzwingt man bei der nicht-partitionierten Variante die Verwendung von mehr PGA-Speicher (über manuelle Einstellungen HASH_AREA_SIZE / SORT_AREA_SIZE), um die TEMP-Aktivität zu vermeiden, ergibt sich folgendes Bild:

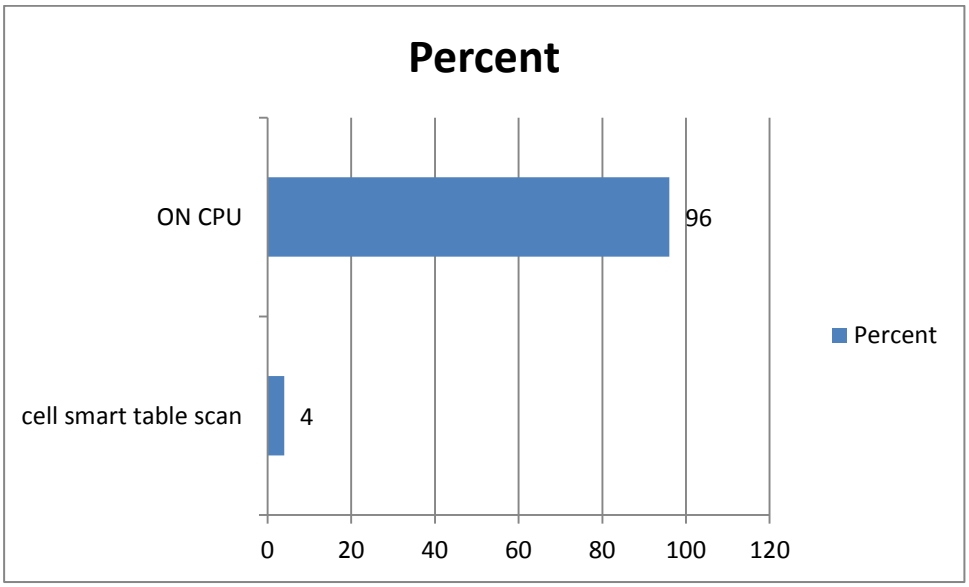
Abfrage (2 x HASH JOIN BUFFERED): 80 Sekunden, im Durchschnitt ca. 2,7GB pro Sekunde gelesen, Peak 10GB pro Sekunde, max. PGA-Speicher 65GB

Das ist also immer noch fast doppelt so lange wie die partitionierte Variante, und fast die zehnfache Menge an max. PGA-Speicher, was allerdings auch damit zu tun hat, dass Oracle leider mehr Speicher bei manuellem PGA-Management als bei automatischem verwendet.

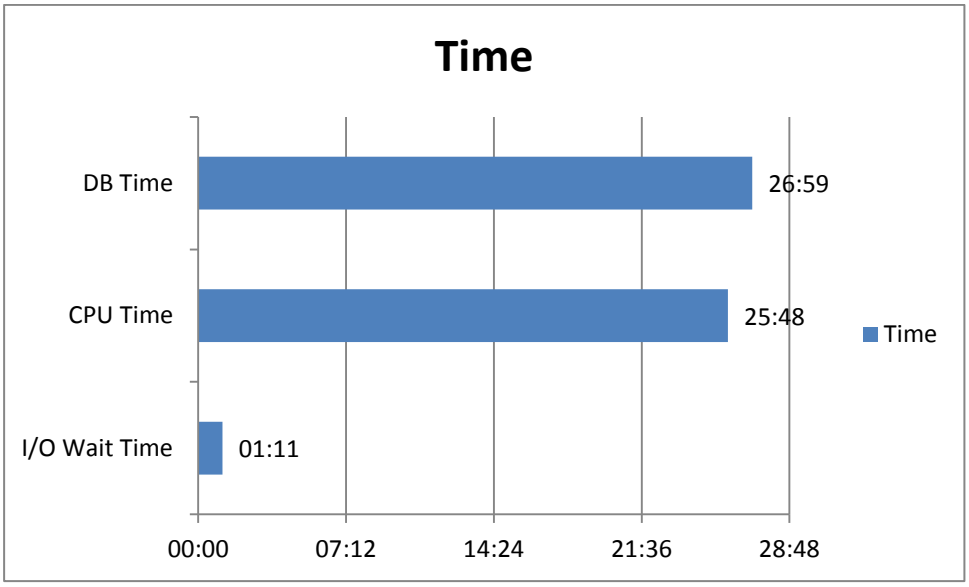
Die Session Statistiken zeigen, dass keine zusätzliche I/O-Aktivität notwendig war:



Das Aktivitätsprofil sieht folgendermaßen aus:

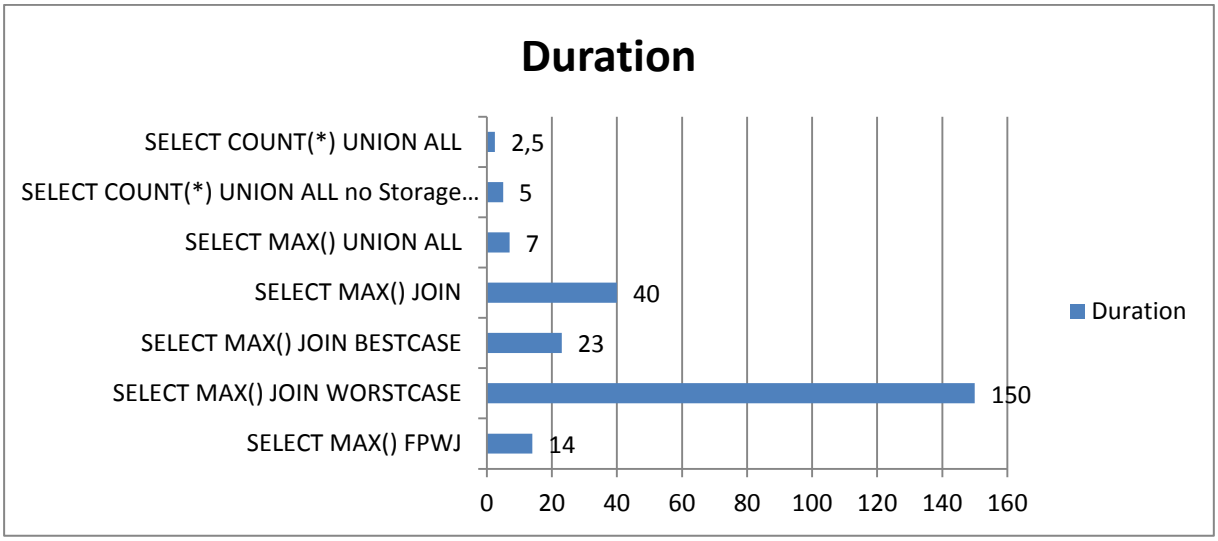


und die Datenbankzeit:

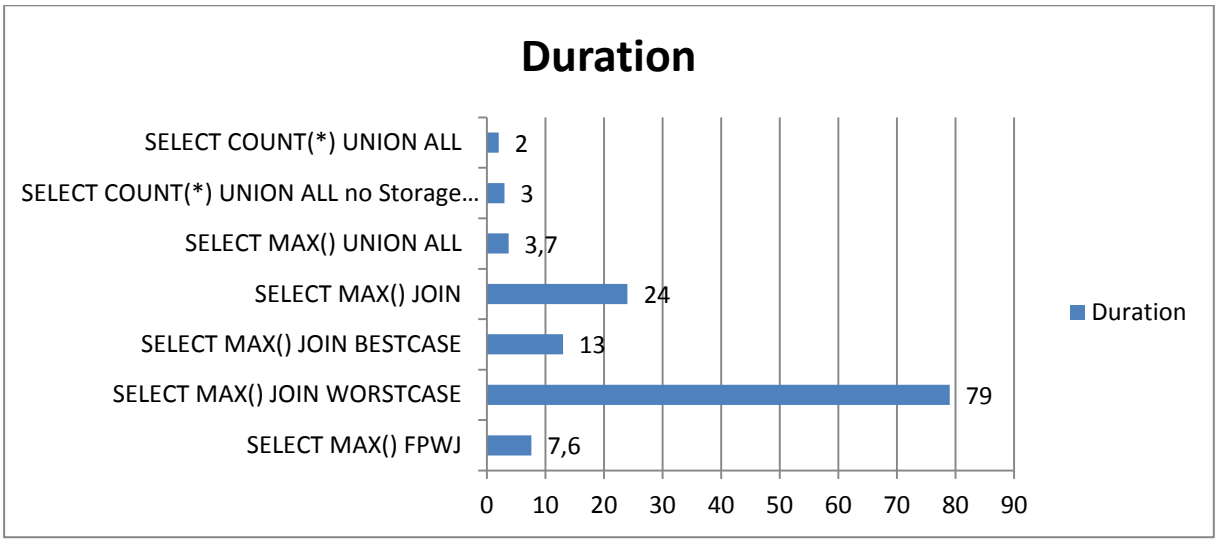

Time
DB Time
CPU Time
I/O Wait Time
26:59
25:48
01:11
Time
00:00
07:12
14:24
21:36
28:48

Die wichtigsten Laufzeiten nochmal im Überblick

X2-8:



X3-8:



Zusammenfassung

- Die Exadata Plattform liefert bei entsprechendem Einsatz hervorragende Performance für Full Segment Scans durch die Smart Scan-Technologie und den zugrundeliegenden Features wie Parallelisierung auf Storage Ebene, Offloading, Storage Indexes und Cell Flash Cache

- Eine hohe „Cell Offloading Percentage“ (in diesem Falle hier >= 91%) alleine sagt noch nichts über den tatsächlichen Vorteil – es kommt darauf an, wie viel Zeit in den Compute Nodes mit der Verarbeitung der Daten verbracht wird

- Je nachdem, wie viele Daten von den Storage Cells in die Compute Nodes transportiert werden müssen, und wie viel CPU-Zeit mit der Verarbeitung der Daten zugebracht wird, kann die Smart Scan-Technologie nicht mehr so große Vorteile verschaffen

- Schon mittels einer relativ geringen Datenmenge (ca. 18GB) können die CPUs auf den Compute Nodes für mehrere Sekunden vollständig ausgelastet werden, wenn komplexere Operationen wie mehrfache Joins durchgeführt werden

- Exadata bietet für Joins, die filtern, über die sogenannte Bloom Filter-Technologie auch die Möglichkeit, Joins über Smart Scans zu optimieren, indem die an die Compute Nodes zu liefernde Datenmenge minimiert werden kann. In dem hier vorgestellten Szenario konnten diese aber nicht eingesetzt werden, da es sich um keine filternde Joins handelte, sondern Outer Joins

- Die bei Parallelverarbeitung notwendige Umverteilung der Daten kann signifikant Zeit in Anspruch nehmen

- Die aufgrund der Umverteilung und der damit einhergehenden Implementationslimitierung teilweise notwendigen BUFFERED Operationen beeinflussen die Laufzeit, den benötigten PGA- und TEMP-Speicher signifikant

- Partitionsweise Verarbeitung (Partition Wise Operation) kann diesen Overhead minimieren und zu deutlich besseren Laufzeiten führen, insbesondere bei niedrigen DOPs oder auch serieller Verarbeitung

Kontaktadresse:
Randolf Geist
Unabhängiger Berater
Mannheim

Telefon: +49 (0) 170-758 1171
E-Mail randolf.geist@sqltools-plusplus.org
Internet: http://oracle-randolf.blogspot.com