Viel zu oft beginnt die Optimierung einer Applikation zu spät – nämlich erst dann, wenn deren Entwicklung bereits abgeschlossen ist. Soweit kann es nur kommen, wenn der Performance eine geringere Bedeutung als anderen Applikations-Anforderungen beigemessen wird. Performance ist jedoch nicht eine Option; sie ist ein Schlüsselkriterium jeder Applikation. Ungenügende Performance wirkt sich nicht nur nachteilig auf die Akzeptanz der Applikation aus, sie führt normalerweise – aufgrund der geringeren Produktivität – auch zu einem niedrigeren Return of Investment. Zusätzlich verursacht schlechte Performance auch höhere Software-, Hardware- und Betriebskosten.

Designing for Performance

Christian Antognini, Trivadis AG

Im Folgenden werden die häufigsten datenbankrelevanten Design-Probleme beschrieben, die in vielen Fällen zu einer schlechten Performance führen. Die aufgeführten Praxistipps basieren auf der jahrelangen Tuning-Erfahrung des Autors. Sie sollen helfen, rechtzeitig notwendige Überlegungen in das Datenbank-Design einfließen zu lassen und so spätere, aufwändige Tuningmaßnahmen zugunsten einer optimalen Performance zu vermeiden.

Unzulänglichkeiten im logischen Datenbank-Design

Früher war es selbstverständlich, einen Datenarchitekt in das Entwicklungsprojekt einzubeziehen. Dieser war oft nicht nur für die Daten und das Datenbank-Design verantwortlich, sondern auch Teil des Teams für die Gesamtarchitektur und das Design der Applikation. Er hatte meist große Erfahrung mit Datenbanken und wusste genau, wie man Datenintegrität und Performance mittels geeigneten Designs sicherstellen kann. Heutzutage ist dies leider immer weniger der Fall. Viel zu oft gibt es in Projekten kein formales Datenbank-Design mehr. Die Applikationsentwickler erstellen das Client- oder das Middle-Tier-Design und plötzlich wird das Datenbank-Design von einem sogenannten „Persistent-Framework-Tool“ (im Java-Umfeld beispielsweise Hibernate oder iBatis) generiert. Die Datenbank wird lediglich als einfacher Daten-Container betrachtet.

Praxistipp: Die Bedeutung des logischen Datenbank-Designs darf nicht unterschätzt werden. Wird dieses nicht korrekt entwickelt, können Performance-Probleme auftreten.

Implementation von generischen Tabellen

Jeder CIO träumt von Applikationen, die sich einfach und schnell an neue oder veränderte Anforderungen anpassen lassen. Der Schlüsselbegriff heißt „Flexibilität“. Solche Träume manifestieren sich oftmals in Form von Applikationen mit generischem Datenbank-Design. Funktionen wie beispielsweise das Hinzufügen neuer Attribute, sind dann nur noch eine Frage der Konfiguration. Die beiden folgenden Datenbank-Design-Modelle erlauben diese Flexibilität:

- *Entity-Attribute-Value-Modell (EAV)*
  Wie der Name sagt, werden für jede Information mindestens drei Spalten benötigt: Entität, Attribut und Wert. Jede Kombination definiert den Wert eines spezifischen Attributs, das mit einer spezifischen Entität verknüpft ist.
- *XML*
  Jede Tabelle hat nur wenige Spalten, wobei zwei Spalten immer vorhanden sind: ein Identifier und eine XML-Spalte, in der fast alles gespeichert ist. Es können auch zusätzliche Spalten für die Speicherung von Metadaten (zum Beispiel wer wann die letzte Modifikation vorgenommen hat) vorhanden sein.

Aus Sicht der Performance sind diese Designs äußerst problematisch. Der Grund liegt in der engen Beziehung von Flexibilität zur Performance. Ist die Flexibilität maximal ausgeprägt, dann ist die Performance minimal (und umgekehrt).

Praxistipp: Ein flexibles Design bedeutet implizit auch suboptimale Performance. In gewissen Situationen mag suboptimal gut genug sein, aber in anderen Situationen kann dies gravierende Folgen haben. Daher sollten generische Tabellen nur zum Einsatz kommen, wenn damit auch die erforderliche Performance erreicht werden kann.

Verzicht auf Constraints

Constraints (Primärschlüssel, Unique-Keys, Fremdschlüssel, NOT NULL-Constraints und Check-Constraints) sind nicht nur fundamental, um die Datenintegrität zu gewährleisten, sie werden auch vom Optimizer für die Generierung der Ausführungspläne verwendet. Ohne Datenbank-Constraints ist der Optimizer nicht in der Lage, bestimmte Optimierungstechniken auszunutzen. Sind Constraints auf Applikationsebene definiert, führt dies zu zusätzlichem Code, der sowohl geschrieben, als auch getestet werden muss. Probleme können dabei auch im Bereich der Daten-Integrität entstehen, weil die Daten auf Datenbank-Ebene jederzeit manuell geändert werden können.

Praxistipp: Es empfiehlt sich, alle Constraints auf Datenbank-Ebene zu definieren.

Unzulängliches physisches Datenbank-Design

Projekte, in denen das logische Design direkt auf das physische Design

abgebildet wird, ohne die Vorteile der Oracle-Datenbank-Eigenschaften zu nutzen, sind nicht ungewöhnlich. Das häufigste und offensichtlichste Beispiel ist die direkte Verknüpfung aller Beziehungen zu einer Heap-Tabelle. Aus Sicht der Performance ist dies nicht optimal. In gewissen Situationen gewährleisten Index-Organized-Tabellen (IOT), Index-Cluster oder Hash-Cluster eine bessere Performance.

Eine Oracle-Datenbank bietet wesentlich mehr Indexierungsmöglichkeiten als normale BTree- und Bitmap-Indizes. Je nach Situation sind komprimierte Indizes, Revers-Indizes, Function-Based-Indizes (FBI), linguistische Indizes oder Text-Indizes zur Verbesserung der Performance geeignet.

Für große Tabellen ist zudem der Einsatz von Partitionierung interessant. Den meisten Datenbank-Administratoren ist dies auch bewusst. Ein häufiges Problem ist jedoch, dass die Entwickler denken, die Partitionierung der Tabellen hätte keinen Einfluss auf das logische Datenbank-Design. Dies trifft nur teilweise zu. Aus diesem Grunde empfehlen wir, Partitionierung gleich von Anfang an einzuplanen.

Praxistipp: Für eine bestmögliche Performance sollte man das logische Design nicht direkt auf das physische Design abbilden.

Falsche Datentyp-Auswahl

Auf den ersten Blick scheint die Wahl des Datentyps keine schwierige Aufgabe zu sein. Trotzdem findet man immer mehr Systeme mit unzulänglichen Datentypen. Es gibt vier Hauptprobleme im Zusammenhang mit der Datentyp-Auswahl:

- *Falsche oder unzureichende Datenvalidierung*
  Daten müssen innerhalb der Datenbank validiert werden können. Das heißt, es sollten beispielsweise keine numerischen Werte in einem „VARCHAR2“-Datentyp gespeichert sein, weil die Daten sonst extern zu validieren sind.
- *Informationsverlust*
  Beim Konvertieren eines Originaldatentyps in einen Datenbank-Datentyp können Informationen verloren gehen. Werden beispielsweise Datum und Zeit in einem „DATE“-Datentyp anstelle eines „TIMESTAMP WITH TIME ZONE“-Datentyps gespeichert, gehen die Sekundenbruchteile und die Zeitzonen-Information verloren.
- *Gewisse Dinge funktionieren nicht wie erwartet*
  Operationen und Eigenschaften, für die die Reihenfolge der Daten wichtig sind, können zu unerwarteten Resultaten führen. Zwei typische Beispiele sind Range-partitionierte Tabellen und „ORDER BY“-Klauseln.
- *Optimizer-Anomalien*
  Der Optimizer kann unter Umständen aufgrund einer falschen Datentyp-Auswahl keine gute Schätzung hervorbringen, was zu nicht optimalen Zugriffspfaden führt. Der Optimizer kann in einem solchen Fall seine Aufgabe nicht erfüllen, weil ihm die korrekte Information fehlt.

Praxistipp: Es gibt viele gute Gründe, die Datentypen korrekt und sorgfältig auszuwählen; man erspart sich damit viele Probleme.

Inkorrekte Verwendung von Bind-Variablen

Aus Sicht der Performance haben Bind-Variablen Vor- und Nachteile. Zu den Vorteilen zählt das Cursor-Sharing im Library-Cache und die damit verbundene Verhinderung von Parsing. Der Nachteil von Bind-Variablen in der „WHERE“-Klausel (und ausschließlich dort) ist, dass entscheidende Informationen dem Optimizer verborgen bleiben. Der Optimizer kann beispielsweise Literale besser verwenden als Bind-Variable. Dies ist speziell dann der Fall, wenn dieser prüfen muss, ob ein bestimmter Wert innerhalb oder außerhalb eines Wertebereichs liegt (größer oder kleiner als der größte oder der kleinste Attributwert), wenn ein Prädikat in der „WHERE“-Bedingung auf einem Wertebereich basiert (beispielsweise „HIREDATE > ‹2009-12-31›“) oder wenn der Optimizer für die Optimierung Histogramme verwendet. Konsequenterweise sollte man Bind-Variable nur dann verwenden, wenn keine der drei erwähnten Bedingungen zutrifft. Ausgenommen davon sind SQL-Anweisungen, deren Parse-Zeit um Größenordnungen kleiner ist, als deren Ausführungszeit. In diesem Fall ist der Einsatz von Bind-Variablen nicht nur irrelevant für die Ausführungszeit, sondern es erhöht sich auch das Risiko von sehr ineffizienten Ausführungsplänen.

Praxistipp: Bind-Variable sollten auf keinen Fall zum Einsatz kommen, wenn der Optimizer maßgeblich von Histogrammen Gebrauch macht. Bind-Variable verringern das Sicherheitsrisiko durch SQL-Injection, weil die Syntax einer SQL-Anweisung über einen Bind-Variablenwert nicht geändert werden kann.

Fehlender Einsatz von Advanced-Datenbank-Features

Oracle verfügt mit seiner High-End-Datenbank-Engine über zahlreiche Advanced-Features. Darunter versteht man Datenbank-Funktionen und -Eigenschaften, die üblicherweise in anderen Datenbank-Produkten nicht verfügbar sind. Der Autor empfiehlt, die datenbankseitigen Möglichkeiten soweit wie möglich zu nutzen und bereits existierende Features nicht manuell nachzubilden. Dies bedeutet natürlich auch, die mit der jeweiligen Version neu eingeführten Features speziell zu beachten und sie nicht nur hinsichtlich ihrer Funktionalität, sondern auch bezüglich der Stabilität zu testen.

Ein verbreitetes Argument gegen den Einsatz der Advanced-Features sind die eingeschränkten Portierungsmöglichkeiten auf Datenbanken anderer Hersteller – und damit eine gewisse Abhängigkeit von Oracle. Auf der anderen Seite werden Unternehmen die Datenbank unter einer bestimmten Applikation nur in Ausnahmefällen austauschen, sondern gleich die ganze Applikation ersetzen.

Praxistipp: Die Entwicklung von datenbankunabhängigem Applikationscode empfiehlt sich nur dann, wenn dafür gute Gründe vorliegen. Auch wenn dies der Fall ist, ist das Thema „Flexibilität vs. Performance“ zu beachten.

Fehlende Verwendung von Stored-Procedures

Ein Spezialfall des oben erwähnten Punkts ist die Verwendung von PL/SQL-Stored-Procedures für die Verarbeitung großer Datenmengen. Das bekannteste Beispiel ist ein Extract-Transform-Load-Prozess (ETL). Erfolgt die Extrakt- und Ladephase in derselben Datenbank, ist es aus Performance-Sicht beinahe fahrlässig, in der Transformationsphase auf die Vorteile der SQL- und PL/SQL-Engine zu verzichten, welche die Quell- und Ziel-Daten verwaltet. Leider führt die Architektur diverser bekannter ETL-Werkzeuge genau zu diesem Verhalten. Das heißt, die Daten werden aus der Datenbank extrahiert (und oft auf einen anderen Server verschoben), transformiert und wieder in die Datenbank geladen.

Praxistipp: Für die optimale Performance empfehlen wir eine möglichst datennahe Verarbeitung mithilfe des Einsatzes von PL/SQL-Prozeduren.

Ausführung von unnötigen Commits

Commit-Operationen führen zu Serialisierung. Der Grund dafür ist einfach: Es gibt einen einzigen Prozess, den Logwriter-Prozess, der Daten in die Redo-Log-Files schreibt. Weil eine serialisierte Operation nicht skaliert, sollte man sie aus Performance-Gründen soweit wie möglich verhindern oder zumindest minimieren. Teilweise ist dies durch das Zusammenfassen einzelner Transaktionen möglich – also beispielsweise das Laden von Datensätzen durch Batch-Jobs. Anstatt nach jedem Datensatz ein Commit auszuführen, ist es vorteilhafter, die Daten in Batches einzufügen und nach jedem Batch einen Commit-Befehl auszuführen.

Praxistipp: Transaktionen sollten nicht zu häufig einen Commit-Befehl enthalten. Das Zusammenfassen von Transaktionen ist daher sinnvoll und meistens auch effizienter.

Häufiges Öffnen und Schließen von Datenbank-Verbindungen

Das Erstellen einer Datenbank-Verbindung und Starten des damit verbundenen dedizierten Prozesses auf dem Datenbank-Server ist bezüglich der erforderlichen Ressourcen und der notwendigen Zeit nicht zu unterschätzen. Schlimmster Fall ist eine Web-Applikation, die für jede Anfrage eine Datenbank-Verbindung öffnet und gleich wieder abbaut. Dies ist natürlich alles andere als optimal. In einer solchen Situation bietet sich der Einsatz eines Connection-Pools an.

Praxistipp: Häufiger Auf- und Abbau von Datenbank-Verbindungen ist zu vermeiden.

Wichtig: Last-Tests

Der Sinn und Zweck von Integrations- und Acceptance-Tests ist die Verifizierung der funktionalen Anforderungen, der Performance sowie der Applikationsstabilität. Es kann nicht oft genug betont werden, dass Performance-Tests dieselbe Wichtigkeit haben wie funktionale Tests. Liefert die Applikation ungenügende Antwortzeiten, ist das ebenso schlecht, als wenn die funktionalen Anforderungen nicht erfüllt werden. In beiden Fällen kann die Applikation unbrauchbar sein.

Auch wenn die Entwicklung, die Implementierung und die Durchführung von aussagekräftigen Integrations- und Acceptance-Tests keine einfache Aufgabe ist, darf man darauf nicht verzichten. Ohne Last-Tests hat man keine Gewähr, dass die Applikation der erwarteten Belastung standhält.

Praxistipp: Um die erwartete Performance zu verifizieren, sollte eine Applikation nie ohne Last-Tests entwickelt werden. Das Mantra sollte heißen: Was nicht getestet ist, wird nicht funktionieren.

Fazit

Leider wird häufig für das Design einer Applikation und Datenbank der Performance-Aspekt unterschätzt oder erst gar nicht berücksichtigt. Diese Praxis führt häufig zu unerwarteten Überraschungen. Der Autor hat in diesem kurzen Artikel die datenbankrelevanten Design-Probleme beschrieben, die in der Praxis am häufigsten auftreten. Leider sind diese Probleme nicht ohne einen größeren Aufwand nachträglich zu lösen. Es empfiehlt sich daher, die dargestellten Problempunkte rechtzeitig zu berücksichtigen.

Weitere Informationen

„Der Oracle DBA – Handbuch für die Administration der Oracle Database 11*g*R2“, Hanser Verlag (ISBN 978-3-446-42081-6).

Christian Antognini
Trivadis AG
christian.antognini@trivadis.com

Vorschau auf die nächste Ausgabe

Die Ausgabe 02/2012 hat das Schwerpunktthema:

„Apex“

Folgende Themen sind im Fokus:

- Grundlagen
- Ausblick
- Plug-ins
- Apex mobile
- Projekterfahrungen

Die Ausgabe 02/2012 erscheint am 2. April 2012.

Die weiteren Schwerpunktthemen und Temine stehen unter www.doag.org/go/doagnews