Performance durch Kombination von Entwickler- und DBA-Logs?

Daniel Stein
Debeka
Koblenz, Deutschland

Schlüsselworte

Performance, Datenbank, Entwickler, DBA

Einleitung

Jede Anwendung führt verschiedene „logische Aktionen“ aus (Programmanweisungen, SQLs etc.), die Systemressourcen und andere Prozesse in Anspruch nehmen (Calls der Anwendungen im user-mode, kernel-mode, IO, Netzwerk, Datenbank etc.). Entwickler und DBAs können mit minimalem Overhead Informationen zum Laufzeitverhalten im Log-File der Anwendung sammeln. Mit den gewonnenen Informationen kann die real-Zeit der Anwendung so untergliedert werden, dass erkennbar wird, ob die meiste Verarbeitungszeit in der Datenbank, in der Anwendung oder off-CPU im System verbraucht wurde. Erscheint einer dieser Zeitabschnitte zu groß oder ist das Verhältnis der „logischen Aktionen“ zu den eingesetzten Systemressourcen überproportional, können mit den in diesem Artikel beschriebenen Methoden die einzelnen Zeitabschnitte detaillierter aufgeschlüsselt werden. So lassen sich schlussendlich Maßnahmen zur Performancesteigerung ableiten.

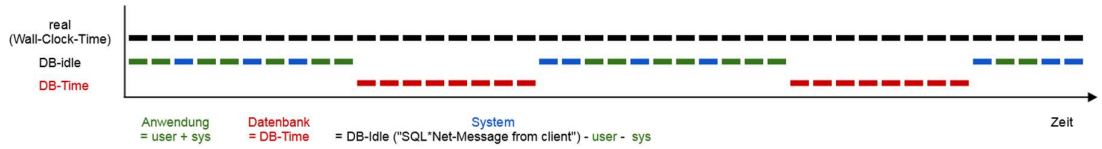

Abbildung 1: Beispielhafte Laufzeitzusammensetzung einer Single-Thread-Batchanwendung

Voraussetzungen für die spätere Analyse

Damit eine wie oben beschriebene Analyse stattfinden kann, sammeln Debeka-Batchanwendungen während der Ausführung Laufzeitinformationen und speichern sie im Log-File der Anwendung. Der Overhead ist gegenüber dem späteren Nutzen so gering, dass dieses Verfahren bei jedem Batchlauf in Produktion eingesetzt wird. Hilfreich dabei ist, dass alle Anwendungen durch ein zentrales Framework gestartet werden. Auch die Interaktion mit Oracle-Datenbanken geschieht über dieses Framework. Das Batchprogramm bekommt zum Start eine neuen Oracle-Session zugewiesen.

Folgende Informationen werden immer gesammelt:

1. Laufzeitinformationen der Anwendung (unix time Befehl)
   real : Gesamte Laufzeit der Anwendung (wall-clock-time)
   user: Ausführungszeit der im user-mode getätigten calls
   sys: Ausführungszeit der im kernel-mode getätigten calls (System-calls)

2. Zugriff auf Datenbank-Objekte und Transaktionsinformationen

```
---------------- DB-STATISTICS -------------
COMMIT                                    1
KV_TABLE1:INSERT                          5
LV_TABLE36:SELECT                         9
...
--------------------------------------------
```

3. Oracle-Session-Informationen aus diversen Oracle-Views

   a. V$MYSTAT / V$STATNAME
   Allgemeine Session-Informationen, wie z .B. die Anzahl der benötigten Roundtrips

   b. V$SESSION_EVENT
   Idle- und Non-Idle-Wartezeiten der Session - Worauf hat die Session wie lange gewartet um die geforderten SQl-Anfragen auszuführen, bzw. gab es Zeiträume in denen gar keine Arbeit verrichtet wurde (SQL*Net-Message-from-client)?

   c. V$SESS_TIME_MODEL
   Verteilung der durch die Session aktiv genutzten Zeit (on cpu) auf SQL-Ausführung, Parses oder ähnlichem.

Bilanzierung der geloggten Informationen

Nach einem Batchlauf lassen sich die gewonnenen Informationen einordnen und auswerten. In einer Laufzeitbilanz werden die Anwendungsinformationen den eingesetzten Systemressourcen gegenübergestellt. Dabei werden diese in verschiedene Zeitabschnitte eingeteilt. Dieses Vorgehen wird nachfolgend am Beispiel einer **Single-Thread-Batch**-Anwendung beschrieben.

Anwendung	Zeitabschnitte (Systemressourcen)
Programmfunktionen SQL-Statements DB-Statistiken Angeforderter IO	**Anwendung** *(user + sys)* **System** *(„SQL*Net Message from client" – user – sys)* **Datenbank** *(DB-Time)*: Untergliederung: - "non idle wait events" - SQL-Ausführung - etc.
real-Laufzeit	real-Laufzeit ("SQL*Net Message from client" + Datenbank)

Durch die Abfrage der oben genannten Oracle-Views ist nach dem Batchlauf eine feine Untergliederung des Zeitabschnitts „Datenbank" aus den geloggten Informationen ersichtlich.

Auf den Vortragsfolien finden sich Beispiele, wo diese Informationen in Kombination mit den Anwendungsinformationen für eine Analyse schon ausreichen. Die anderen beiden Zeitabschnitte müssen weiter untergliedert werden, um einem dort verortetem Performance-Problem auf die Schliche zu kommen. Doch dazu später mehr. Man sieht auf einen Blick, wo die meiste Zeit verbraucht wurde. Da die Informationen immer zur Verfügung stehen, fallen beim Durchsehen von Log-Files auch zu viel genutzte Ressourcen auf, auch wenn gerade nicht nach Engpässen geforscht wird.

Beschreibung der Untergliederung

Das Oracle Idle-Wait-Event „SQL*Net Message from client“ misst die Zeit, in der die Datenbank-Session auf Aufgaben von der Anwendung wartet. Wenn bei einer Performanceanalyse die Datenbank isoliert betrachtet wird, ist das Event wenig hilfreich, da es nur aussagt, dass die DB in diesem Zeitraum nichts zu tun hatte. In Kombination mit Laufzeitinformationen der Applikation (real, user, sys) ist das Event nützlich, um die Programmlaufzeit grob in drei **Zeitabschnitte** verrichteter Arbeit zu unterteilen: „Zeit des Anwendungs-Prozesses“, „System“ (Netzwerk, Interrupt-Handling etc.) und durch die Oracle-Datenbank notwendige Arbeit um die Aufgaben der Anwendung zu erledigen (Datenbank). Voraussetzung, um „SQL*Net Message from client“ wie beschrieben zu nutzen ist, dass die Session während der gesamten Laufzeit der Anwendung bestand.

Nach der groben Einteilung ergeben sich nun zwei Analysewege:

1. Passen die Aktionen der Anwendung zu der von der Datenbank verrichteten Arbeit und ist der Einsatz der Mittel gerechtfertigt?

    a. Ja!
    Beispiel: Eine hohe Anzahl an Commits wird auch zu viel transaktionsbezogener Arbeit auf der Datenbankseite führen.
    Die Aktionen passen zu den eingesetzten Mitteln, aber vielleicht kommt die Anwendung ja mit weniger Commits aus. So könnte die benötigte Zeit und Arbeit verringert werden.

    b. Nein!
    Beispiel: Eine Anwendung verarbeitet große Teile eines Schemas. Aber der Zugriff auf die Daten erfolgt nur per Single-Block-Read. Hier passt die Intention der Anwendung nicht zu der in der Datenbank verrichteten Arbeit. Es sollte überprüft werden, warum die Daten nur häppchenweisen und nicht im Ganzen verarbeitet werden.

2. Ist einer der drei Zeitabschnitte überproportional groß?
    Wenn in einem der Abschnitte viel Zeit verbraucht wird, dann muss man die Laufzeitverteilung des Bereiches weiter untergliedern, bis man den Verursacher gefunden hat, bzw. ihn einer Aktion auf der Anwendungsseite zuordnen kann. Es ist so ähnlich wie bei Google-Maps auf eine Stadt zu zoomen, wo man zunächst nur Stadteile sieht, aber bei jeder weiteren Zoom-Stufe mehr Details sichtbar werden. Im Unterschied zu Google-Maps erfordert hier jeder Zeitabschnitt seine eigene Methodik zum „heranzoomen“. Wie bereits erwähnt, ist man dank Oracle beim Abschnitt „Datenbank“ im Vorteil.

Im folgenden Abschnitt werden Ansätze zur detaillierten Aufschlüsselung der Laufzeit gezeigt. So können pro Zeitabschnitt die Verursacher langer Laufzeiten gefunden werden.

Auf den Vortrags-Folien finden sich dazu detaillierte Praxisbeispiele.

Aufgliederung der einzelnen Zeitabschnitte

Zeitabschnitt System
In diesem Bereich ist das Aufzeigen von Laufzeitverteilungen etwas komplizierter, da Systeminterna für die meisten Entwickler und Admins nicht zum Tagesgeschäft gehören.

Um den Zeitabschnitt zu kartografieren, empfiehlt es sich, einmalig alle Systemaufrufe zu tracen und Anzahl und Dauer aufzuzeichnen. Dazu eignen sich **statische** Tracing-Tools wie strace oder truss. Wenn ein System-Call häufig aufgerufen wird, bzw. viel Laufzeit beansprucht, sollte zunächst der Verursacher ausgemacht werden.

Dazu bedient man sich **dynamischen** Tracing-Tools wie dtrace (Linux) bzw. probeVUE (AIX). Diese definieren verschiedene Hooks für System-Calls, den System-Scheduler, Netzwerk etc. Der Entwickler kann Skripte mit Call-Back-Methoden schreiben, die sich auf diese Hooks beziehen. Diese werden dann von den Tracing-Tools aufgerufen, wenn die zu untersuchende Anwendung den entsprechenden Aufruf tätigt. So ist das Sammeln von Informationen einfach möglich.

Beispiel:

Eine Anwendung ruft mehrere Millionen Mal den System-Call „stat“ auf und zwar immer für die Datei "/etc/localtime". Mit dem folgenden Script lässt sich herausfinden, wer diesen System-Call für die genannte Datei aufruft.

```
#!/bin/probevue

int stat(const char *pathname, struct stat *statbuf);

/* Wenn der aktuell getracete Prozess ($1) den Systemcall stat aufruft, und
die Datei "/etc/localtime" lesen will, dann geben wir den Stacktrace 5
Stufen aus und beenden das tracing */

 @@syscall:$1:stat:entry {
   if (strcmp(get_userstring(__arg1,-1), "/etc/localtime") == 0) {
     printf("%t\n", get_stktrace(5));
     exit();
   }
}
```

Anhand des Stacktraces kann nun die Stelle in der Anwendung identifiziert werden, die den System-Call ausgelöst hat. Nun ist zu prüfen, ob der System-Call sinnvoll und notwendig ist oder ob es eine schnellere Alternative gibt. Vielleicht wird die den System-Call auslösende „logische Aktion“ im Programm auch öfter als notwendig ausgeführt? Weiterführende Details zu dem Beispiel findet man auf den Vortragsfolien.

Eine hervorragende Quelle zu dem Thema Tracing ist die Homepage von Brendan Gregg [1] (Performance-Engineer; Netflix). Er beschäftigt sich zwar in der Hauptsache mit der Performance und dem Tracing von Linux, aber seine Methoden lassen sich auch auf Unix-Systeme wie AIX adaptieren.

[1] http://www.brendangregg.com/

Zeitabschnitt Anwendung:
Diesem Bereich kann man sich auf mehreren Wegen nähern. Man kann die bei dem Zeitabschnitt „System“ beschriebene Methodik anwenden. Am Anfang der Betrachtung einer Anwendung sind dieser Bereich und der zuvor genannte Bereich noch nicht klar getrennt, denn eine Anwendung tätigt System-Calls und diese wiederum können, müssen aber nicht, zu Interrupts, Netzwerk-Verkehr oder ähnlichem führen. Diese Dinge gehören in den Bereich „System“ und fallen damit in die Zuständigkeit des System-Admins. Exzessive User- / System-Calls, die nicht in den Bereich fallen, sollte sich hingegen der Entwickler ansehen und prüfen, warum diese ausgelöst werden und ob sie notwendig sind. Durch den Einsatz der Tracing-Tools wird aber die Trennung klarer.

Ein weiterer Weg ist der Einsatz von Profilern. Profiler eignen sich, um herauszufinden, wie häufig welche Funktionen einer Anwendung aufgerufen wurden. Eine Einordnung wieviel Zeit einzelne Funktionen benötigen ist schwierig, denn Profiler verursachen einen relativ großen Overhead bei ihrem Einsatz, da die Zeitmessung meistens mittels des System-Call „times“ erfolgt. Der Einsatz sollte daher im Test und in Produktion nur bei Bedarf erfolgen!

Zeitabschnitt: „Datenbank“
Im Gegensatz zu den anderen beiden Bereichen hat man es hier leichter, da durch die Instrumentierung der Oracle-DB die Untergliederung der Laufzeitinformationen schon erfolgt ist und nur noch, wie oben gezeigt, abgefragt werden muss. So ist zum Beispiel die direkte Gegenüberstellung der von der Anwendung abgesetzten SQL-Statements bzw. der Intention der Statements und der von der Datenbank ausgeführten Aktionen und der dafür eingesetzten Systemressourcen möglich. In der Praxis hat sich gezeigt, dass es immer sinnvoll ist, die Intention eines Statements zu hinterfragen, denn es kommt vor, dass das SQL und das Schema nicht zu den Intentionen der Anwendung passen. Wenn zum Beispiel große Teile einer Tabelle eines Schemas verarbeitet werden sollen und man findet in den gesammelten Metriken viele SQL-Net-Roundtrips und notwendigen IO per „db file sequential read“, sollte sowohl in der Anwendung, als auch in der DB nach der Ursache geforscht werden, warum die Daten nur in kleinen Häppchen und nicht im Ganzen verarbeitet werden.

Im Folgenden einige interessante Metriken (im Fließtext kursiv geschrieben), die auch Entwickler kennen sollten:

1. IO-bezogen
   Oracle unterscheidet, ob in einem Lesevorgang ein Block *„db file sequential read“* oder mehrere Blöcke *„db file scattered read“* gelesen werden. Es ist immer effizienter, direkt mehrere benötigte Blöcke am Stück lesen zu können. Dem steht meistens entgegen, dass die Daten in der Tabelle nicht so geclustert sind, wie es für die Anfrage benötigt wird. Hier ist es zum Beispiel von Vorteil, wenn man die Möglichkeit hat, Partitionierung einzusetzen. Dann kann man nämlich die Tabelle so modellieren, dass die Daten den Anfragen entsprechend geclustert vorliegen.

2. Transaktions-bezogen
   Wenn ein Commit abgesetzt wird, wartet der Benutzerprozess bis der Logwriter-Prozess der Datenbank zurückmeldet, dass die Redo-Informationen sicher auf die Festplatte geschrieben wurden. Diese Zeit wird mit dem Event *„log file sync"* getrackt. Wenn dieses Event sehr viel Zeit beansprucht und die Anzahl der Commits in den eingangs genannten DB-Statistiken hoch ist, sollte der Entwickler überprüfen, ob er nicht mit weniger Commits auskommt. Falls das nicht möglich ist, sollte der DBA den Logwriter-Prozess beschleunigen. Wenn sehr viele Informationen geschrieben werden, und der Platz in den Log-Files nicht ausreicht, kommt es zu *„log file switch“*-Waits um die sich wiederum der DBA kümmern sollte.

3. Kommunikation mit der Anwendungsseite
   Wenn in den gesammelten Informationen das Wait-Event *„SQL*Net more data to client“* auftaucht, dann haben manche Fetches mehr Speicher gebraucht, als der Netzwerkpuffer der DB-Session (SDU) zur Verfügung hatte. Bei einer hohen Anzahl dieser Waits sollte, wenn möglich die SDU Größe angepasst werden, da viele Netzwerke heute in der Lage sind, mit einer MTU von mehr als 1500 bytes zu arbeiten. Wenn die SDU-Größe die MTU-Größe unterschreitet, bleiben Netzwerkkapazitäten ungenutzt. Wenn eine Änderung der SDU nicht möglich ist, sollte der Entwickler weniger Daten mit einem Fetch anfordern (Prefetch-Value ändern). Die eigentliche Wartezeit ist nur der Transfer des user-mode-Netzwerkbuffers (SDU) in den kernel-mode TCP-Socket-Buffer. Es sagt nichts über die Transferzeit der Daten zum Client aus. Diese Zeit versteckt sich wieder im Wait-Event „*SQL-Net message from client*“.

   Das Wait-Event *„SQL*Net roundtrips to/from client“* gibt an, wie viele Netzwerk-Roundtrips nötig waren, um die Anfragen an die Datenbank und die Ergebnismengen zurück zu übertragen.

Fazit

Durch geeignete Instrumentierung des eigenen Systems lassen sich in einer Produktionsumgebung mit geringem Overhead prophylaktisch viele nützliche Informationen sammeln, die beim Vorliegen eines Engpasses einen schnellen und gezielten Einstieg in die Analyse ermöglichen. Darüber hinaus lässt sich durch die gezeigte Zusammenführung verschiedener Informationen häufig Potential zur Performancesteigerung finden.

Kontaktadresse:

Daniel Stein

Debeka

Ferdinand-Sauerbruch-Str, 18

D-56072 Koblenz

Telefon: +49 (0) 261-498 3824

E-Mail daniel.stein@debeka.de

Internet: www.debeka.de