Kapitel 6

Die goldenen Regeln (in neuem Licht)

Bei der Programmierung von optimierten Datenbankanwendungen wird oft von den goldenen Regeln gesprochen. Wir zeigen Ihnen in diesem Kapitel, wie wichtig diese Regeln sind und dass diese nicht nur für die Datenbankschicht gelten, sondern auch in anderen Schichten Ihrer Applikation eine bedeutende Rolle spielen.

Der Ausdruck *goldene Regeln* bezeichnet eine Sammlung von Regeln für den performanten Datenbankzugriff. In diesem Kapitel dehnen wir die Anwendung dieser Regeln auf den kompletten Technologie-Stack aus und zeigen Ihnen, wie Sie durch die Einhaltung dieser Regeln bei der Programmierung die Performance Ihrer Anwendungen auf allen Ebenen (vom Frontend bis zur Datenbank) verbessern können. Zunächst stellen wir die fünf Regeln im ersten Abschnitt kurz allgemein vor, bevor wir auf die einzelnen Ebenen genauer eingehen. Wenn Sie mit den goldenen Regeln für Datenbankzugriffe bereits vertraut sind, können Sie den ersten Abschnitt überspringen. In den darauffolgenden übertragen wir die Regeln auf die folgenden Ebenen:

- Frontend-Programmierung und SAP Gateway
- Programmierung im ABAP-Backend

In jedem Abschnitt sehen wir uns anhand der goldenen Regeln an, was Sie bei der Programmierung beachten sollten. Wir zeigen Ihnen auch, mit welchen Werkzeugen Sie die Einhaltung dieser Regeln überprüfen können und wo es Ausnahmen gibt bzw. wann die Einhaltung der Regeln vernachlässigt werden kann, wenn dies keine große Rolle für die Performance spielt.

6.1 Die 5 goldenen Regeln, so wie Sie sie kennen

In diesem Abschnitt stellen wir die Regeln kurz allgemeingültig vor. Anhand eines kurzen Beispiels verdeutlichen wir jeweils, worum es bei den einzelnen Regeln geht.

Regel 1 und Regel 2 Bei der ersten und zweiten Regel geht es darum, die zu verarbeitende Datenmenge so klein wie möglich zu halten. Dabei bezieht sich Regel 1 darauf, die Anzahl der übertragenen Datensätze klein zu halten, und Regel 2 darauf, die Anzahl der von der Abfrage betroffenen Spalten zu reduzieren.

Nur das Nötige!

Sie planen, ein aufwendiges Gericht zu kochen, für das viele verschiedene Zutaten notwendig sind. Es wäre nun nicht sinnvoll, alle Zutaten, die Sie im Haus haben, für das Kochen bereitzustellen, sondern nur diejenigen aus dem Keller zu besorgen, die Sie auch wirklich brauchen. Würden Sie alle vorrätigen Zutaten bereitstellen, müssten Sie möglicherweise mehrfach (falls Sie nicht alles auf einmal tragen können) zwischen der Küche und dem Keller hin- und hergehen.

Sinnvoller ist es, sich vorher genau zu überlegen, welche Zutaten Sie brauchen und nur diese zu besorgen. Darüber hinaus sollten Sie je Zutat nur die benötigte Menge bereitstellen und nicht unnötig viele Zutaten transportieren. So benötigen Sie beispielsweise fünf festkochende Kartoffeln, zwei Zwiebeln, drei Karotten usw. und holen diese dann möglichst auf einmal aus dem Keller.

Mit diesen Regeln wird also sichergestellt, dass keine unnötigen Dinge (Daten) transportiert werden.

Regel 3 Bei der dritten Regel geht es darum, die Anzahl der Zugriffe auf eine Ressource so gering wie möglich zu halten. *Roundtrips* sollten also reduziert werden.

Nur ein Weg!

Wenn Sie wissen, welche Zutaten Sie genau benötigen, werden Sie diese möglichst auf einmal in einer Kiste aus dem Keller holen (bzw. gehen Sie nur so oft wie nötig zwischen der Küche und dem Keller hin und her, wenn Sie nicht alles auf einmal tragen können).

Mit dieser Regel wird sichergestellt, dass der notwendige *Overhead* (z. B. die Kellertreppe runter- und wieder hochzugehen) nur so oft wie nötig und nicht für jede einzelne Zutat (Zwiebeln, Karotten oder im Fall der Daten Zeilen und Objekte) anfällt.

Regel 4 Bei der vierten Regel geht es darum, den Suchaufwand so gering wie möglich zu halten. Dies wird bei Datenbankzugriffen z. B. durch Indizes gewährleistet.

Ordnung schaffen!

Ihre umfangreiche Gewürzsammlung haben Sie alphabetisch nach Namen sortiert im Regal stehen, damit Sie ein gewünschtes Gewürz schnell auffinden können.

Mit dieser Regel wird sichergestellt, dass Suchen effizient durchgeführt werden können. Sie brauchen so nicht jedes Mal alle Gewürze durchzusehen, bis sie das Gewünschte gefunden haben, sondern können mit »Sprüngen« gezielter suchen.

Regel 5 Bei der fünften Regel geht es darum, die Zugriffe auf (teure) zentrale Ressourcen zu minimieren.

Last vermeiden!

Sie besitzen eine Küchenmaschine, in der Sie nicht nur das Hauptgericht, sondern auch das Dessert zubereiten wollen. Um das Gerät nicht unnötig lange zu blockieren, können Sie für den Hauptgang das Gemüse und das Fleisch in einem Durchgang garen (anstatt dies getrennt und nacheinander zu tun). Das Gerät wird so nur so lange wie nötig belegt und steht danach für das Dessert zur Verfügung.

Oder alternativ: Zur Abwechslung gehen Sie regelmäßig in einem italienischen Restaurant essen. Um einen Tisch zu reservieren, werden Sie sich die Telefonnummer Ihres Lieblingsrestaurants nicht jedes Mal im Internet oder per Telefonauskunft heraussuchen, sondern diese auf einen Zettel notieren oder in Ihrem Telefon abspeichern.

Mit dieser Regel wird sichergestellt, dass zentrale (teure) Ressourcen nur so oft wie nötig und so kurz wie möglich benutzt werden. So können teure Zugriffe reduziert werden.

Keine Regel ohne Ausnahme

Zu sämtlichen Regeln gibt es allerdings auch Ausnahmen, bei denen die Regeln vernachlässigt werden können.

Verhältnismäßigkeit der Maßnahmen

Wenn Ihre Gewürzsammlung nur aus Salz, Pfeffer, Paprika und Curry besteht, können Sie auf eine alphabetische Sortierung verzichten. Sie werden das gewünschte Gewürz auch so schnell finden.

Schonung von Hardwareressourcen

Allen Regeln ist gemeinsam, dass Sie durch ihre Anwendung Verarbeitungsschritte einsparen und so Hardwareressourcen (CPU, Speicher, Netzwerk etc.) entlasten können.

6.2 Die goldenen Regeln angewandt auf die Datenbankperformance

Kommen wir nun zu den klassischen goldenen Regeln der Datenbankperformance. Diese Regeln gelten natürlich nicht nur für Datenbankzugriffe mit Open SQL, sondern auch dann, wenn Sie native Anwendungen mit Core Data Services (CDS) und ABAP Managed Database Procedures (AMDP) erstellen oder generell natives SQL benutzen. Die Anwendung einiger Regeln ist bei In-Memory-Datenbanken und spaltenorientierten Tabellen wichtiger geworden als bei herkömmlichen Datenbanken, andere haben auf den In-Memory-Plattformen an Bedeutung verloren. Auf diese Unterschiede gehen wir in Kapitel 10, »Das Ende aller Performanceprobleme? Der Umstieg auf SAP HANA«, näher ein.

6.2.1 Regel 1: Treffermenge klein halten

Anzahl der Zeilen reduzieren

Bei der ersten Regel geht es darum, die Anzahl der übertragenen Zeilen möglichst gering zu halten. Wenn diese Regel verletzt wird, erkennen Sie dies in der Transaktion ST05 im SQL-Trace (siehe Abschnitt 5.2.2, »SQL-Trace auswerten«) in der Spalte **Anzahl Sätze** bzw. in der Transaktion STAD (siehe Kapitel 2, »Am Anfang war der Statistiksatz«) in den Kennzahlen zur Datenbank. Dort werden in einem solchen Fall viele Datensätze ausgewiesen.

Je größer die Ergebnismenge ist und je breiter ein Datensatz bzw. die Summe der selektieren Felder dieser Tabelle ist (siehe Abschnitt 6.2.2, »Regel 2: Zu übertragende Datenmenge klein halten«), desto wichtiger ist die Einhaltung dieser Regel. Bei sehr kleinen Tabellen (Customizing- oder Konfigurationstabellen) könnte diese Regel noch vernachlässigt werden, beachten Sie aber, dass solche Tabellen mit der Zeit wachsen können. Auch bei kleinen Tabellen sind Sie auf der sicheren Seite, wenn Sie *immer* nur die tatsächlich benötigten Datensätze lesen, indem Sie die Treffermenge einschränken und somit so klein wie möglich halten. Um die Anzahl der übertragenen Zeilen zu reduzieren, gibt es verschiedene Möglichkeiten, die wir Ihnen im Folgenden vorstellen.

WHERE-Bedingungen verwenden

Die wichtigste Möglichkeit zur Reduzierung der Datensätze ist sicherlich die Angabe von Filtern in der `WHERE`-Bedingung. Wichtig ist hierbei, dass Sie alle Bedingungen an die Datenbank übergeben und nicht erst später in der weiteren Verarbeitung auf dem Applikationsserver Daten filtern, die nicht verarbeitet werden. Das folgende sehr einfache Beispiel veranschaulicht dies.

Es sollen alle Buchungen der ersten Klasse für eine bestimmte Fluggesellschaft aufgelistet werden. In Listing 6.1 wird der Filter für die Buchungsklasse erst nach der Selektion im ABAP-Programm angewendet. Dafür müssen in unserem System 294.496 Datensätze von der Datenbank gelesen werden, das dauert ca. 488 Millisekunden.

```
SELECT bookid, customid, class FROM sbook
WHERE carrid = 'LH' INTO @DATA(wa).
   CHECK wa-class = 'F'.
...
ENDSELECT.
```

Listing 6.1 Ermittlung aller Kunden für einen Flug in der ersten Klasse, Filter für die Flugklasse in der Applikation

In Listing 6.2 werden nun alle Filter bereits in der `WHERE`-Bedingung angegeben. Es werden nur noch 14.758 Datensätze übertragen und ca. 15 Millisekunden benötigt.

```
SELECT bookid, customid, class FROM sbook
WHERE carrid = 'LH' AND class = 'F' INTO @DATA(wa).
...
ENDSELECT.
```

Listing 6.2 Ermittlung aller Kunden für einen Flug in der ersten Klasse, alle Filter in der WHERE-Bedingung

An diesem einfachen Beispiel sehen Sie, wie sich der zusätzliche Filter in der Laufzeit bemerkbar macht. Es ist sehr wichtig, dass Sie alle Einschränkungen so früh wie möglich (in der `WHERE`-Bedingung bei der Datenselektion) angeben, so dass nicht benötigte Datensätze erst gar nicht gelesen und übertragen werden müssen.

Selbstverständlich ist das späte Filtern in einer Anwendung nicht immer so einfach zu erkennen wie in dem Beispiel in Listing 6.1. Schwierig zu erkennen sind sie insbesondere dann, wenn die Filter erst viel später und in ganz anderen Programmteilen angewendet werden oder wenn mit den ermittelten Datensätzen noch Berechnungen oder Transformationen durchgeführt und anschließend Einschränkungen darauf vorgenommen werden. An dieser Stelle möchten wir auf die neuen Möglichkeiten in Open SQL hinweisen, die wir in Kapitel 12, »Push the Code – neue Open-SQL-Funktionen und ABAP Core Data Services«, vorstellen. Seit dem Release SAP NetWeaver 7.40 gibt es viele Erweiterungen in Open SQL, die sehr viel komplexere Berechnungen und Transformationen auf der Datenbank erlauben, als Sie es in der Vergangenheit von Open SQL kannten.

In Regel 4 (siehe Abschnitt 6.2.4) kommen wir auf Indizes und deren Nutzung zu sprechen. Dabei spielen die Filter in der `WHERE`-Bedingung ebenfalls

wieder eine zentrale Rolle. So viel schon einmal vorab: Grundsätzlich gilt, dass Sie alle Filter in der WHERE-Bedingung angeben sollten, auch wenn es sich um negative Filter (<>, NOT etc.) handelt, da auch diese dazu beitragen, die Datenmenge auf das notwendige Minimum zu reduzieren.

Datensätze limitieren

Wenn Sie nur eine bestimmte Anzahl von Datensätzen benötigen, sollten Sie auf die Open-SQL-Zusätze UP TO n ROWS und SELECT SINGLE zurückgreifen, um die Übertragung von unnötigen Zeilen zu vermeiden. Im folgenden Beispiel werden die zehn Buchungen eines Fluges mit dem höchsten Gepäckgewicht ermittelt. In Listing 6.3 wird in einer SELECT ... ENDSELECT-Schleife nach dem zehnten Datensatz abgebrochen. Da diese Schleife gegenüber der Datenbank mit *Arrays* arbeitet (also die Datensätze nicht einzeln, sondern im Paket überträgt), wird nach dem ersten Paket abgebrochen. An dieser Stelle wurden aber schon zu viele Datensätze übertragen: In unserem System wurden 43.690 Datensätze selektiert und in 757 Millisekunden gelesen. Noch schlimmer wäre es freilich gewesen, alle Datensätze zu lesen und im ABAP-Programm zu sortieren, um dann die gewünschten zehn Datensätze weiterzuverarbeiten.

```
SELECT customid, luggweight
FROM sbook INTO @DATA(wa) ORDER BY luggweight DESCENDING.
...
   IF sy-dbcnt = 10. EXIT.
   ENDIF.
ENDSELECT
```

Listing 6.3 Limitierung im ABAP-Programm

UP TO n ROWS

In Listing 6.4 hingegen wird die Anfrage bereits auf der Datenbank auf zehn Datensätze limitiert. Es werden also nur die tatsächlich benötigten zehn Datensätze mit dem höchsten Gepäckgewicht übertragen. Die Ausführung dauert nun in unserem System nur noch 59 Millisekunden.

```
SELECT customid, luggweight
FROM sbook UP TO 10 ROWS INTO @DATA(wa)
ORDER BY luggweight DESCENDING.
...
ENDSELECT.
```

Listing 6.4 Limitierung auf der Datenbank

Wenn nur ein einziger Datensatz benötigt wird, können Sie auf den Open-SQL-Zusatz SELECT SINGLE zurückgreifen. Dies wird häufig gemacht, wenn die Existenz eines Datensatzes überprüft werden soll. Dass es noch besser geht (ganz ohne die Übertragung irgendeines Datensatzes bzw. Feldes),

sehen Sie in Abschnitt 12.1.3, »SQL-Ausdrücke verwenden«, anhand der Existenzchecks. Dort wird statt eines Feldes nur noch eine Konstante selektiert, die nicht von der Datenbank übertragen wird.

Aggregatfunktionen nur zur Aggregation

Keinesfalls sollten Sie auf Aggregatfunktionen wie die im folgenden Beispiel verwendete Funktion COUNT(*) zurückgreifen, wenn nur ein Existenzcheck benötigt wird!

DISTINCT

Wenn Sie nicht alle Felder des Primärschlüssels in der Feldliste haben, kann es zu doppelten Datensätzen in der Ergebnismenge kommen. Wenn dies nicht gewünscht ist, sollten Sie nur eindeutige Datensätze übertragen und den Open-SQL-Zusatz DISTINCT nutzen, damit keine unnötigen Datensätze übertragen werden.

Damit die Datenbank die Duplikate entfernen kann, müssen die Daten auf der Datenbank sortiert werden. Auch das hat seinen Preis. Daher lohnt sich der Einsatz des Zusatzes DISTINCT erst dann, wenn die Ergebnismenge damit signifikant reduziert werden kann (also beispielsweise 50 % weniger Daten übertragen werden müssen).

Aggregationen

Um Daten zu aggregieren, sollten Sie die Aggregatfunktionen aus Open SQL benutzen und nicht die Daten an das ABAP- Programm übertragen und die Aggregationen dort ausführen. In Listing 6.5 werden als Negativbeispiel alle Kürzel der Fluggesellschaften für jede Buchung an das ABAP-Programm übertragen. Auf unserem System liest die SELECT-Anweisung dazu 2.592.398 Zeilen, und das dauert ca. 1.952 Millisekunden.

Anschließend werden die Daten im ABAP-Programm nach Fluggesellschaften gruppiert, und die Anzahl der Buchungen wird addiert.

```
DATA cnt TYPE i.
SELECT carrid FROM sbook INTO TABLE @DATA(it).
LOOP AT it INTO DATA(all_bookings)
     GROUP BY  all_bookings-carrid.
  CLEAR cnt.
  LOOP AT GROUP all_bookings INTO DATA(booking).
    cnt = cnt + 1.
  ENDLOOP.
...
ENDLOOP.
```

Listing 6.5 Aggregation in ABAP

Verwendung von GROUP BY

Beachten Sie, dass die Gruppierung in diesem Beispiel über den Zusatz `GROUP BY` der `LOOP`-Anweisung durchgeführt wurde. Diese sehr bequeme Möglichkeit der Gruppierung in ABAP besteht seit ABAP 7.40 SP08. Selbstverständlich könnte man auch eine klassische Gruppenwechselverarbeitung der internen Tabelle anstoßen bzw. mit der Anweisung `AT NEW` arbeiten oder die Anweisung `COLLECT` benutzen (wenn man pro Buchung noch den konstanten Wert »1« mit selektiert, siehe Abschnitt 12.1.3, »SQL-Ausdrücke verwenden«).

Die einfachen `LOOP`-Varianten werden in diesem Beispiel eine bessere Laufzeit aufweisen, wir wollten Sie hier trotzdem auf den Zusatz `GROUP BY` für die `LOOP`-Anweisung aufmerksam machen, weil es sich um eine sehr mächtige (und gleichzeitige bequeme) Anweisung handelt, deren Funktionalität weit über einfache Gruppierungen hinausgeht. Näheres finden Sie in der SAP-Onlinedokumentation unter: *http://s-prs.de/v428001*

Aggregatfunktion verwenden

In Listing 6.6 wird zum Vergleich die Anzahl der Buchungen je Fluggesellschaft direkt über den Zusatz `GROUP BY` der `SELECT`-Anweisung und die Aggregatfunktion `COUNT(*)` auf der Datenbank ermittelt. So wird pro Fluggesellschaft nur eine Zeile mit der Anzahl der Buchungen an das ABAP-Programm übertragen. Die `SELECT`-Anweisung überträgt nur noch neun Datensätze und läuft ca. 15 Millisekunden.

```
SELECT carrid, COUNT(*) AS cnt
FROM sbook INTO @DATA(wa) GROUP BY carrid ORDER BY carrid.
...
ENDSELECT.
```

Listing 6.6 Aggregation in der Datenbank

HAVING

Abschließend möchten wir Sie noch auf den Zusatz `HAVING` der `SELECT`-Anweisung hinweisen. Mit diesem Zusatz ist es möglich, Filter für aggregierte Werte anzugeben. Sie können damit also von bereits aggregierten Ergebnissen nur diejenigen Zeilen übertragen, deren berechnete Aggregatwerte bestimmten Filtern genügen.

In Listing 6.7 verwenden wir den Code aus Listing 6.6 wieder und fügen den `HAVING`-Zusatz ein. Damit werden nur noch die Fluggesellschaften angezeigt, für die mehr als 300.000 Buchungen vorliegen. Die Zahl der Datensätze reduziert sich von neun auf zwei, die Laufzeit der Anfrage geht auf ca. 13 Millisekunden zurück.


```
SELECT carrid, COUNT(*) AS cnt
FROM sbook INTO @DATA(wa) GROUP BY carrid ORDER BY carrid
HAVING COUNT(*) > 300000.
...
ENDSELECT.
```

Listing 6.7 Aggregation in der Datenbank mit HAVING-Filter

Ähnlich wie bei dem `DISTINCT`-Zusatz lohnt sich der Einsatz der `HAVING`-Klausel vor allem dann, wenn es große Einschränkungen gibt und viele Datensätze nicht übertragen werden müssen.

In diesem Abschnitt haben wir Ihnen verschiedene Möglichkeiten zur Reduktion der gelesenen Datensätze gezeigt. Sie sollten nur die jeweils benötigte Anzahl der Datensätze an das ABAP-Programm übertragen. Dabei sollten Sie aber die gesamte Logik des Programms im Auge behalten und wenn möglich alle benötigten Datensätze auf einmal (d. h. in einer SQL-Anweisung) übertragen (siehe Abschnitt 6.2.3, »Regel 3: Anzahl der Zugriffe klein halten«). Bei sehr großen Datenmengen sollten Sie außerdem die Anzahl der übertragenen Spalten (siehe folgenden Abschnitt 6.2.2) und den Speicherbedarf beachten.

6.2.2 Regel 2: Zu übertragende Datenmenge klein halten

Anzahl der Spalten reduzieren

Bei der zweiten Regel geht es darum, die Anzahl der übertragenen Spalten so gering wie möglich zu halten. Je mehr Datensätze übertragen werden, desto wichtiger ist diese Regel. Für kleine Ergebnismengen oder bei `SELECT`-Anweisungen, die nur einen Datensatz lesen, kann die Regel vernachlässigt werden.

Eine Verletzung dieser Regel ist nicht so einfach zu erkennen. Die Spalte **Länge** in Transaktion ST05 bezieht sich nämlich auf den ganzen Datensatz und nicht auf die selektierten Felder. Das heißt, der Wert dieses Feldes ist nur bei einer `SELECT *`-Anweisung ein Indikator dafür, dass möglicherweise zu viele Daten übertragen wurden, nicht aber, wenn mit Feldlisten gearbeitet wird. Ein weiterer Indikator ist die Spalte **Sätze** zur `FETCH`-Anweisung, die Sie bei den Einzelsätzen in der Transaktion ST05 finden. Je kleiner die Breite der selektierten Felder, desto mehr Datensätze passen in ein Paket der `FETCH`-Anweisung. Dieser Zusammenhang bewirkt zum einen, dass weniger Daten von der Datenbank an den Applikationsserver übertragen werden müssen, zum anderen kann es innerhalb der Datenbank zu weiteren Optimierungen kommen, wenn z. B. alle selektierten Spalten in einem Index definiert sind. Dann genügt es, den Index zu lesen, und ein weiterer Zugriff auf die Tabelle ist nicht mehr nötig.

Feldlisten verwenden

Die Anzahl der Felder wird über die *Feldliste* der SELECT-Anweisung festgelegt. In der folgenden Anweisung werden alle Felder der Tabelle SBOOK für die Fluggesellschaft LH gelesen:

```
SELECT * FROM sbook WHERE carrid = 'LH' INTO TABLE @DATA(it).
```

Für diese Anweisung werden in unserem System 294.496 Datensätze in 1.785 Millisekunden übertragen. Pro FETCH-Anweisung werden 3.854 Sätze übertragen.

Wenn die Feldliste eingeschränkt wird, wie in der folgenden Anweisung, werden ebenfalls 294.496 Datensätze übertragen:

```
SELECT carrid, connid, fldate, bookid
FROM sbook WHERE carrid = 'LH' INTO TABLE @DATA(it).
```

Es dauert dann allerdings nur noch 499 Millisekunden, und pro FETCH-Anweisung werden 22.793 Datensätze übertragen, weil aufgrund der verringerten Breite pro Datensatz viel mehr Datensätze in ein Paket passen.

Projektions-Views verwenden

Außer über die Feldliste können Sie die Anzahl der Spalten auch über einen *Projektions-View* reduzieren, indem Sie z. B. im ABAP Dictionary einen View anlegen, der fünf Spalten einer Tabelle definiert, die 30 Spalten hat. Solche Views können Sie in verschiedenen ABAP-Programmen wiederverwenden. Eine SELECT *-Anweisung auf einen solchen View ist weit weniger kritisch als es eine SELECT *-Anweisung auf die Tabelle (wenn in beiden Fällen viele Datensätze selektiert werden).

INTO CORRESPONDING FIELDS OF

Der Zusatz INTO CORRESPONDING FIELDS OF der SELECT-Anweisung kann ebenfalls die Anzahl der selektierten Spalten reduzieren. Betrachten wir die folgende SELECT-Anweisung:

```
SELECT * FROM sbook WHERE carrid = 'LH'
INTO CORRESPONDING FIELDS OF TABLE @it.
```

Bei dieser Open-SQL-Anweisung handelt es sich zwar um eine SELECT *-Anweisung, allerdings wird diese in der Datenbankschnittstelle so umgeschrieben, dass in der nativen SQL-Anweisung, die zur Datenbank geschickt wird, nur diejenigen Spalten in die Feldliste aufgenommen werden, deren Namen auch in der Zielstruktur (hier die interne Tabelle it) vorhanden sind. Über die Transaktion ST05 und den SQL-Trace können Sie dieses Verhalten selbst gut nachvollziehen. Der Namensabgleich der Zielstruktur und der Datenbanktabelle kostet zwar ebenfalls etwas Zeit, diese Zeit kann aber vernachlässigt werden. Die Zeit, die durch den verbesserten Zugriff eingespart wird und der Effekt der reduzierten Datenmenge, der entsteht, wenn

weniger Spalten von der Datenbank übertragen werden, wiegen diese minimalen Kosten bei Weitem auf.

Nicht für Tabellenpuffer geeignet

[!]

Handelt es sich bei der Anweisung SELECT * um einen Zugriff auf eine Tabelle im SAP-Tabellenpuffer, sieht der Fall anders aus. Im Tabellenpuffer sind die Zugriffszeiten so gering, dass die Zeit für den Namensabgleich die Zugriffszeit insgesamt signifikant verschlechtern kann.

Einfluss der Spaltenorientierung

Bei spaltenorientierter Ablage der Daten entstehen im Vergleich zur zeilenorientierter Ablage weitere Aufwände, die berücksichtigt werden müssen. Bei spaltenorientierten Architekturen hat Regel 2 daher eine höhere Priorität als bei den klassischen zeilenorientierten Architekturen.

Auch bei Regel 2 sollten Sie die gesamte Programmlogik im Auge behalten, alle benötigten Felder auf einmal lesen und die Ergebnisse gegebenenfalls im Applikationsserver zwischenspeichern. Anstatt also Spalte 2 und 3 eines bestimmten Datensatzes an Aufrufstelle A und Spalte 4 und 5 desselben Datensatzes an Aufrufstelle B zu lesen, sollten Sie gleich alle Spalten 1 bis 4 an der Aufrufstelle A lesen. So wird die Anzahl der Zugriffe auf die Datenbank minimiert, was uns zur nächsten Regel bringt.

6.2.3 Regel 3: Anzahl der Zugriffe klein halten

Roundtrips reduzieren

Bei der dritten Regel geht es darum, den Overhead von Datenbankzugriffen zu minimieren, indem die Anzahl der Zugriffe auf die Datenbank auf ein Minimum eingeschränkt wird. Jeder Datenbankzugriff bringt einen gewissen Overhead mit sich. Das liegt z. B. daran, dass die Anwendung und die Datenbanksoftware meist nicht auf demselben physikalischen Server laufen, sondern durch ein Netzwerk getrennt sind. Dadurch können 300 (oder mehr) Millisekunden Roundtripzeit zu der Zeit für den Datenzugriff innerhalb der Datenbank hinzukommen.

Wenn diese Regel verletzt wird, erkennen Sie dies in der Transaktion ST05 im SQL-Trace in der Spalte **Gesamtzahl der Ausführungen** (siehe Abschnitt 5.2.2, »SQL-Trace auswerten«). Hier werden in diesem Fall viele Ausführungen ausgewiesen. Zu sehr vielen Zugriffen auf der Datenbank kommt es, wenn Datenbankanweisungen in *Schleifen* ausgeführt werden. Es hängt von der Art der Schleife ab, wie Sie solche Konstrukte optimieren können.

Geschachtelte SELECT-Anweisungen

Eine Fehlerquelle sind geschachtelte SELECT-Anweisungen, d. h., wenn innerhalb einer SELECT ... ENDSELECT-Anweisung weitere SELECT-Anweisungen ausgeführt werden. Im Beispiel in Listing 6.8 wird die innere SELECT-Anwei-

sung an die Tabelle SFLIGHT für jede darin enthaltene Buchung ausgeführt, um Daten für den jeweiligen Flug zu ermitteln. Die Laufzeit für beide SQL-Anweisungen beträgt in unserem System etwas über 620 Millisekunden.

```
SELECT carrid, connid, fldate FROM sbook
INTO @DATA(wa_sbook) WHERE customid = 1631
ORDER BY carrid, connid.
   SELECT SINGLE price, currency FROM sflight
   INTO @DATA(wa_sflight)
   WHERE carrid = @wa_sbook-carrid
   AND connid   = @wa_sbook-connid
   AND fldate   = @wa_sbook-fldate.
...
ENDSELECT.
```

Listing 6.8 SELECT-Anweisung in einer SELECT ... ENDSELECT-Schleife

JOIN-Anweisung

Das Beispiel aus Listing 6.8 lässt sich über einen JOIN optimieren (siehe Listing 6.9).

```
SELECT b~carrid, b~connid, b~fldate, s~price, s~currency
FROM sbook AS b
JOIN sflight AS s
  ON b~carrid = s~carrid
  AND b~connid = s~connid AND b~fldate = s~fldate
INTO @DATA(wa_sbook)
WHERE b~customid = 1631 ORDER BY b~carrid, b~connid.
...
ENDSELECT.
```

Listing 6.9 SELECT-Anweisung mit JOIN

Im Beispiel in Listing 6.9 wird nur noch eine einzige Anweisung zur Datenbank geschickt. Die Laufzeit dieser Anweisung auf unserem System beträgt damit nur noch ca. 11 Millisekunden.

[!]

Identische SELECT-Anweisungen

Generell kann es bei Lesezugriffen in Schleifen dazu kommen, dass ein und derselbe Datensatz mehrfach gelesen wird und so mehrfach von der Datenbank an die Anwendung übertragen wird. Solche Zugriffe werden als *identische Selects* bezeichnet. Diese Anweisungen sollten über eine geeignete Pufferung vermieden werden. Dazu erfahren Sie mehr in Abschnitt 6.2.5, »Regel 5: (Zentrale) Ressourcen entlasten«.

SELECT-Anweisungen im LOOP

Darüber hinaus gibt es SELECT-Anweisungen, die in einem LOOP über interne Tabellen im ABAP-Programm ausgeführt werden. In Listing 6.10 sehen Sie eine solche LOOP-Anweisung über die interne Tabelle SFLIGHT. Innerhalb der LOOP-Anweisung wird eine SELECT-Anweisung für die Tabelle SBOOK ausgeführt, um die Buchungsnummern für den jeweiligen Flug zu lesen. Die SELECT-Anweisung wird für jede Zeile der internen Tabelle SFLIGHT (in unserem Beispiel 1.296-mal) ausgeführt, was in unserem System zu einer Laufzeit von 1.846 Millisekunden führt.

```
LOOP AT i_sflight INTO DATA(wa_sflight).
  SELECT carrid, connid, fldate, bookid
  FROM sbook
  WHERE carrid = @wa_sflight-carrid
    AND connid = @wa_sflight-connid
    AND fldate = @wa_sflight-fldate
  APPENDING TABLE @DATA(it_sbook).
ENDLOOP.
```

Listing 6.10 SELECT in einer LOOP-Anweisung

FOR ALL ENTRIES

In Listing 6.11 wird dieses Beispiel mit dem Zusatz FOR ALL ENTRIES optimiert. Dabei wird die interne Tabelle I_SFLIGHT, die die Schlüsselwerte für den Zugriff auf die Tabelle SBOOK enthält, an die Datenbank übergeben. Je nach Konfiguration und Datenbankplattform geschieht dies an einem Stück oder in mehreren Blöcken. In unserem Beispiel gibt es nun nur noch eine SELECT-Anweisung an die Tabelle SBOOK. Die Laufzeit beträgt in unserem System damit nur noch 627 Millisekunden.

```
SELECT carrid, connid, fldate, bookid
FROM sbook
FOR ALL ENTRIES IN @i_sflight
WHERE carrid = @i_sflight-carrid
  AND connid = @i_sflight-connid
  AND fldate = @i_sflight-fldate
APPENDING TABLE @DATA(it_sbook).
```

Listing 6.11 SELECT mit dem Zusatz FOR ALL ENTRIES

Der Zusatz FOR ALL ENTRIES sollte immer dann zum Einsatz kommen, wenn keine JOIN-Anweisung möglich ist. Dies ist z. B. dann der Fall, wenn die interne Tabelle mit den Schlüsselwerten über Schnittstellen zur Verfügung gestellt wird.

[!]

Richtiger Einsatz von FOR ALL ENTRIES

Achten Sie immer darauf, dass die Treibertabelle (die interne Tabelle, die hinter `FOR ALL ENTRIES IN` angegeben ist) Zeilen enthält. Wird diese Tabelle leer übergeben, wird keine `WHERE`-Bedingung erzeugt, und es werden alle Datensätze von der Datenbanktabelle gelesen.

Achten Sie auch darauf, dass es in den verwendeten Datensätzen der Treibertabelle keine Duplikate gibt. Sie sollten generell nur eindeutige Daten zur Selektion an die Datenbank übergeben.

Falls es die Anwendungslogik erlaubt, nur ein Feld aus der Treibertabelle in der `WHERE`-Klausel zu referenzieren, sollten Sie dies machen. In solchen Fällen können die Datenbankplattformen in der Regel eine sogenannte `IN`-Liste verwenden, die eine sehr effiziente Verarbeitung der Anweisung ermöglicht.

Weitere Schleifentypen

Darüber hinaus gibt es natürlich noch verschiedene andere Schleifen, in denen `SELECT`-Anweisungen ausgeführt werden können. Hier entscheidet die Verarbeitungslogik, welche Maßnahmen zur Optimierung eingesetzt werden können. Häufig können hier ebenfalls der Zusatz `FOR ALL ENTRIES` oder Ranges-Tabellen eingesetzt werden. Beim Einsatz von Ranges-Tabellen müssen Sie allerdings darauf achten, dass diese nicht zu groß werden, weil hier keine Beschränkung der Anweisungsgröße gewährleistet ist.

Schreibende SQL-Anweisungen

Auch bei den schreibenden SQL-Anweisungen kann es zu Problemen kommen, wenn diese in Schleifen ausgeführt werden. Wenn Sie mehrere Datensätze aus einer internen Tabelle einfügen wollen, sollten Sie bei der `INSERT`-Anweisung den Zusatz `FROM TABLE` verwenden. Wenn die Daten, die Sie einfügen wollen, direkt aus einer Datenbanktabelle stammen, sollten Sie ab dem SAP-NetWeaver-Release 7.50 auch die Anweisung `INSERT ... FROM SELECT` verwenden (siehe Abschnitt 12.1.1, »Schnelles Kopieren von Daten mit INSERT ... FROM (SELECT...)«). Dabei kann die Verarbeitung der Daten komplett auf der Datenbank ausgeführt werden, ohne dass Daten an den Applikationsserver übertragen werden müssen.

Ändern und Löschen

Beim Ändern und beim Löschen von Daten steht der Zusatz `FROM TABLE` auch für die Anweisungen `UPDATE` und `DELETE` zur Verfügung. Verwenden Sie diese, wo möglich, anstatt Datensätze einzeln zu ändern. Dies kann zwar die Fehlerbehandlung komplizieren, ist aber aus Performancegesichtspunkten bei der Verarbeitung von großen Datenmengen auf jeden Fall vorzuziehen. Je nach Anwendungslogik können aber auch die Anweisungen `UPDATE ... SET ... WHERE` oder `DELETE ... WHERE` verwendet werden, die oft eine noch effizientere Verarbeitung ermöglichen, weil keine großen Datenmengen an die Datenbank übertragen werden.

6.2.4 Regel 4: Zu durchsuchende Datenmenge klein halten

Indizes benutzen

Bei der vierten Regel geht es darum, die zu durchsuchende Datenmenge einzuschränken. Wenn diese Regel verletzt wird, erkennen Sie dies z. B. in der Transaktion ST05 an einem hohen Wert in der Spalte **Dauer/Satz [µs]**, sofern es sich um einen einfachen Zugriff ohne Aggregationen, teure Berechnungen oder Sortierungen handelt (siehe Abschnitt 5.2.2, »SQL-Trace auswerten«).

Wir besprechen im folgenden Abschnitt zunächst die Grundlagen, wie die Funktionsweise eines Index, die Selektivität einer Abfrage, die wichtigsten Zugriffspfade und wie `WHERE`-Bedingungen aussehen müssen, damit sie von Indizes Gebrauch machen können.

Funktionsweise eines Index

Bei einem Index handelt es sich im Prinzip um eine Datenstruktur, die es über eine Sortierung der Daten ermöglicht, effizient auf diese Daten zuzugreifen. Dadurch müssen nicht alle Daten gelesen werden, sondern nur noch ein kleiner Bruchteil. Am einfachsten kann man sich einen Index wie ein Telefonbuch vorstellen. Hier werden die Daten nach Stadt, Nachname, Vorname und Adresse sortiert.

Binäre Suche

Mit diesen Informationen gelingt es ganz leicht, eine Telefonnummer zu finden, indem man eine *binäre Suche* anwendet. Sie können das Telefonbuch in der Mitte aufschlagen und sehen, ob die gesuchte Stadt in der alphabetischen Reihenfolge der Städte in der ersten oder zweiten Hälfte liegt. Die entsprechende Hälfte schlagen Sie wieder in der Mitte auf und prüfen erneut, in welcher Hälfte sich der gesuchte Eintrag befindet. So fahren Sie fort, bis Sie den gesuchten Eintrag gefunden haben. Dank der Sortierung sind nur wenige Blättervorgänge nötig, und Sie müssen nicht das ganze Telefonbuch von vorne nach hinten durchsuchen.

B*-Baumstruktur

In Datenbanksystemen handelt es sich bei Indizes häufig um *B*-Baumstrukturen*, die noch effizienter verwendet werden können als eine einfache sortierte Liste. Je nach Größe des Datenbestandes sind nur wenige Zugriffe (für den Wurzel- und die Blattknoten) nötig. Ein Beispiel für so eine B*-Baumstruktur sehen Sie in Abbildung 6.1.

Datenverteilung und Selektivität

Bei Suchvorgängen spielen die Datenverteilung und die Selektivität eine große Rolle. Wenn Sie beispielsweise nur den Nachnamen und die Stadt für eine Suche verwenden, wird die Kombination München/Maier ein sehr viel größeres Ergebnis liefern als die Kombination München/Gahm.

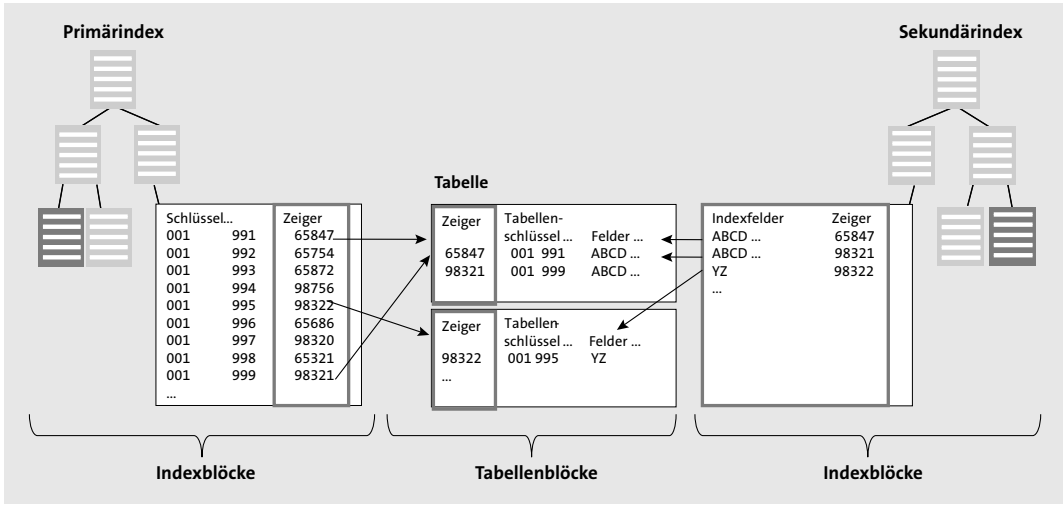

Abbildung 6.1 Schematische Darstellung eines Index

Bei ungleichen Datenverteilungen, wenn es also wie im Telefonbuch für verschiedene Nachnamen unterschiedlich viele Einträge gibt, spielt es also eine Rolle, nach welchem Wert gesucht wird und ob die Treffermenge groß oder klein ist. Bei Datenbanken werden generell *Statistiken* (Anzahl der verschiedenen Werte, Minimal- und Maximalwerte etc.) und bei ungleichen Datenverteilungen *Histogramme* (welche Werte kommen wie oft vor) genutzt, um die Daten zu beschreiben. Der Datenbankoptimierer kann anhand der Abfrage, der zur Verfügung stehenden Indizes und deren Statistiken den bestmöglichen Zugriffspfad festlegen.

Full Table oder Column Scan und Indexzugriffe

Bei den *Zugriffspfaden* unterscheiden wir prinzipiell zwischen einem *Full Table Scan* (bei In-Memory-Datenbanken einem *Column Scan*) und Zugriffen über einen Index. Bei den *Zugriffen* unterscheiden wir den *Index Range Scan* und den *Index Unique Scan*. Darüber hinaus gibt es noch viele weitere Zugriffsarten (Index Skip Scan, Index (Fast) Full Scan etc.), die abhängig von der Datenbankplattform sind. Es würde den Rahmen dieses Kapitels sprengen, auf alle einzugehen, weswegen wir uns hier auf die wichtigsten drei beschränken, die bei jeder Datenbank eine Rolle spielen:

- **Scan**
  Bei einem Scan (über eine Tabelle oder Spalte) werden immer alle Einträge gelesen, und es können erst nach dem Lesen Einträge gefiltert werden. Die Ergebnismenge kann also durchaus klein sein, obwohl zuvor alle Einträge verarbeitet werden müssen. Ein Scan muss immer dann ausgeführt werden, wenn kein Index vorhanden ist oder vorhandene Indizes nicht benutzt werden können, weil es keine `WHERE`-Bedingung gibt oder die `WHERE`-Bedingung eine Indexnutzung unmöglich macht.
- **Index Range Scan**
  Beim Index Range Scan wird ein Bereich aus einem Index gelesen. Es hängt von der Selektivität der `WHERE`-Bedingung und vom Aufbau des Index ab, wie groß dieser Bereich ist. Je größer der zu lesende Bereich ist, desto mehr Zeit wird bei der Ausführung der Abfrage benötigt.
- **Index Unique Scan**
  Beim Index Unique Scan wird nur ein (oder kein) Datensatz gelesen. Voraussetzung ist, dass alle Felder eines eindeutigen Index (z. B. dem Primärschlüssel) mit `=` spezifiziert und mit `AND`-Bedingungen verknüpft sind. Solche Abfragen sind sehr schnell ausführbar.

Für eine gute Performance sollte also der Scan auf großen Datenmengen vermieden werden. Ziel für alle häufig ausgeführten Abfragen sollte es sein, dass nur kleine Bereiche eines Index oder nur einzelne Zeilen gelesen werden müssen. Um das zu erreichen, müssen die `WHERE`-Bedingung und die Indexdefinition zusammenpassen. Im Folgenden gehen wir zunächst auf die `WHERE`-Bedingung ein und besprechen, wie diese aufgebaut sein muss, damit sie von einem Index Gebrauch machen kann. Anschließend gehen wir darauf ein, wie Indizes angelegt werden sollten.

Indexunterstützung von WHERE-Bedingungen

Die `WHERE`-Bedingungen einer Abfrage können im Index einschränkend oder filternd genutzt werden. Mit *einschränkenden Bedingungen* ist gemeint, dass bestimmte Bereiche des Index nicht gelesen werden müssen, weil sie mit der binären Suche »übersprungen« werden können. In klassischen Datenbanken können so Festplattenzugriffe (I/O = Input/Output) minimiert werden. In In-Memory-Datenbanken wird so der zu durchsuchende Bereich im Hauptspeicher reduziert.

Wenn Bedingungen nur gefiltert werden, heißt das, dass zunächst die Daten im Index gelesen werden müssen und anschließend *mit der Bedingung gefiltert* werden. Nachfolgende Aktivitäten wie Zugriffe auf Tabellendaten oder die Übertragung von Zeilen in der Ergebnismenge können von diesen Filtern dennoch reduziert werden. Eine Einschränkung der zu lesenden Daten aus dem Index kann aber nicht stattfinden.

Gleichheitsbedingungen und IN-Listen

Bei unserer Betrachtung des Aufbaus einer `WHERE`-Bedingung beginnen wir mit den einfachen Gleichheitsbedingungen (=) und den `IN`-Listen. Bei einer `IN`-Liste handelt es sich um mehrere =-Bedingungen, die mit `ODER` verknüpft

sind. Diese beiden Bedingungen können von Indizes einschränkend unterstützt werden. Dies ist der Fall, wenn die Bedingungen Folgendes erfüllen:

- Die Bedingungen betreffen die 1. Spalte im Index.
- Die Bedingungen betreffen eine Spalte im Index, bei der zuvor nur Spalten stehen, die ihrerseits auch mit = oder IN-Listen angegeben und mit der AND-Bedingung miteinander verknüpft sind.

Sind diese Punkte nicht erfüllt, wirken die Bedingungen nur filternd und reduzieren gegebenenfalls die Ergebnismenge.

Beispiele für Gleichheitsbedingungen

Wenn Sie für eine Suche im Telefonbuch die Stadt, den Nachnamen und den Vornamen kennen, müssen Sie nur wenige Einträge lesen. Die binäre Suche erlaubt es Ihnen, einen großen Teil der Einträge nicht zu lesen.

Wenn Sie für eine Suche im Telefonbuch die Stadt und den Vornamen kennen aber nicht den Nachnamen, müssen Sie nur den Bereich mit der Stadt durchsuchen, diesen aber komplett lesen.

Wenn Sie für eine Suche im Telefonbuch weder die Stadt noch den Nachnamen kennen, sondern nur den Vornamen, müssen Sie das ganze Telefonbuch nach dem Eintrag durchsuchen. Auch wenn es sich um einen sehr seltenen Vornamen handeln sollte, müssen Sie alle Einträge lesen.

Bereichsbedingungen

Bereichsbedingungen wie BETWEEN, kleiner (gleich), größer (gleich) oder LIKE mit einer Wildcard am Ende können von Indizes einschränkend benutzt werden, wenn folgende Punkte erfüllt sind:

- Die Bedingungen betreffen die 1. Spalte im Index.
- Die Bedingungen betreffen eine Spalte im Index, bei der zuvor nur Spalten stehen, die mit = oder IN-Listen angegeben sind und mit der AND-Bedingung miteinander verknüpft sind.

Beispiele für Bereichsbedingungen

Wenn Sie für eine Suche im Telefonbuch die Stadt kennen und der Nachnamen mit Y beginnen soll, können Sie Bereiche des Telefonbuches von Ihrer Suche ausschließen. So müssen Sie im Bereich der gesuchten Stadt nur die Seiten lesen, auf denen die Nachnamen mit Y beginnen.

Wenn Sie für eine Suche im Telefonbuch die Stadt haben und der Vorname mit Y beginnen soll, hilft Ihnen diese selektive Bedingung wenig. Zwar können Sie die Bereiche, die eine andere Stadt als die gesuchte haben, überspringen, aber innerhalb der gesuchten Stadt müssen Sie alle Einträge lesen.

In allen anderen Fällen können die Bereichsbedingungen nicht einschränkend, sondern nur filternd eingesetzt werden.

Ungleich- und LIKE-Bedingungen

Zum Schluss haben wir noch die WHERE-Bedingungen, die *nicht* von einem Index unterstützt werden. Das bedeutet, eine Einschränkung der zu lesenden Datenmengen ist nicht möglich. Die folgenden WHERE-Bedingungen wirken nur filternd; sie können erst nach dem Lesen angewandt werden:

- Ungleichbedingungen (<, >, NOT)
- LIKE-Bedingungen, die mit einer Wildcard beginnen

Beispiel für LIKE-Bedingungen

Wenn Sie für eine Suche im Telefonbuch die Stadt kennen und der Nachnamen ein M enthalten soll, müssen Sie im Bereich der gesuchten Stadt alle Einträge durchsuchen.

Diese Bedingungen sollten sie dennoch immer in der WHERE-Bedingung angeben. Zum einen können Sie Zugriffe auf Tabellendaten vermeiden, wenn schon im Index gefiltert werden kann, zum anderen können Sie die Ergebnismenge (und somit die Menge an zu übertragenden Daten, siehe Abschnitt 6.2.1, »Regel 1: Treffermenge klein halten«) reduzieren. Wenn eine WHERE-Bedingung nur aus LIKE-Bedingungen mit einer Wildcard am Anfang oder aus Ungleichbedingungen besteht, hat dies immer einen Scan zur Folge. Die Selektivität der Bedingungen bestimmt die Ergebnismenge.

Performantes Indexdesign

Indexperformance

Wie sollten Indizes aus Performancegesichtspunkten aufgebaut sein? Bei der Beantwortung dieser Frage unterscheiden wir zwischen den klassischen Datenbanken mit zeilenorientierter Ablage und In-Memory-Datenbanken mit spaltenorientierter Ablage. Generell gilt, dass Sie nur Indizes für Abfragen anlegen sollten, die häufig ausgeführt werden oder die in wichtigen Geschäftsprozessen vorkommen. Selten ausgeführte teure Abfragen mit Full Table Scans sind in der Regel kein Problem. Große Index Range Scans, die eventuell mit hohem CPU-Verbrauch einhergehen und parallel zu anderen wichtigen Prozessen laufen, können hingegen ein Problem darstellen. Sie sollten optimiert werden, da sie die Ressourcen (CPU, Cache) beanspruchen, die von den wichtigen Prozessen benötigt werden.

Kosten abwägen

Wenn Indizes angelegt werden, entstehen auch Kosten, die berücksichtigt werden müssen. So brauchen die Indizes Speicher. Außerdem benötigen schreibende SQL-Anweisungen auf Tabellen mit Indizes etwas mehr Zeit

und Ressourcen, da neben den Daten auch die Indizes gepflegt werden müssen.

Zeilenorientierte Datenbanken

Bei zeilenorientierten Datenbanken empfiehlt es sich, die Indizes so zu bauen, dass die selektiven Felder der Abfrage, die mit `=`, `IN` oder den Bereichsoperatoren abgefragt werden, im Index vorkommen. Weiterhin können Felder aufgenommen werden, die über die Filter die Datenmenge deutlich reduzieren. Ziel ist es, dass die Datenmenge nach Anwendung der Bedingungen im Index sehr klein ist, damit möglichst wenige Zugriffe auf die Tabelle nötig sind.

Bei spaltenorientieren Datenbanken wie SAP HANA gelten eigene Regeln, die in Kapitel 10, »Das Ende aller Performanceprobleme? Der Umstieg auf SAP HANA«, beleuchtet werden.

Indextabellen

Für einige wichtige Bewegungstabellen werden keine Sekundärindizes ausgeliefert. Stattdessen werden im System proprietäre Indextabellen gepflegt, die einen alternativen effizienten Zugriff erlauben. Um diese nutzen zu können, ist eine gewisse Grundkenntnis des SAP-Datenmodells notwendig. Die in Tabelle 6.1 aufgeführten SAP-Hinweise beschreiben die vorhandenen Indextabellen und deren Verwendung.

SAP-Hinweisnummer	Beschreibung
185530	Performance: Kundenentwicklungen in SD
191492	Performance: Kundenentwicklungen in MM/WM
187906	Performance: Kundenentwicklungen in PP und PM

Tabelle 6.1 Proprietäre Indextabellen im SAP-System

Beachten Sie diese SAP-Hinweise, wenn Sie Anwendungen entwickeln, die auf die in den Hinweisen genannten Tabellen zugreifen. Anstatt Sekundärindizes für die Bewegungstabellen anzulegen, sollte, wo es möglich ist, der alternative Zugriffspfad gewählt werden.

6.2.5 Regel 5: (Zentrale) Ressourcen entlasten

Bei der fünften Regel geht es darum, die Datenbank als zentrale Ressource zu entlasten. Wenn diese Regel verletzt wird, kann dies an verschiedenen Stellen sichtbar werden. Es kann zu Wartesituationen und/oder einer Überlastung von Speicher oder CPU kommen.

Datenbanksperren

Es gibt Wartesituationen beim Zugriff auf zentrale Ressourcen wie Nummernkreise oder zentrale Datensätze, die von parallelen Prozessen zeitgleich verändert werden sollen. Dies lässt sich meist nur mit datenbankabhängigen Monitoren erkennen, die die Datenbanksperren anzeigen.

Ein Indikator sind lang laufende schreibende Anweisungen, die eine Datenbanksperre benötigen. Neben den Anweisungen `INSERT`, `UPDATE`, `DELETE` und `MODIFY` gehört hier auch die Anweisung `SELECT FOR UPDATE` dazu. Näheres zur Analyse des kritischen Pfades finden Sie in Abschnitt 5.3, »Auf den Spuren des kritischen Pfades: Wie Sie der Ursache langer Datenbanksperren auf die Schliche kommen«.

Änderungen und Commit

Wenn große Datenmengen in parallelen Prozessen verarbeitet werden, ist es wichtig, Sperren nur so kurz wie möglich zu halten. Generell sollte also die Zeit zwischen der Änderung eines Datensatzes und dem Commit so kurz wie möglich sein. Es sollten zwischen den Datensatzänderungen und dem Commit nur die Verarbeitungsschritte stattfinden, die wirklich nötig sind. Des Weiteren trägt effizienter ABAP-Code, der die goldenen Regeln einhält, maßgeblich dazu bei, die Sperrzeit zu minimieren, wie Sie im folgenden Abschnitt noch sehen werden.

INSERT vor UPDATE

Des Weiteren sollten Sie `INSERT`-Anweisungen vor `UPDATE`-Anweisungen ausführen. `INSERT`-Anweisungen sperren sich nicht gegenseitig auf Datensatzebene (wenn in den parallelen Prozessen nicht dieselben Datensätze eingefügt werden, was normalerweise nicht der Fall ist). Bei `UPDATE`-Anweisungen können gegenseitige Sperren auftreten, wenn parallele Prozesse dieselben Datensätze ändern wollen, was in der Realität häufiger vorkommt. Daher sollten Sie zunächst die `INSERT` Anweisungen ausführen, bei denen nicht mit Sperren zu rechnen ist, und dann die `UPDATE`-Anweisungen. So wird die Zeit zwischen der ersten Datensatzsperre und dem Commit reduziert.

Identische Selects

Dann gibt es noch die sogenannten *identischen Selects*, also Abfragen, die (innerhalb einer Transaktion oder eines Prozesses) mehrfach dieselben (unveränderten) Datensätze von der Datenbank lesen. Dies lässt sich in der Transaktion ST05 in den Spalten **Redundante Identische Anweisungen** und **Identische Anweisungen [%]** erkennen.

Identische Zugriffe auf die Datenbank vermeiden Sie, indem Sie mehrfach benötigte Daten puffern. Hierzu gibt es viele Möglichkeiten, von denen wir auf die folgenden genauer eingehen:

- Pufferung in internen Tabellen/Verwendung von Lesebausteinen
- Verwendung des Tabellenpuffers

Pufferung in internen Tabellen

Bei der Pufferung der Daten in internen Tabellen werden die Ergebnisse einer SELECT-Anweisung in internen Tabellen gespeichert. Bevor die SELECT-Anweisung ausgeführt wird, wird zunächst in der internen Tabelle gesucht, ob die Daten schon vorhanden sind. Beachten Sie dabei, dass ein Ergebnis auch negativ sein kann. Wenn beispielsweise ein Datensatz mit einem bestimmten Schlüssel nicht gefunden wird, sollte der Schlüssel ebenfalls in der internen Tabelle mit einer Kennzeichnung »nicht gefunden« abgelegt werden. So vermeiden Sie auch identische Zugriffe auf Datensätze, die es nicht gibt. Achten Sie darauf, effizient auf die internen Tabellen zuzugreifen, wie es in Abschnitt 6.3.3, »Regel 4: Zu durchsuchende Datenmenge klein halten«, beschrieben wird. Damit die internen Tabellen nicht zu groß werden, sollten Sie ein Limit vorsehen.

Pufferung mit Lesebausteinen

Falls Sie eine solche Pufferung in mehreren Programmen benötigen, können Sie auch einen Funktionsbaustein entwickeln, der solch eine Pufferung in internen Tabellen implementiert. Solche *Lesebausteine* gibt es für viele Tabellen schon im SAP-Standard. Prüfen Sie, ob es bereits fertige geeignete Lesebausteine gibt, bevor Sie eine eigene Pufferung, sei es lokal im Programm oder als Lesebaustein, implementieren. Berücksichtigen Sie in diesem Zusammenhang den SAP-Hinweis 109533. Wenn Sie einen neuen Baustein entwickeln, beachten Sie dabei die zur lokalen Pufferung in internen Tabellen genannten Punkte.

[+]

Standardlesebausteine finden

Ein einfaches Beispiel für einen Lesebaustein ist MARV_SINGLE_READ. Sie finden solche Funktionsbausteine in der Transaktion SE37. Ihre Namen sind nach dem Muster *<TABELLE>*SINGLE*READ* aufgebaut. Es gibt auch Lesebausteine für Massenzugriffe. Diese finden Sie häufig mit dem Schlüsselwort *<TABELLE>*ARRAY*READ*.

Tabellenpuffer verwenden

Eine weitere Möglichkeit ist die Pufferung der Daten im Tabellenpuffer. Dies ist allerdings nur dann möglich, wenn die Tabellen folgenden Kriterien genügen:

- Durch die Synchronisation der Tabellenpuffer kann es zu kurzen Verzögerungen kommen, bis die aktuellsten Daten sichtbar sind. Dies muss für die Anwendung akzeptabel sein.
- Die Tabelle darf nicht zu groß sein (maximal 5 % des gesamten Platzes im Tabellenpuffer).
- Die Tabelle darf nur selten geändert werden, um Synchronisationen und Ladevorgänge zu reduzieren. Weniger als 1 % aller Zugriffe sollten ändernde Zugriffe sein.

Berücksichtigt man diese Kriterien, sind es häufig Tabellen mit Customizing-Daten oder kleinere Tabellen mit Stammdaten, die sehr selten geändert werden, die für eine Pufferung infrage kommen.

Die Pufferung von Tabellen im Tabellenpuffer ist auch dann noch relevant, wenn Sie eine In-Memory-Datenbank einsetzen, da die lokale Pufferung auf dem Applikationsserver um etwa den Faktor 10 schneller ist, als ein Datenbankzugriff.

[!]

ABAP-Sprachelemente, die eine Puffernutzung verhindern

Beachten Sie, dass es eine Reihe von SQL-Anweisungen gibt, die den Tabellenpuffer nicht nutzen können und von der Datenbank lesen:

- IN bei der Verwendung einer IN-Liste
- SELECT ... FOR UPDATE
- Aggregatfunktionen (z. B. SUM und GROUP BY)
- DISTINCT
- IS NULL
- Unterabfragen
- ORDER BY, wenn es sich bei den angegebenen Feldern nicht um den Primärschlüssel handelt
- JOIN-Anweisungen
- native SQL-Anweisungen (Generell dürfen gepufferte Tabellen nur über Open SQL geändert werden, da ansonsten die Puffersynchronisation umgangen wird und Inkonsistenzen entstehen können.)

Neben dem Tabellenpuffer gibt es noch weitere Möglichkeiten, um Daten zu puffern und so identische Zugriffe auf die Datenbank zu vermeiden. Dazu gehören der Shared Memory, der Shared Buffer und die *Shared Objects*. Daten, die direkt aus Datenbanktabellen stammen, puffern Sie jedoch am besten in internen Tabellen oder dem Tabellenpuffer. Die anderen Puffer werden häufig auch zum Datenaustausch zwischen verschiedenen Programmen oder Prozessen verwendet. In der ABAP-Onlinedokumentation finden Sie die Beschreibung der verschiedenen Puffer und sehen, wie diese verwendet werden können (siehe *http://s-prs.de/v428002*).

Nummernkreispuffer verwenden

Eine weitere Möglichkeit, Laufzeitprobleme mit Datenbanksperren zu vermeiden, ist die Verwendung der Nummernkreispufferung. Dadurch können Sperrprobleme im Umfeld der Nummernkreise (Tabelle NRIV) minimiert werden. Es hängt stark von der Anwendung und von länderspezifischen gesetzlichen Regelungen ab, ob und wie eine Nummernkreispufferung ver-

wendet werden darf. Dieses Thema ist daher immer in Zusammenarbeit mit Anwendungs- bzw. SAP-Komponentenexperten zu analysieren.

6.3 Die goldenen Regeln angewandt auf die ABAP-Performance

Die goldenen Regeln in ABAP-Programmen spielen auf verschiedenen Ebenen eine Rolle. Sowohl beim Remote Function Call (RFC) als auch beim Zugriff auf interne Tabellen sind die goldenen Regeln von Bedeutung. Weiterhin gehen wir in diesem Abschnitt noch auf den Zugriff auf den Tabellenpuffer ein. Wir erläutern für jede Regel, was Sie auf dieser Ebene beachten müssen. Sie erfahren auch, woran Sie erkennen können, ob die Regel eingehalten wurde, und sehen Beispiele für Optimierungen.

Kommunikation zwischen Systemen

Neben dem Datenbankzugriff spielt der Datenaustausch zwischen verschiedenen SAP- oder Nicht-SAP-Backend-Systemen eine große Rolle. Auf die Kommunikation zwischen Frontend und Backend gehen wir in Abschnitt 6.4, »Die goldenen Regeln angewandt auf die SAP-Gateway-Kommunikation«, und Abschnitt 6.5, »Die goldenen Regeln angewandt auf die SAPUI5-Frontend-Performance«, noch genauer ein. Bevor wir die Details der einzelnen Regeln erläutern, möchten wir noch kurz die Grundlagen der Kommunikation zwischen ABAP-Systemen per RFC ins Gedächtnis rufen.

Remote Function Call

Grundsätzlich gilt alles im Folgenden Gesagte auch für andere Protokolle (wie z. B. HTTP, Webservices, transaktionaler oder Background-RFC [tRFC] etc.). Diese Protokolle sind in der Regel nicht so »schlank« wie der synchrone RFC (sRFC), bieten dafür aber einen größeren Funktionsumfang. Der besprochene sRFC soll hier nur als ein Vertreter für die Kommunikation mit anderen Systemen behandelt werden. Die hier vorgestellten grundlegenden Prinzipien können also problemlos auf andere Formen der Kommunikation zwischen Systemen übertragen werden.

Kommunikations-Overhead

Neben der reinen Antwortzeit der Funktion, die mittels RFC aufgerufen wird, fallen noch Zeiten für die Kommunikation selbst an. Diese bezeichnet man auch als *Kommunikations-Overhead*. Je nach verwendetem Protokoll und Konfiguration des Systems bzw. Art und Umfang der Daten spielen hierbei verschiedene Faktoren eine Rolle:

- Bestimmung des Empfangssystems und Erstellen des Message Headers
- Mapping von Daten
- (De-)Serialisierung von Daten
- Ver- und Entschlüsselung von Daten
- (De-)Komprimierung von Daten

Latenzzeit und Bandbreite

Zu der für diese Abläufe anfallenden Zeit kommt die Zeit für den physischen Datentransfer. Dieser wird wiederum maßgeblich von folgenden zwei Faktoren beeinflusst:

- *Latenzzeit* (die Zeit, die ein Signal vom Sender zum Empfänger benötigt) bzw. die Zeit für einen Roundtrip (= Latenzzeit mal 2)
- *Bandbreite* (das Maß der Datenübertragungsrate, bzw. die Übertragungsgeschwindigkeit)

Als ABAP-Softwareentwickler haben Sie auf die Latenzzeit und die Bandbreite keinen Einfluss. Sie sollten sich aber über die folgenden Punkte im Klaren sein. Je nach Entfernung der beteiligten Systeme können die Zeiten für einen Roundtrip zwischen 5 und 400 Millisekunden liegen. Bei Satellitenverbindungen liegen die Zeiten für einen Roundtrip sogar über 500 Millisekunden. Daher ist es von entscheidender Bedeutung, wie viele Roundtrips von einer Anwendung gemacht werden.

Darüber hinaus spielt die Menge der übertragenen Daten eine Rolle. Je kleiner die zu übertragende Datenmenge ist, desto schneller kann diese übertragen werden. Wenden wir uns daher diesen Punkten zu, da Sie diese als ABAP-Entwickler beeinflussen können.

6.3.1 Regel 1 und 2: Datenmenge klein halten

Zeilen und Spalten reduzieren

Wir fassen in diesem Abschnitt die Regeln 1 und 2 zusammen und behandeln sie allgemein unter dem Aspekt, die verarbeitete Datenmenge klein zu halten. Für den RFC, aber auch für die internen Tabellen und Tabellenpuffer bietet es sich an, die beiden ersten Regeln in einem Abschnitt zu behandeln.

Datenmenge beim Remote Function Call reduzieren

Um die Effekte der hier behandelten Maßnahmen zu verdeutlichen, nehmen wir ein kleines Testprogramm zur Hand. Das Testprogramm überträgt eine `WHERE`-Bedingung und eine Liste mit Spalten an einen RFC-Baustein. In diesem Baustein werden die Inhalte einer vollständig gepufferten Tabelle gemäß der `WHERE`-Bedingung gelesen und in einer internen Tabelle zurückgegeben. Das aufrufende System steht in Walldorf und der RFC-Server, der die Daten selektiert und überträgt, in Shanghai. Wir haben zwischen Sender und Empfänger also ein *Wide Area Network* (WAN), in dem wir die Effekte deutlich sehen können. Wir gehen aber auch auf LAN-Verbindungen (Local Area Network) ein.

Datenmenge im WAN

Im ersten Test selektieren wir 4.393 Zeilen mit acht Spalten (1.970 Bytes Strukturbreite, die Daten sind aber dünn besetzt, d. h., vier Spalten haben

überwiegend Initialwerte). Die Daten werden über den Parameter TABLES übertragen. Die RFC-Verbindung nutzt das klassische tRFC-Protokoll (transaktionaler RFC).

Wir erhalten mit dem RFC-Trace, den wir mit der Transaktion ST05 erstellt haben, die Daten in der ersten Zeile in Tabelle 6.2. Der Aufruf dauerte ca. 1 Sekunde. Das Ausführen der Funktion im RFC-Server (in Tabelle 6.2 nicht gezeigt) dauerte weniger als 15 Millisekunden. Es wurden 237.888 Bytes gelesen, und das Versenden der Daten hat ca. 669 Millisekunden gedauert. Die restlichen (in Tabelle 6.2 nicht dargestellten) Zeiten beinhalten die Latenzzeit (die hier im WAN recht hoch ist) und die Zeiten für das Senden und Komprimieren der Empfangs- und Rückgabedaten.

Wir wiederholen den Test, indem wir nur 204 Zeilen mit acht Spalten selektieren. Die Ergebnisse sehen Sie in der zweiten Zeile der Tabelle 6.2. Die empfangene Datenmenge hat sich drastisch reduziert (auf 9.762 Bytes) und mit ihr die Zeit für das Empfangen und Komprimieren der Daten. Die Aufrufzeit beträgt nun noch ca. 233 Millisekunden.

Im dritten Test reduzieren wir die selektierten Spalten. Wir lesen nun nur noch 204 Zeilen mit vier Spalten (530 Bytes). Die Ergebnisse sehen Sie in der dritten Zeile in Tabelle 6.2. Wie erwartet, reduziert sich die empfangene Datenmenge weiter (auf 5.952 Bytes). Ebenso reduziert sich die Zeit für das Komprimieren und Empfangen der Daten deutlich. Die gesamte Aufrufzeit ist mit 221 Millisekunden allerdings nur geringfügig besser als im vorherigen Test. Das liegt daran, dass die Roundtrip-Zeit im WAN in unserem Setup (Walldorf – Shanghai) bei ca. 215 Millisekunden liegt. Diesen Aspekt behandeln wir in Abschnitt 6.3.2, »Regel 3: Anzahl der Zugriffe klein halten«.

WAN TABLES-Parameter tRFC-Protokoll	Aufrufzeit [µs]	Empfangene Datenmenge [Bytes]	Empfangszeit [µs]
4.393 Zeilen, 8 Spalten	1.072.368	237.888	669.514
204 Zeilen, 8 Spalten	233.052	9.792	966
204 Zeilen, 4 Spalten	220.921	5.952	529

Tabelle 6.2 Beispiel für die übertragene Datenmenge im WAN mit TABLES-Parameter und tRFC-Protokoll

Die Menge der übertragenen Daten beeinflusst also maßgeblich die Geschwindigkeit der RFC-Aufrufe im WAN.

Datenmenge im LAN

Wenn wir in einem LAN einen ähnlichen Test durchführen, erhalten wir die Werte von Tabelle 6.3. Beachten Sie, dass sich die Datenmengen im Vergleich zum Test im WAN (siehe Tabelle 6.2) aufgrund der verschiedenen Systeme etwas unterscheiden. Die Zeiten sind im lokalen Netzwerk deutlich kürzer. Dies liegt an der höheren Bandbreite und der geringeren Latenzzeit. Dennoch lassen sich hier dieselben Effekte auf einem anderen Niveau beobachten. Auch hier sehen Sie, dass die Laufzeiten im LAN maßgeblich von der übertragenen Datenmenge abhängen.

LAN TABLES-Parameter tRFC-Protokoll	Aufrufzeit [µs]	Empfangene Datenmenge [Bytes]	Empfangszeit [µs]
7.428 Zeilen, 8 Spalten	170.362	383.565	34.297
203 Zeilen, 8 Spalten	4.743	9.805	785
203 Zeilen, 4 Spalten	2.052	5.965	330

Tabelle 6.3 Beispiel für die übertragene Datenmenge im LAN mit TABLES-Parameter und tRFC-Protokoll

Beim RFC sollten Sie also die zu übertragende Datenmenge so gering wie möglich halten. Prinzipiell sollten keine unnötigen Daten übertragen werden. Schränken Sie die Datenmenge soweit wie möglich ein, und übertragen Sie nur die tatsächlich benötigten Spalten und Zeilen per RFC.

Neben der Reduktion über die Daten in der Applikation gibt es noch weitere Aspekte, die auf die Datenmenge und die Performance einen Einfluss haben. Es geht dabei um die Frage, wie die Daten übertragen werden.

TABLES oder IMPORTING/ EXPORTING

In den vorangehenden Beispielen haben wir den TABLES-Parameter verwendet. Wenn die Daten auf diese Weise übertragen werden, werden sie immer komprimiert. Diese *Komprimierung* kostet aber auch Zeit. Achten Sie bei den folgenden Tests auf die übertragene Datenmenge und die Zeit. Bei der Verwendung von IMPORTING-/EXPORTING-Parametern werden die Daten nicht mehr komprimiert. Sie werden in eine XML-artige Struktur überführt. Neben den reinen Daten kommen noch die sogenannten *Tags* zu der übertragenen Datenmenge hinzu, die Beginn und Ende der Zeilen und Spalten markieren, zwischen denen die Nutzdaten stehen.

Die Datenmenge ist in diesem Beispiel deutlich größer, da wir die Daten per IMPORTING-/EXPORTING-Parameter übertragen und nicht mehr den TABLES-Parameter verwenden. Wenn Sie die Daten in Tabelle 6.4 mit denen in Tabelle 6.2 vergleichen, sehen Sie, dass im WAN die Datenmengen größer werden und die Laufzeiten steigen.

WAN IMPORTING-Parameter tRFC-Protokoll	Aufrufzeit [µs]	Empfangene Datenmenge [Bytes]	Empfangszeit [µs]
7.428 Zeilen, 8 Spalten	4.024.854	1.125.695	3.788.742
203 Zeilen, 8 Spalten	221.463	55.043	883
203 Zeilen, 4 Spalten	217.171	33.798	566

Tabelle 6.4 Beispiel für die übertragene Datenmenge im WAN mit IMPORTING-Parameter und tRFC-Protokoll

Vergleich der WAN- und LAN-Ergebnisse

Im LAN ist es hingegen anders. Vergleichen Sie die Daten in Tabelle 6.5 mit denen in Tabelle 6.3. Die Datenmenge ist nun beträchtlich höher, aber die Laufzeit ist kleiner. Dies lässt sich zum einen durch die höhere Bandbreite und zum anderen dadurch erklären, dass die Zeit für die Komprimierung entfällt. Des Weiteren haben wir in unserem Beispiel eine eher dünne Besetzung der Daten: Vier von acht Spalten waren in den meisten Fällen initial. Auch diese Initialwerte müssen komprimiert werden, während sie im Beispiel ohne die Komprimierung nur getagt werden, d. h., hier existiert dann nur eine Anfangs- bzw. Endmarkierung ohne weiteren Inhalt.

LAN IMPORTING-Parameter tRFC-Protokoll	Aufrufzeit [µs]	Empfangene Datenmenge [Bytes]	Empfangszeit [µs]
7.428 Zeilen, 8 Spalten	46.638	1.900.979	31.886
203 Zeilen, 8 Spalten	2.196	54.796	936
203 Zeilen, 4 Spalten	1.687	33.650	554

Tabelle 6.5 Beispiel für die übertragene Datenmenge im LAN mit IMPORTING-Parameter und tRFC-Protokoll

Art der übertragenen Daten

Es spielt auch eine Rolle, welche Art von Daten übertragen werden. Als Extreme seien hier genannt:

- wenige Spalten mit großer Länge (wenig Overhead für Tags)
- viele Spalten mit kurzer Länge (viel Overhead für Tags)
- Füllgrad der Spalten gering (viel Overhead für Tags, volle Komprimierungskosten)
- Füllgrad der Spalten hoch (wenig Overhead für Tags, volle Komprimierungskosten)

Deltaverarbeitung nutzen

Ein weiterer Aspekt ist die Deltaverarbeitung. Wenn z. B. 5.000 Zeilen per RFC übertragen werden und nur eine Zeile verändert wird und dann alle 5.000 Zeilen zurückübertragen werden, erlaubt der `TABLES`-Parameter eine sogenannte *Deltaverarbeitung*. Es wird dann nur die geänderte Zeile zurückübertragen. Diese Möglichkeit steht nur beim `TABLES`-Parameter zur Verfügung. Wenn Sie so ein Szenario umsetzen müssen, sollten Sie, wenn möglich, den `TABLES`-Parameter verwenden.

Welchen Parameter verwenden?

Die Frage, wie Sie Ihre Daten übertragen sollten (ob per `TABLES`-oder `EXPORTING`-/`IMPORTING`-Parameter) ist also nicht so einfach zu beantworten. Es kommt wie immer darauf an. Wir haben Ihnen in den Beispielen dieses Abschnitts die Vor- und Nachteile der verschiedenen Arten gezeigt. Im LAN (bzw. bei großen Bandbreiten) kann sich eine Komprimierung durchaus negativ auf die Laufzeiten auswirken. Im WAN hingegen ist die fehlende Komprimierung ein Grund für steigende Laufzeiten.

In den folgenden Fällen *müssen* Sie die `IMPORTING`-/`EXPORTING`-Parameter verwenden:

- wenn Sie andere Tabellentypen als Standardtabellen übertragen wollen
- wenn Sie tiefe Strukturen (Tabellen in Tabellen etc.) übertragen wollen
- wenn sich die Datenstrukturen für Sender und Empfänger unterscheiden (Mit dem `TABLES`-Parameter sind Unterschiede am Ende der Struktur möglich, wenn Sie z. B. im Sender acht Felder und im Empfänger zehn Felder haben und sich die beiden »neuen Felder« am Ende der Struktur befinden. In diesem Fall können Sie acht Felder per RFC und `TABLES`-Parameter übertragen.)

Übertragungsprotokoll basXML

Eine weitere Option, die Datenmenge zu reduzieren, ist es, *Binary ABAP Serialization XML* (basXML) als Übertragungsprotokoll zu verwenden. Es handelt sich dabei um ein binäres komprimiertes XML-artiges Format. Um von dieser Möglichkeit Gebrauch zu machen, müssen folgende Voraussetzungen erfüllt sein:

- Der aufgerufene RFC-Baustein muss basXML-fähig sein. Dies erkennen Sie in Transaktion SE37 auf der Registerkarte **Eigenschaften**.
- Sowohl der Sender als auch der Empfänger müssen basXML beherrschen.
- In der Transaktion SM59 muss basXML für die Destination auf der Registerkarte **Spezielle Optionen** aktiviert sein.

Option »basXML erzwungen«

Wenn Sie in der Transaktion SM59 für die Destination die Option **basXML erzwungen** auswählen, muss der gerufene Funktionsbaustein nicht basXML-fähig sein. Es genügt, wenn das aufgerufene System basXML unterstützt.

Komprimierungsergebnis

Die Komprimierung mit basXML wirkt sowohl auf den TABLES-Parameter als auch auf IMPORTING-/EXPORTING-Parameter. Mit ihr können im WAN bessere Komprimierungsergebnisse und schnellere Übertragungen realisiert werden, wie Tabelle 6.6 für unser Testprogramm zeigt.

Art des Aufrufes	Aufrufzeit [µs]	Empfangene Datenmenge [Bytes]	Empfangszeit [µs]
WAN mit TABLES-Parameter und tRFC-Protokoll 4.393 Zeilen, 8 Spalten	1.072.368	237.888	669.514
WAN mit TABLES-Parameter und basXML-Protokoll 4.393 Zeilen, 8 Spalten	672.049	127.757	403.492
WAN mit IMPORTING-Parameter und tRFC-Protokoll 7.428 Zeilen, 8 Spalten	4.024.854	1.125.695	3.788.742
WAN mit IMPORTING-Parameter und basXML-Protokoll 7.428 Zeilen, 8 Spalten	674.942	127.820	402.440

Tabelle 6.6 Verwendung von tRFC und basXML im WAN

Auch im LAN kann die Komprimierung verbessert werden. Die Komprimierung mit basXML kostet aber auch Zeit, so dass im LAN (bei großen Bandbreiten) ein nicht komprimierter RFC-Aufruf schneller sein kann (siehe Tabelle 6.7).

Fast RFC

Mit dem SAP NetWeaver Application Server (AS) ABAP 7.51 steht Ihnen noch eine neue *schnelle Serialisierung* zur Verfügung, die auch *Fast RFC* genannt wird. Diese Option aktivieren Sie in der RFC-Verbindung in Transaktion SM59 auf der Registerkarte **Spezielle Optionen** beim Übertragungsprotokoll.

Art des Aufrufes	Aufrufzeit [µs]	Empfangene Datenmenge [Bytes]	Empfangszeit [µs]
LAN mit TABLES-Parameter und tRFC-Protokoll 4.393 Zeilen, 8 Spalten	170.362	383.565	34.297
LAN mit TABLES-Parameter und basXML-Protokoll 4.393 Zeilen, 8 Spalten	65.330	206.322	39.314
LAN mit IMPORTING-Parameter und tRFC-Protokoll 7.428 Zeilen, 8 Spalten	46.638	1.900.979	31.886
LAN mit IMPORTING-Parameter und basXML-Protokoll 7.428 Zeilen, 8 Spalten	70.303	206.334	40.806

Tabelle 6.7 Verwendung von tRFC und basXML im LAN

Wie bei basXML müssen sowohl der Sender als auch der Empfänger dieses Protokoll beherrschen. Eine weitere Einstellung am RFC-Baustein ist nicht nötig.

Der Fast RFC bietet in LAN-Szenarien (wenn das Kontrollkästchen **Langsame RFC Verbindung** für die RFC-Destination auf der Registerkarte **Spezielle Optionen** nicht gesetzt ist) eine noch schnellere Komprimierung. Die Komprimierungsrate ist nur geringfügig schlechter als bei basXML (siehe Tabelle 6.8).

Art des Aufrufes	Aufrufzeit [µs]	Empfangene Datenmenge [Bytes]	Empfangszeit [µs]
LAN mit TABLES-Parameter und tRFC-Protokoll 4.393 Zeilen, 8 Spalten	170.362	383.565	34.297
LAN mit TABLES-Parameter und basXML-Protokoll 4.393 Zeilen, 8 Spalten	65.330	206.322	39.314

Tabelle 6.8 tRFC, basXML und schnelle Serialisierung im LAN

Art des Aufrufes	Aufrufzeit [µs]	Empfangene Datenmenge [Bytes]	Empfangszeit [µs]
LAN mit TABLES-Parameter und schneller Serialisierung 7.428 Zeilen, 8 Spalten	45.552	268.446	17.894
LAN mit IMPORTING-Parameter und tRFC-Protokoll 7.428 Zeilen, 8 Spalten	46.638	1.900.979	31.886
LAN mit IMPORTING-Parameter und basXML-Protokoll 7.428 Zeilen, 8 Spalten	70.303	206.334	40.806
LAN mit IMPORTING-Parameter und schneller Serialisierung 7.428 Zeilen, 8 Spalten	44.493	268.441	17.466

Tabelle 6.8 tRFC, basXML und schnelle Serialisierung im LAN (Forts.)

Nachdem wir die wichtigsten Punkte für den RFC behandelt haben, schauen wir uns nun das Thema übertragene Datenmenge auf weiteren Ebenen an.

Datenmenge beim Zugriff auf interne Tabellen reduzieren

Beim Verarbeiten von internen Tabellen gibt es verschiedene Möglichkeiten, die zu verarbeitende Datenmenge – in diesem Fall die zu kopierende Datenmenge – zu reduzieren.

LOOP über interne Tabellen

Bei einer LOOP-Schleife über interne Tabellen sollten Sie generell mit Feldsymbolen oder Referenzen arbeiten. So müssen überhaupt keine Daten kopiert werden, was sich positiv auf die Laufzeit auswirkt. Ein LOOP über eine interne Tabelle mit 80 Spalten und ca. 7.000 Zeilen dauert in unserem Testprogramm mit der Anweisung LOOP INTO bei 1.000 Ausführungen ca. 2,4 Sekunden. Arbeiten wir mit Feldsymbolen (LOOP ASSIGNING <FIELDSYMBOL>) sind es bei 1.000 Ausführungen nur noch ca. 300 Millisekunden. Der Einsatz von Feldsymbolen lohnt sich noch mehr, wenn die interne Tabelle auf einer tiefen Struktur basiert, also Tabellen in der Tabelle enthält.

Wenn Sie innerhalb einer Schleife über eine interne Tabelle ändernd auf die Daten zugreifen, sollten Sie ebenfalls Feldsymbole verwenden. Sie ersparen

sich so unnötige Kopiervorgänge von Daten und ändern diese direkt per Referenz. Ein LOOP über eine interne Tabelle mit 80 Spalten und ca. 7.000 Zeilen dauert in unserem Testprogramm mit den Anweisungen LOOP INTO und MODIFY bei 1.000 Ausführungen ca. 9,6 Sekunden. Arbeiten wir mit Feldsymbolen (LOOP ASSIGNING <FIELDSYMBOL>) sind es bei 1.000 Ausführungen nur noch ca. 1,6 Sekunden. Der Einsatz von Feldsymbolen lohnt sich auch in diesem Fall noch mehr bei tiefen Strukturen.

READ-Anweisung

Ähnlich wie mit der LOOP-Anweisung verhält es sich mit der READ-Anweisung. Auch hier sollten Sie wenn möglich mit Datenreferenzen arbeiten anstatt die Daten zu kopieren. Dies lohnt sich in der Regel bei breiten Strukturen (>500 kB Zeilenbreite) oder wenn eine interne Tabelle tiefe Strukturen enthält.

Falls Sie beim READ-Zugriff gar nicht an den Inhalten der Zeile interessiert sind, sondern nur wissen möchten, ob der Datensatz existiert (und nur den Return-Code sy-subrc auswerten), können Sie auch vom Zusatz TRANSPORTING NO FIELDS der READ-Anweisung Gebrauch machen. Generell werden die Kopierkosten beim Zugriff auf interne Tabellen aber deutlich von den Kosten für die Suche der Daten übertroffen. Darauf gehen wir in Abschnitt 6.3.3, »Regel 4: Zu durchsuchende Datenmenge klein halten«, näher ein.

Zugriff auf Tabellenpuffer

Beim Zugriff auf Tabellenpuffer ist die Zahl der übertragenen Zeilen ebenfalls von Bedeutung. Wenn wir beispielsweise von einer vollständig gepufferten Tabelle alle Spalten (80) und alle Zeilen (ca. 7.000) lesen und dies mit einem Zugriff vergleichen, der nur ca. 200 Zeilen mit allen Spalten (80) liest, erhalten wir bei 1.000 Zugriffen einen Laufzeitunterschied von ca. 5 Sekunden.

Die Zahl der übertragenen Spalten hingegen ist beim Zugriff auf den Tabellenpuffer weniger von Bedeutung. Wenn wir von einer einzelsatzgepufferten Tabelle nur eine statt vier Spalten bei einem Datensatz lesen, erhalten wir erst bei 100.000 Zugriffen einen Laufzeitunterschied von ca.13 Millisekunden. Wenn wir von einer vollständig gepufferten Tabelle von allen Datensätzen (ca. 7.000) nur fünf statt 80 Spalten lesen, erhalten wir bei 1.000 Zugriffen einen Laufzeitunterschied von ca. 1,6 Sekunden. Wie Sie sehen, fallen die Effekte je nach Datenmenge unterschiedlich groß aus. Es bedarf aber immer einer größeren Anzahl von Zugriffen, um die Effekte überhaupt sichtbar zu machen.

Parameterübergaben

Generell sollten Sie bei Parameterübergaben die Übergabe per Referenz bevorzugen, da so keine Daten kopiert werden müssen. Dies gilt insbesondere bei großen Datenmengen oder tiefen Datenstrukturen (Tabellen in Tabellen).

6.3.2 Regel 3: Anzahl der Zugriffe klein halten

Anzahl der Aufrufe reduzieren

Bei der dritten Regel sind in erster Linie die Aufrufe von (RFC-)Funktionsbausteinen von Bedeutung. Wir betrachten daher zunächst die RFC-Aufrufe. In der Einführung zu Abschnitt 6.3 haben wir bereits den Kommunikations-Overhead vorgestellt. In Abbildung 6.2 sehen Sie eine schematische Darstellung eines RFC-Aufrufes, die die Aufrufzeit und die Remote-Ausführungszeit berücksichtigt.

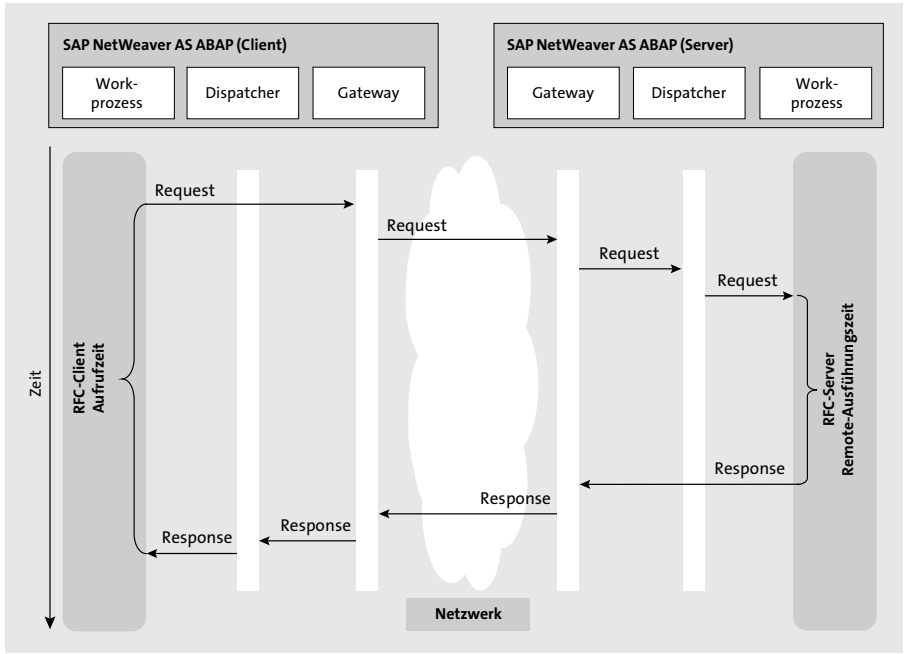

Abbildung 6.2 RFC-Kommunikation

Der Kommunikations-Overhead spielt im Zusammenhang mit der Anzahl der Aufrufe eine entscheidende Rolle für die Performance. Je größer der Anteil der Kommunikationszeit an der Ausführungszeit für einen Aufruf ist und je mehr Aufrufe in kurzer Zeit benötigt werden, desto größer ist der Anteil der Kommunikationszeit an der gesamten Antwortzeit. Dies gilt insbesondere für Netzwerkverbindungen, die an sich schon eine hohe Latenzzeit haben (WAN, Satellitenverbindung etc.). Eine Reduktion der Aufrufe ist in solchen Szenarien einer der wichtigsten Aspekte, die es zu berücksichtigen gilt.

Anzahl der RFC-Aufrufe im WAN

In Abschnitt 6.3.1, »Regel 1 und 2: Datenmenge klein halten«, haben Sie bereits gesehen, dass die Latenzzeit eines Aufrufes im WAN in unserem Beispiel (Walldorf – Shanghai) bei über 200 Millisekunden liegen kann. Betrachten wir dies nun an einem Beispiel mit einer unterschiedlichen Zahl

von Aufrufen. Im Beispiel werden 203 Datensätze vom Server gelesen: Im ersten Test ein Datensatz nach dem anderen und im zweiten Test alle Datensätze auf einmal. Die Ergebnisse sehen Sie in Tabelle 6.9. Die 203 Aufrufe benötigen fast 45 Sekunden, während ein einziger Aufruf nur ca. 223 Millisekunden benötigt. Dies liegt an der hohen Latenzzeit in diesem Szenario. Diese Latenzzeit fällt bei jedem Aufruf an.

Art des Aufrufes	Aufrufzeit [µs]	Remote Ausführungszeit [µs]
203 x 1 Datensatz im WAN	44.828.085 (203 x ~220)	162.400 (203 x ~800)
1 x 203 Datensätze im WAN	223.229	1.046

Tabelle 6.9 Anzahl der Aufrufe im WAN

Anzahl der RFC-Aufrufe im LAN

Die Anzahl der Aufrufe spielt aber auch im LAN eine Rolle. Betrachten Sie die Ergebnisse in Tabelle 6.10. Die Unterschiede sind hier wesentlich geringer, aber auch hier fallen bei jedem Aufruf (vermeidbare) Kosten an, die sich in der Laufzeit äußern.

Art des Aufrufes	Aufrufzeit [µs]	Remote Ausführungszeit [µs]
203 x 1 Datensatz im LAN	172.648	40.600 (203 x ~200)
1 x 203 Datensätze im WAN	2.240	214

Tabelle 6.10 Anzahl der Aufrufe im LAN

RFC-Schnittstelleninformationen für komplexe Schnittstellen effizient lesen

Manchmal werden die Schnittstelleninformationen (z. B. die Details der Übergabestrukturen) vom RFC-Baustein auf dem RFC-Server benötigt. Hierzu wird häufig die Funktion `DDIF_FIELDLIST_GET` benutzt. Diese muss bei komplexeren Schnittstellen (geschachtelte, tiefe Strukturen) aber mehrfach aufgerufen werden, um alle benötigen Details zu lesen. Der Funktionsbaustein `RFC_METADATA_GET` kann in solchen Fällen alle benötigten Informationen in einem Aufruf besorgen.

Schnittstellen massendatenfähig gestalten

Die Reduktion der Aufrufe von RFC-Bausteinen hat noch einen weiteren Vorteil, der auch bei normalen Funktionsbausteinen zum Tragen kommt. Nur wenn die Schnittstellen von (RFC-)Funktionsbausteinen massendaten-

fähig sind, kann die Verarbeitung innerhalb der Funktion auch effizient weitergeführt werden. Wenn Sie sich nochmals die goldenen Regeln der Datenbankperformance vor Augen führen, spielt auch dort die Anzahl der Aufrufe einer SQL-Anweisung eine Rolle (siehe Abschnitt 6.2.3, »Regel 3: Anzahl der Zugriffe klein halten«). Diese Regel kann aber nur dann ausgeführt werden, wenn mehrere Datensätze in der Schnittstelle übergeben wurden. Wenn immer nur ein Datensatz übergeben wird, kann im Funktionsbaustein auch immer nur ein Datensatz von der Datenbank gelesen werden.

Interne Tabellen und Tabellenpuffer

Beim Zugriff auf interne Tabellen bzw. den Tabellenpuffer ist die Anzahl der Zugriffe wenig relevant für die Performance, da diese Zugriffe im kleinen ein- bis zweistelligen Mikrosekundenbereich liegen (sofern eine effiziente Suche benutzt wurde, siehe folgenden Abschnitt 6.3.3). Hier würden Sie also erst bei sehr, sehr vielen Zugriffen kleine Laufzeitunterschiede feststellen.

6.3.3 Regel 4: Zu durchsuchende Datenmenge klein halten

Bei der dritten goldenen Regel geht es bei der ABAP-Schicht hauptsächlich um den Zugriff auf interne Tabellen und den Tabellenpuffer. Wenn diese Regeln verletzt werden, erkennen Sie dies in der Transaktion ST12 an hohen Zeiten pro Ausführung einer `READ`-Anweisung (siehe Abschnitt 5.2.1, »ABAP-Trace auswerten«). Hohe Zeiten für `LOOP`- oder `SELECT`-Anweisungen für gepufferte Tabellen können auch auf eine Verletzung der Regel hindeuten.

Arten von internen Tabellen

Wir unterscheiden bei den internen Tabellen die folgenden drei Typen:

- **Standardtabelle**
  Die Standardtabelle erlaubt keine eindeutigen Datensätze (Schlüssel sind immer uneindeutig) und bietet keine automatisch optimierten Zugriffe. Eindeutigkeit und optimierte Zugriffe müssen im ABAP-Programm sichergestellt werden.
- **Sortierte Tabelle**
  Die sortierte Tabelle kann eindeutige Schlüssel (*Unique Keys*) verwenden. Dies ist aber kein Muss. Doppelte Schlüssel (*Non-Unique Keys*) sind genauso möglich. Die sortierte Tabelle wird vom ABAP Kernel immer sortiert gehalten und erlaubt daher automatisch optimierte Zugriffe über eine binäre Suche, wenn die Voraussetzungen dafür erfüllt sind (darauf gehen wir im Folgenden näher ein).
- **Hash-Tabelle**
  Die Hash-Tabelle kann nur eindeutige Schlüssel verwenden. Beim optimierten Zugriff über einen sogenannten *Hash-Lookup* wird genau ein (oder kein) Datensatz adressiert. Dieser Zugriff wird automatisch durchgeführt, wenn die Voraussetzungen dafür erfüllt sind (darauf gehen wir im Folgenden ein).

Sekundärschlüssel

Zur Zugriffsoptimierung gibt es *Sekundärschüssel*. Diese können für jeden Tabellentyp definiert werden (maximal 15 Stück pro interne Tabelle). Es stehen sortierte oder gehashte Sekundärschlüssel zur Verfügung. Die Eigenschaften der Sekundärschlüssel entsprechen denen der Tabellentypen.

Zugriffspfade

Bei den internen Tabellen gibt es die folgenden drei Zugriffspfade:

- Hash-Lookup
- binäre Suche
- Scan

Darüber hinaus gibt es noch den direkten Indexzugriff über die Zeilennummer. Dieser ist aber in Geschäftsapplikationen meist von untergeordneter Bedeutung und kommt nur selten vor. Er kann nur bei Standard- oder sortieren Tabellen (bzw. sortierten Sekundärschlüsseln) verwendet werden. Performanceprobleme kann es bei diesem Zugriff nicht geben, da keine Suche stattfindet, sondern eine Zeile direkt über die Zeilennummer adressiert wird.

Zugriffe auf interne Tabellen werden meist in Schleifen, also sehr häufig ausgeführt. Bei diesen Zugriffen auf interne Tabellen ist es für die Performance sehr wichtig, dass entweder die binäre Suche oder der Hash-Lookup benutzt werden. Im Folgenden erklären wir Ihnen, wie diese Zugriffe funktionieren und unter welchen Umständen sie benutzt werden können.

Hash-Lookup

Beim Hash-Lookup werden alle Teile des Primärschlüssels verwendet, um einen Hash-Wert zu erzeugen. Dieser Hash-Wert kann dann genau einem (oder keinem, wenn es den gesuchten Wert nicht gibt) Eintrag in der internen Tabelle zugeordnet werden. Der Hash-Wert kann nur dann berechnet werden, wenn alle Bestandteile des Schlüssels bekannt sind. Die Laufzeit für den Hash-Lookup ist konstant, also unabhängig von der Größe der internen Tabelle.

Bestandteile des Primärschlüssels [zB]

Sie können nur dann mit einer Telefonnummer den Namen einer Person ermitteln, wenn sowohl das Land als auch die vollständige Vorwahl und Telefonnummer vorliegen.

Einzelsatzzugriff

Bei der `READ`-Anweisung (dem Einzelsatzzugriff) wird der Hash-Lookup verwendet, wenn die folgenden Punkte erfüllt sind:

- Es handelt sich um eine Hash-Tabelle, oder es existiert ein Sekundärschlüssel vom Typ Hash.
- Alle Schlüsselfelder des Primärschlüssels (bei der Hash-Tabelle) oder des Sekundärschlüssels (beim Sekundärschlüssel vom Typ Hash) sind mit = spezifiziert.

Mengenzugriff

Bei der Anweisung `LOOP WHERE` (Mengenzugriff) kann ein Hash-Lookup ebenfalls verwendet werden. Dazu müssen dieselben Bedingungen zutreffen wie bei der `READ`-Anweisung. Zusätzlich müssen die Schlüsselfelder mit `AND` verknüpft werden (die Anweisung `LOOP WHERE` filtert dann auch nur auf einen oder keinen Datensatz). Wenn diese Bedingungen nicht zutreffen, wenn also z. B. nicht alle Schlüsselfelder beim Zugriff auf eine Hash-Tabelle angegeben sind, wird ein Scan ausgeführt.

Binäre Suche

Bei der binären Suche wird die interne Tabelle im sogenannten *Intervall-Halbierungsverfahren* durchsucht. Voraussetzung für die binäre Suche ist, dass die interne Tabelle nach dem bei der Suche verwendeten Suchschlüssel sortiert ist. Ist dies nicht der Fall, kann das Ergebnis fehlerhaft sein. Die Suche beginnt dann in der Mitte der internen Tabelle. Je nachdem, ob der gesuchte Wert kleiner oder größer als der Wert in der Mitte ist, wird die Suche dann in der oberen oder unteren Hälfte fortgesetzt. In der jeweiligen Hälfte wird dann wieder in der Mitte gesucht usw. So wird nach wenigen Schritten der gesuchte Eintrag (so vorhanden) gefunden.

Die Laufzeit ist logarithmisch von der Größe der Tabelle abhängig. Eine Verdopplung der Einträge in einer internen Tabelle hat nur einen weiteren Suchschritt zur Folge, d. h., es besteht nur eine sehr schwache Abhängigkeit von der Größe der internen Tabelle.

Einzelsatzzugriff

Bei der `READ`-Anweisung wird die binäre Suche verwendet, wenn die folgenden Voraussetzungen erfüllt sind:

- Es handelt sich um eine sortierte Tabelle, oder es existiert ein sortierter Sekundärschlüssel.
- Es wurde mit dem Zusatz `BINARY SEARCH` gesucht (bei Standardtabellen). Wenn der Zusatz `BINARY SEARCH` benutzt wird, muss die in der `READ`-Anweisung angegebene Reihenfolge der Schlüsselfelder der Sortierreihenfolge der internen Tabelle entsprechen, sonst kann es zu falschen Ergebnissen kommen.
- Beim Primärschlüssel (bei der sortieren Tabelle) oder dem Sekundärschlüssel (beim sortieren Sekundärschlüssel) sind alle oder ein Teil der Felder linksbündig (d. h. ohne Lücken im Schlüssel) mit = spezifiziert.

Duplikate bei der binären Suche mit der READ-Anweisung [!]

Eine Besonderheit der binären Suche mit der `READ`-Anweisung in ABAP ist, dass bei einem nicht eindeutigen Schlüssel und mehrfach vorkommenden Werten immer der erste Wert in der Tabelle zurückgeliefert wird. Wenn die binäre Suche den gesuchten Wert gefunden hat, wird in der Tabelle so lange der vorangehende Wert angeschaut, bis er Wert nicht mehr zum gesuchten Schlüssel passt, also der erste Datensatz gefunden wurde. Bei sehr vielen Duplikaten (wenn im Extremfall alle Datensätze denselben Wert haben) kommt dies einem Scan der halben internen Tabelle gleich.

Sortiertes Einfügen von Daten [+]

Dieses Verhalten können Sie sich beim Aufbau von internen Tabellen zunutze machen. Da der Return-Code `sy-tabix` immer auf den ersten Datensatz mit dem gesuchten Schlüssel zeigt bzw. auf die Position, an der der Datensatz stehen sollte, wenn dieser nicht vorhanden ist, können Sie diese Position zum Einfügen eines neuen Datensatzes mit diesem Schlüssel nutzen und die interne Tabelle so sortiert aufbauen, ohne eine anschließende `SORT`-Anweisung zu benötigen.

Mengenzugriff

Bei der Anweisung `LOOP WHERE` (Mengenzugriff) kann ebenfalls eine binäre Suche verwendet werden. Dazu müssen dieselben Bedingungen zutreffen wie bei der `READ`-Anweisung. Zusätzlich müssen die Schlüsselfelder mit `AND` verknüpft werden. Bei vielen Duplikaten kann die Laufzeit beeinträchtigt sein und im Extremfall so lange dauern wie beim Scan.

Scan

Beim Scan wird die interne Tabelle vom ersten Datensatz an satzweise durchsucht. Beim ersten Treffer ist die Suche beendet. Wenn in einer internen Tabelle also alle vorkommenden Werte gesucht werden, wird bei jedem Zugriff im Durchschnitt die halbe Tabelle gelesen. Bei diesem Zugriff handelt es sich daher um die zeitaufwendigste Suche. Dieser Scan wird immer dann verwendet, wenn kein Hash-Lookup oder keine binäre Suche möglich ist.

Laufzeit für die READ-Anweisung

Betrachten wir ein Beispiel, in dem wir 100-mal auf die interne Tabelle `IT_SBOOK` (mit 2.608.652 Einträgen) zugreifen, um Flugbuchungen zu suchen. Die in Transaktion ST12 für diesen Zugriff in unserem System ausgegebenen Zeiten sind in Tabelle 6.11 aufgeführt.

Im zweiten Beispiel in Tabelle 6.12 wurde die interne Tabelle `IT_SFLIGHT` (mit 4.910 Einträgen) in einer `LOOP`-Anweisung über die interne Tabelle `IT_SBOOK` (2.608.652 Einträge) mit der `READ`-Anweisung gelesen. Wir vergleichen in den beiden Tabellen jeweils die Ergebnisse für den Hash-Lookup, die binäre Suche und den Scan.

	Hash-Lookup	Binäre Suche	Scan
100 READ-Anweisungen [ms]	0,11 ms	0,13 ms	92 ms
pro Aufruf [µs]	~1 µs	~1 µs	~920 µs

Tabelle 6.11 Laufzeiten der READ-Anweisung

	Hash-Lookup	Binäre Suche	Scan
2.608.652 READ-Anweisungen [s]	1,6 s	1,9 s	170 s
pro Aufruf [µs]	~1 µs	~1 µs	~65 µs

Tabelle 6.12 Laufzeiten der LOOP-Anweisung

Bei vielen Aufrufen und/oder großen internen Tabellen ist eine optimierte Suche also von großer Bedeutung, wenn es um die Optimierung der Laufzeiten geht.

Weitere zeitaufwendige ABAP-Operationen

Es gibt noch weitere Anweisungen, die zu teuren Suchen im ABAP-Code führen können:

- Für die Anweisungen `MODIFY` und die `DELETE` gelten dieselben Regeln wie für die Anweisungen `READ` und `LOOP`.
- Die Anweisung `SORT` führt keine Suche aus, ist aber eine Voraussetzung für die binäre Suche, wenn die Daten nicht schon sortiert vorliegen (z. B. über den Zusatz `ORDER BY` bei der `SELECT`-Anweisung). Da die `SORT`-Anweisung auch eine teure Anweisung ist, sollte diese so sparsam wie möglich eingesetzt werden.
- Die Anweisung `COLLECT`, die Daten verdichtet, benutzt intern ebenfalls eine Suche. Bei sortierten bzw. Hash-Tabellen (bzw. Standardtabellen mit sortierten oder Hash-Sekundärschlüsseln) wird die optimierte Suche benutzt.

  Bei Standard-Tabellen ohne Sekundärschlüssel wird eine temporäre Hash-Verwaltung aufgebaut. Solange die interne Tabelle nicht mit anderen ändernden Anweisungen bearbeitet wird, kann diese Hash-Verwaltung für eine optimierte Suche benutzt werden. Falls die Tabelle anderweitig geändert wurde ist, die Hash-Verwaltung zerstört, und es muss ein zeitaufwendiger Scan benutzt werden.
- Die Anweisungen `FIND IN TABLE` und `REPLACE IN TABLE` suchen nach bestimmten Zeichenfolgen innerhalb von Feldern der Tabelle. Dabei wird die interne Tabelle gescannt. Benutzen Sie diese Anweisungen daher, so selten es geht.

Tabellenpuffer

Beim Zugriff auf den Tabellenpuffer gelten die gleichen Regeln wie beim Zugriff auf die Datenbank (siehe Abschnitt 6.2.4, »Regel 4: Zu durchsuchende Datenmenge klein halten«). Vor dem Release SAP NetWeaver 7.40 stand nur der Primärschlüssel einer Datenbanktabelle für eine optimierte Suche im Tabellenpuffer zur Verfügung. Daher kam es in solchen Systemen oft zu teuren Scans im Tabellenpuffer, wenn eine SQL-Anweisung so formuliert war, dass der Primärschlüssel nicht (effizient) benutzt werden konnte.

Sekundärschlüssel im Tabellenpuffer

Seit SAP NetWeaver 7.40 basiert der Tabellenpuffer auf internen Tabellen und bildet anhand der Sekundärschlüssel auch die Sekundärschlüssel der Datenbanktabellen ab. Nun können auch SQL-Anweisungen, die einen effizienten Zugriff auf einen Sekundärschlüssel verwenden, optimal im Tabellenpuffer umgesetzt werden. Achten Sie beim Zugriff auf den Tabellenpuffer also einfach auf die Regeln, die bereits in Abschnitt 6.2, »Die goldenen Regeln angewandt auf die Datenbankperformance«, besprochen wurden.

6.3.4 Regel 5: (Zentrale) Ressourcen entlasten

Auch auf der Applikationsebene im ABAP-Programm gibt es zentrale Ressourcen, die Sie entlasten können. Es geht dabei um das Netzwerk und den Enqueue-Service. Darüber hinaus können Sie durch das Puffern von Ergebnissen von Funktionsbausteinen die Central Processing Units (CPUs) des ABAP-Applikationsservers entlasten.

Netzwerk entlasten

Das Netzwerk können Sie entlasten, falls die benötigten Daten im RFC-Client und im RFC-Server zur Verfügung stehen. Dann ist es in der Regel günstiger, die Daten im RFC-Server (wenn diese dort benötigt werden) von der Datenbank zu lesen, anstatt diese im RFC-Client zu lesen und an den RFC-Server zu übertragen. Die reduzierte Datenübertragung entlastet das Netzwerk.

Enqueue-Service entlasten

Der Enqueue-Service verwaltet die Sperren auf der Ebene des ABAP-Applikationsservers. Sperren auf der Applikationsserverebene werden explizit vom Programmierer gesetzt (mit der Anweisung `CALL FUNCTION 'ENQUEUE_...'`). Ist das zu sperrende Objekt bereits gesperrt, wird eine sofortige Antwort in Form eines entsprechenden Rückgabewertes (`FOREIGN_LOCK`) gegeben. Die Anwendung muss dann entsprechend darauf reagieren, z. B. mit einem Abbruch und einer Fehlermeldung oder mit dem Versuch, die Sperre erneut zu bekommen.

Wenn Sie Sperren erneut anfordern, weil der vorhergehende Versuch fehlgeschlagen ist, achten Sie darauf, dies nach einer Wartezeit zu tun. Sie können dazu entweder den `_WAIT`-Parameter des Sperrbausteins benutzen oder mit der ABAP-Anweisung `WAIT UP TO n SECONDS` selbst eine Wartezeit implementieren. So schützen Sie den Enqueue-Service vor sich schnell wiederholenden Anfragen derselben Sperre.

Array Interface

Eine weitere Möglichkeit, um den Enqueue-Service zu entlasten, ist das sogenannte *Array Interface*. Hierbei übertragen Sie nicht jede Sperranfrage einzeln, sondern sammeln die Anfragen zunächst in einem Paket und übertragen dieses dann en bloc an den Enqueue-Service. Mithilfe des `COLLECT`-Parameters können Sie die Sammlung in einem Paket veranlassen, das dann mit dem Funktionsbaustein `FLUSH_ENQUEUE` an den Enqueue-Service übertragen wird.

Dieser Funktionsbaustein kann dazu genutzt werden, Sperren nach dem Prinzip »alles oder nichts« zu realisieren. Kann nur eine Sperranfrage im Container nicht erfüllt werden, schlägt das ganze Paket fehl. Seien Sie deswegen vorsichtig mit Sperren, die logisch nicht voneinander abhängig sind, und wählen Sie nicht zu große Pakete. Wenn Sie große Pakete mit Sperren mehrfach anfragen müssen, um die Sperren zu erhalten, belasten die Fehlschläge den Enqueue-Service ebenfalls. Mit diesen Vorgehensweisen sparen Sie Kommunikationsschritte und Anfragen an den Enqueue-Service ein.

[!]

Sperrmodus X

Beim Sperrmodus X (erweiterte Schreibsperre) sollten Sperren nicht in den Container geschrieben werden, wenn sich sehr viele Sperren auf dieselbe Sperrtabelle beziehen und die Wahrscheinlichkeit für Sperrkollisionen hoch ist. In diesem Fall kommt es nämlich im Vergleich zum direkten Abschicken der Sperren zu erheblichen Performanceverlusten.

Ergebnisse von Funktionsbausteinen puffern

Wenn Sie identische `SELECT`-Zugriffe auf die Datenbank (siehe Abschnitt 6.2.5, »Regel 5: (Zentrale) Ressourcen entlasten«) entdeckt haben, ist dies häufig nur die Spitze des Eisberges. Bevor Sie einen Puffer auf der Ebene des Datenbankzugriffs implementieren, prüfen Sie daher, ob nicht schon auf einer darüberliegenden Ebene Funktionsbausteine mit denselben Parametern gerufen wurden. Eine Pufferung auf der Ebene der Funktionsaufrufe kann viel mehr bringen, weil sie dann nicht nur die identischen Datenbankzugriffe vermeiden, sondern auch die gesamte Verarbeitung der Daten nicht wiederholt für dieselben Daten durchführen. Dies gilt vor allem dann, wenn es sich um Funktionsbausteine handelt, die per RFC gerufen werden. Hier sollten keine Daten doppelt übertragen werden.