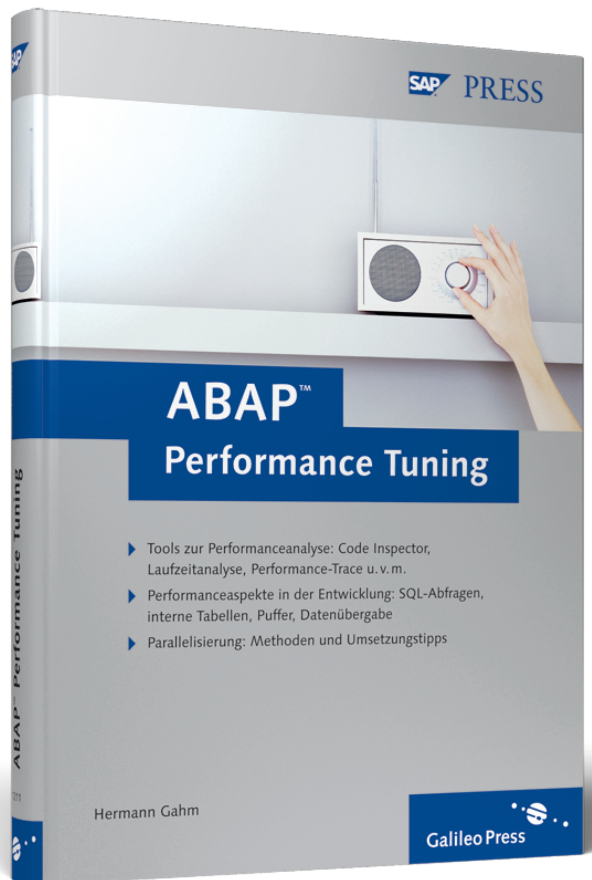
Hermann Gahm

ABAP™ Performance Tuning



Bonn • Boston

Auf einen Blick

1 Einführung ... 17

2 SAP-Systemarchitektur für ABAP-Entwickler ... 21

3 Werkzeuge zur Performanceanalyse ... 29

4 Parallelisierung ... 133

5 Datenverarbeitung per SQL ... 155

6 Pufferung von Daten ... 241

7 Verarbeitung interner Tabellen ... 275

8 Kommunikation mit anderen Systemen ... 309

9 Spezielle Themen ... 317

10 Ausblick ... 325

A Ausführungspläne der verschiedenen Datenbanken ... 343

B Der Autor ... 363

Inhalt

Geleitwort 13
Vorwort und Danksagung 15

1 Einführung 17

1.1 Tuning-Methoden 17
1.2 Aufbau des Buches 19
1.3 Hinweise zur Verwendung des Buches 20

2 SAP-Systemarchitektur für ABAP-Entwickler 21

2.1 Die SAP-Systemarchitektur 21
2.1.1 Die Dreischichtenarchitektur 22
2.1.2 Verteilung der drei Schichten 23
2.2 Performanceaspekte der Architektur 25
2.2.1 Frontend 26
2.2.2 Applikationsschicht 26
2.2.3 Datenbank 27
2.2.4 Zusammenfassung 27

3 Werkzeuge zur Performanceanalyse 29

3.1 Übersicht über die Werkzeuge 30
3.2 Einsatzzeitpunkte der Werkzeuge 32
3.3 Die Analyse und die Werkzeuge im Detail 34
3.3.1 SAP Code Inspector (SCI) 35
3.3.2 Selektivitätsanalyse (DB05) 41
3.3.3 Prozessanalyse (SM50/SM66) – Status eines Programms 45
3.3.4 Debugger – Speicheranalyse 48
3.3.5 Memory Inspector (S_MEMORY_INSPECTOR) 50
3.3.6 ST10 – Table Call Statistics 52
3.3.7 Performance-Trace – Allgemeines (ST05) 55
3.3.8 Performance-Trace – SQL-Trace (ST05) 58
3.3.9 Performance-Trace – RFC-Trace (ST05) 72
3.3.10 Performance-Trace – Enqueue-Trace (ST05) 74
3.3.11 Performance-Trace – Tabellenpuffer-Trace (ST05) 76
3.3.12 ABAP-Trace (SE30) 79
3.3.13 Single Transaction Analysis (ST12) 93

3.3.14 E2E-Trace ... 105
3.3.15 Einzelsatzstatistik (STAD) ... 114
3.3.16 Dump-Analyse (ST22) ... 125
3.4 Tipps zur Performanceanalyse ... 129
3.4.1 Konsistenzchecks ... 129
3.4.2 Zeitbasierte Analyse ... 129
3.4.3 Vermeidung ... 129
3.4.4 Optimierung ... 130
3.4.5 Laufzeitverhalten bei Massendaten ... 130
3.5 Zusammenfassung ... 130

4 Parallelisierung ... 133

4.1 Paketierung ... 133
4.2 Parallelisierung ... 135
4.2.1 Motivation ... 136
4.2.2 Herausforderungen und Lösungsansätze für parallelisierte Programme ... 137
4.2.3 Techniken zur Parallelisierung ... 147
4.2.4 Zusammenfassung ... 152

5 Datenverarbeitung per SQL ... 155

5.1 Die Architektur einer Datenbank ... 155
5.2 Ausführung von SQL ... 160
5.2.1 Ausführung im SAP NetWeaver AS ABAP ... 160
5.2.2 Ausführung in der Datenbank ... 162
5.3 Effizientes SQL: Grundsätzliches ... 164
5.4 Zugriffsstrategien ... 164
5.4.1 Logische Strukturen ... 165
5.4.2 Indizes als Suchhilfe ... 167
5.4.3 Operatoren ... 177
5.4.4 Entscheidung für einen Zugriffspfad ... 179
5.4.5 Analyse und Optimierung in ABAP ... 181
5.4.6 Zusammenfassung ... 196
5.5 Ergebnismenge ... 197
5.5.1 Reduktion der Spalten ... 200
5.5.2 Reduktion der Zeilen ... 203
5.5.3 Eine bestimmte Anzahl von Zeilen lesen ... 205
5.5.4 Aggregate bilden ... 207
5.5.5 Existenzchecks ... 209

5.5.6 Updates ... 210
5.5.7 Zusammenfassung ... 211
5.6 Indexdesign ... 211
5.6.1 Lesende oder schreibende Verarbeitung? ... 214
5.6.2 Wie wird auf die Daten zugegriffen? ... 217
5.6.3 Zusammenfassung ... 219
5.7 Ausführungshäufigkeit ... 219
5.7.1 Array Interfaces ... 220
5.7.2 SELECTS in Schleifen und geschachtelte SELECTS ... 222
5.7.3 View ... 224
5.7.4 Join ... 226
5.7.5 FOR ALL ENTRIES ... 227
5.7.6 Änderungen in Schleifen und COMMIT ... 230
5.8 Verwendetes API ... 232
5.8.1 Open SQL statisch ... 232
5.8.2 Open SQL dynamisch ... 233
5.8.3 Native SQL statisch ... 233
5.8.4 Native SQL dynamisch ... 233
5.8.5 Zusammenfassung ... 233
5.9 Spezialfälle und Ausnahmen ... 234
5.9.1 Sortieren ... 234
5.9.2 Pool- und Cluster-Tabellen ... 235
5.9.3 Hints und Anpassung von Statistiken ... 237
5.10 Zusammenfassung ... 240

6 Pufferung von Daten ... 241

6.1 SAP-Speicherarchitektur aus Sicht des Entwicklers ... 241
6.1.1 Benutzerbezogener Speicher ... 243
6.1.2 Benutzerübergreifender Speicher ... 244
6.2 Benutzerbezogene Pufferungsarten ... 245
6.2.1 Pufferung im internen Modus ... 245
6.2.2 Pufferung über interne Modi hinweg ... 249
6.2.3 Pufferung über externe Modi hinweg ... 250
6.2.4 Zusammenfassung ... 250
6.3 Benutzerübergreifende Pufferungsarten ... 251
6.3.1 Pufferung im Shared Buffer ... 251
6.3.2 Pufferung im Shared Memory ... 252
6.3.3 Pufferung über Shared Objects ... 253
6.3.4 Zusammenfassung ... 255
6.4 SAP-Tabellenpufferung ... 256
6.4.1 Architektur und Übersicht ... 257

6.4.2 Welche Tabellen können gepuffert werden? ... 263
6.4.3 Performanceaspekte der Tabellenpufferung ... 264
6.4.4 Analysemöglichkeiten ... 272
6.5 Zusammenfassung ... 273

7 Verarbeitung interner Tabellen ... 275

7.1 Überblick über interne Tabellen ... 276
7.2 Organisation im Hauptspeicher ... 277
7.3 Die Tabellentypen ... 281
7.4 Performanceaspekte ... 287
7.4.1 Füllen ... 287
7.4.2 Lesen ... 291
7.4.3 Ändern ... 296
7.4.4 Löschen ... 297
7.4.5 Verdichten ... 298
7.4.6 Sortieren ... 299
7.4.7 Kopierkostenreduzierter bzw. -freier Zugriff ... 300
7.4.8 Sekundärindizes ... 302
7.4.9 Kopieren ... 303
7.4.10 Geschachtelte Schleifen und nicht-lineares Laufzeitverhalten ... 305
7.4.11 Zusammenfassung ... 308

8 Kommunikation mit anderen Systemen ... 309

8.1 RFC-Kommunikation zwischen ABAP-Systemen ... 310
8.1.1 Synchroner RFC ... 310
8.1.2 Asynchroner RFC ... 311
8.2 Performanceaspekte bei der RFC-Kommunikation ... 312
8.3 Zusammenfassung ... 316

9 Spezielle Themen ... 317

9.1 Lokale Verbuchung ... 317
9.1.1 Asynchrone Verbuchung ... 317
9.1.2 Lokale Verbuchung ... 318
9.2 Parameterübergaben ... 320
9.3 Typkonvertierungen ... 321
9.4 Indextabellen ... 321
9.5 Frontendressourcen schonen ... 322
9.6 Enqueue- und Message-Service schonen ... 323

10 Ausblick ... 325

10.1 Wichtige Änderungen an den Werkzeugen zur Performanceanalyse ... 325
10.1.1 Performance-Trace (ST05) ... 325
10.1.2 ABAP-Trace (SAT) ... 330
10.2 Wichtige Änderungen bei internen Tabellen (Sekundärschlüssel) ... 336
10.2.1 Definition ... 337
10.2.2 Verwaltungskosten und Lazy Index Update ... 338
10.2.3 Lesezugriffe ... 338
10.2.4 Active Key Protection ... 339
10.2.5 Delayed Index Update bei inkrementellen Schlüsseländerungen ... 340
10.2.6 Zusammenfassung ... 341

Anhang ... 343

A Ausführungspläne der verschiedenen Datenbanken ... 343
B Der Autor ... 363

Index ... 365

Eine häufige Ursache lang laufender ABAP-Programme sind ineffiziente Zugriffe auf interne Tabellen. Dies trifft besonders bei der Verarbeitung großer Datenmengen zu. Dieses Kapitel beschreibt die für ABAP-Entwickler wichtigsten Aspekte bei der Verarbeitung interner Tabellen.

7 Verarbeitung interner Tabellen

Interne Tabellen gehören zu den komplexesten Datenobjekten, die es im ABAP-Umfeld gibt. Mit internen Tabellen ist es möglich, dynamische Datenmengen im Hauptspeicher abzulegen. Interne Tabellen sind vergleichbar mit Arrays und entlasten aufgrund ihrer Dynamik den Programmierer vom Aufwand der programmgesteuerten Speicherverwaltung. Die Daten in internen Tabellen werden zeilenweise verwaltet, wobei jede Zeile die gleiche Struktur hat.

Meistens werden interne Tabellen zur Pufferung oder Aufbereitung von Inhalten aus Datenbanktabellen verwendet. Die Art des Zugriffs auf interne Tabellen spielt, wie auch bei den Datenbanktabellen, eine große Rolle für die Performance. Die Praxis zeigt, dass über das Tuning der internen Tabellen ähnlich große Effekte wie beim Tuning der Datenbankzugriffe möglich sind. Die negativen Effekte ineffizienter Zugriffe auf interne Tabellen für das Gesamtsystem lassen sich durch das Hinzufügen weiterer CPUs oder Application Server aber leichter ausgleichen als ineffiziente Datenbankzugriffe. Ineffiziente Datenbankzugriffe belasten die Datenbank als zentrale Ressource, während ineffiziente Zugriffe auf interne Tabellen die besser skalierbare Applikationsschicht (siehe Kapitel 2) belasten.

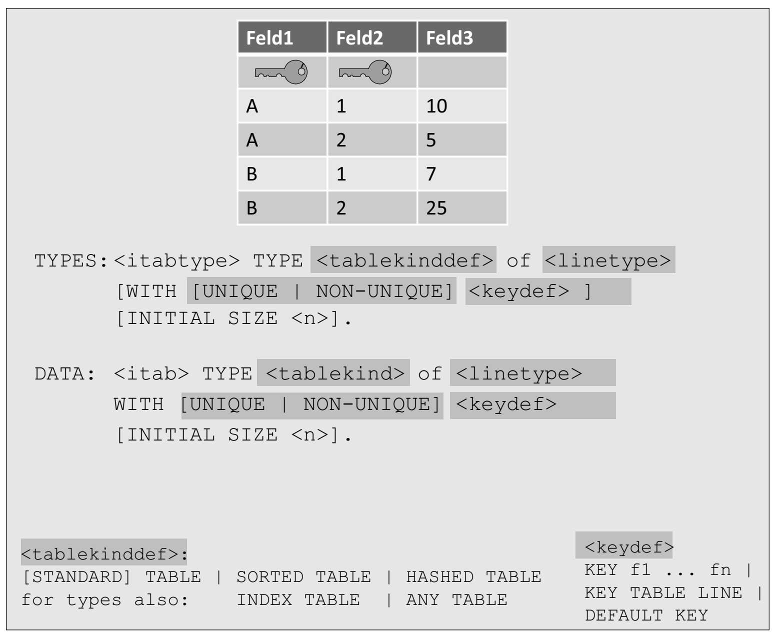
Die folgenden Abschnitte geben zunächst einen Überblick über interne Tabellen im Allgemeinen. Danach sehen wir uns an, wie interne Tabellen im Hauptspeicher organisiert werden. Anschließend betrachten wir die verschiedenen Arten interner Tabellen. Es folgt dann der wesentliche Teil des Kapitels, die Performanceaspekte bei der Verarbeitung interner Tabellen. Hier werden typische problematische Beispiele und Lösungsmöglichkeiten vorgestellt.

7.1 Überblick über interne Tabellen

Interne Tabellen werden durch vier Eigenschaften vollständig spezifiziert:

1. **Tabellentyp**
   Die Zugriffsart auf den Tabellentyp bestimmt, wie ABAP auf einzelne Tabellenzeilen zugreift. Dieses Thema wird ausführlich in Abschnitt 7.3 besprochen.
2. **Zeilentyp**
   Der Zeilentyp einer internen Tabelle kann ein beliebiger ABAP-Datentyp sein.
3. **Eindeutigkeit des Schlüssels**
   Der Schlüssel kann als eindeutig (unique) oder nicht eindeutig (non-unique) festgelegt werden. Bei eindeutigen Schlüsseln gibt es keine mehrfachen Einträge (bezüglich des Schlüssels) in internen Tabellen. Die Eindeutigkeit richtet sich nach dem Tabellentyp. Standardtabellen erlauben nur Non-unique-Schlüssel und Hash-Tabellen nur Unique-Schlüssel.
4. **Schlüsselkomponenten (unter Berücksichtigung der Reihenfolge)**
   Die Schlüsselkomponenten und ihre Reihenfolge legen die Kriterien fest, anhand derer die Identifikation von Tabellenzeilen erfolgt.

In Abbildung 7.1 wird dies nochmals syntaktisch dargestellt.



Feld1	Feld2	Feld3
A	1	10
A	2	5
B	1	7
B	2	25

Abbildung 7.1 Interne Tabellen – Deklaration

Für die Performance ist hauptsächlich die Kombination aus Zugriffsart und Tabellentyp relevant. Die verschiedenen Zugriffsarten und Tabellentypen werden in Abschnitt 7.3 behandelt. Bevor wir uns die Tabellentypen im Detail ansehen, betrachten wir zunächst, wie die internen Tabellen im Hauptspeicher organisiert sind.

7.2 Organisation im Hauptspeicher

Im Hauptspeicher werden die internen Tabellen, wie auch Datenbanktabellen, in Blöcken bzw. Seiten organisiert. Im Zusammenhang mit den internen Tabellen sprechen wir im Folgenden von *Seiten*.

Wenn eine interne Tabelle in einem ABAP-Programm deklariert wird, wird im Hauptspeicher zunächst nur eine Referenz (Table Reference) angelegt. Erst wenn Einträge in die Tabelle geschrieben werden, werden ein Tabellenkopf (Table Header) und ein Tabellenkörper (Table Body) angelegt. Abbildung 7.2 zeigt eine schematische Darstellung der Organisation im Hauptspeicher.

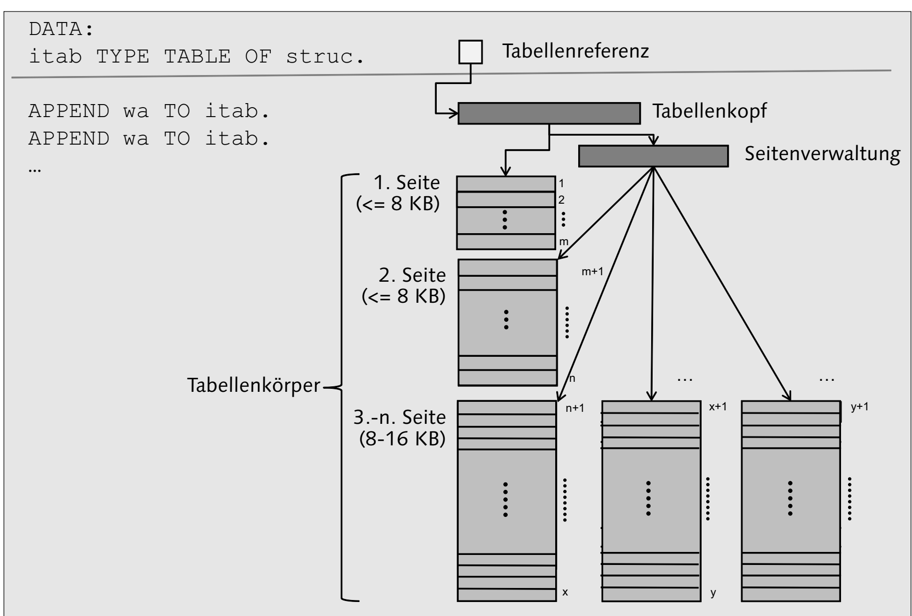


Abbildung 7.2 Schematische Darstellung der Organisation interner Tabellen im Hauptspeicher

Der Tabellenkopf hat eine Referenz auf die erste Seite des Tabellenkörpers und eine weitere auf die Seitenverwaltung. In der Seitenverwaltung werden die Adressen der Seiten im Hauptspeicher verwaltet.

Die Tabellenreferenz belegt zurzeit 8 Byte an Speicherplatz. Der Tabellenheader belegt, abhängig von der Plattform, ca. 100 Byte an Speicherplatz. Der benötigte Platz für die Seitenverwaltung hängt von der Anzahl der Seiten ab.

Der Tabellenkörper besteht aus Seiten, die die Tabellenzeilen aufnehmen können. Die ersten beiden Seiten sind, in Abhängigkeit von der Zeilenlänge und weiteren Faktoren, in der Regel kleiner als die Seiten 3–n (wenn die Zeilenlängen nicht so groß sind, dass gleich zu Beginn die maximale Seitengröße erreicht wird).

Ab der dritten Seite werden die Seiten mit der maximalen Seitengröße angelegt, die zwischen 8–16 KB liegt. Dies ist von der Länge einer Zeile abhängig. Anders als bei Datenbanktabellen erfolgt der Zugriff nicht seiten-, sondern zeilenweise. Beim Zugriff auf eine Zeile einer internen Tabelle wird also immer nur eine Zeile gelesen. Vergleichbar mit den Datenbanktabellen ist aber wieder der Aufwand für das Suchen von Tabelleneinträgen (bzw. Datensätzen). Auch für die internen Tabellen gibt es hierzu Unterstützung durch Index- oder Hash-Verwaltung. Über interne Tabellen werden Sie im Abschnitt 7.3 bei den Tabellentypen mehr erfahren, da sie direkt mit ihnen zusammenhängen.

Im Tabellenkopf befinden sich die wichtigsten Informationen zu einer internen Tabelle. So lässt sich zum Beispiel die Anzahl der Zeilen mit `DESCRIBE TABLE <itab> LINES <lines>` oder der eingebauten Funktion `LINES( itab )` sehr schnell aus dem Tabellenkopf abfragen.

Da es bei sehr kleinen internen Tabellen mit wenig Zeilen, bedingt durch den Speicherverbrauch der automatisch kalkulierten ersten Seite, zu einem Verschnitt kommen kann, gibt es den Zusatz `INITIAL SIZE` bei der Deklaration interner Tabellen. Über diesen kann ein Hinweis für die Größe der ersten Seite gegeben werden, so dass eine kleinere Speicherallokation als im Standardfall entsteht.

Wenn deutlich mehr Zeilen, als ursprünglich für `INITIAL SIZE` angegeben, benötigt werden, wird allerdings schneller die dritte Seite in der maximalen Seitengröße angelegt. Wenn z. B. 4 für `INITIAL SIZE` angegeben wurde, kann schon ab der 13. Zeile die dritte Seite benötigt werden, wenn die zweite Seite doppelt so groß wie die erste Seite ist. Das bedeutet, dass schon für relativ wenig Zeilen, z. B. 13, relativ viel Speicher (drei Seiten, dritte Seite 8–16 KB groß) benötigt wird, während bei einer höheren Angabe für `INITIAL SIZE` (z. B. 14) eine Seite ausgereicht hätte. Für kleine Tabellen ist also wichtig, dass `INITIAL SIZE` nicht zu klein gewählt wird. Es sollte so gewählt werden, dass für die meisten Anwendungsfälle genügend Platz in der ersten (bzw. ersten und zweiten) Seite zur Verfügung steht.

`INITIAL SIZE` sollte immer dann angegeben werden, wenn nur wenige Zeilen benötigt werden und die interne Tabelle häufig existiert. Dies ist bei geschachtelten Tabellen, also wenn eine interne Tabelle Teil einer Zeile einer anderen internen Tabelle ist, für die innere Tabelle potenziell der Fall. Es kann aber auch bei Attributen einer Klasse vorkommen, wenn es sehr viele Instanzen dieser Klasse gibt.

Achtung: INITIAL SIZE und APPEND SORTED BY

Im Zusammenhang mit dem Befehl `APPEND wa SORTED BY comp` hat der Zusatz `INITIAL SIZE` nicht nur eine syntaktische, sondern auch eine semantische Bedeutung (siehe Dokumentation). Der Befehl `APPEND wa SORTED BY comp` sollte aber nicht mehr verwendet werden. Stattdessen sollten Sie mit dem `SORT`-Befehl arbeiten.

Je nach Tabellentyp bzw. Art der Verarbeitung wird noch eine Verwaltung für den Zugriff auf die Zeilen benötigt, d. h. ein Index für die Indextabellen und eine Hash-Verwaltung für die Hash-Tabellen. An dieser Stelle soll nur darauf hingewiesen werden, dass es zusätzlich zu den Seiten auch noch Speicherbedarf für die Verwaltung der Einträge geben kann. Diese belegt ebenfalls Speicher. Sowohl im Debugger als auch im Memory Inspector wird dieser Speicher zum Tabellenkörper hinzugezählt und nicht separat ausgewiesen. Diese Verwaltung kann im Vergleich zu den Nutzdaten aber im Allgemeinen vernachlässigt werden.

Wie kann nun einmal allokierter Platz in internen Tabellen wieder freigegeben werden? Das Löschen einzelner oder mehrerer Zeilen aus der internen Tabelle mit dem Befehl `DELETE itab` führt zu keinerlei Speicherfreigabe. Die betroffenen Zeilen werden lediglich als gelöscht »markiert« und nicht aus den Seiten gelöscht.

Erst die Anweisungen `REFRESH` bzw. `CLEAR` führen dazu, dass die Seiten der internen Tabelle freigegeben werden. Es bleiben nur der Header und ein kleiner Speicherbereich bestehen.

Hinweis

Freigegeben bedeutet in diesem Zusammenhang, dass der einmal belegte Speicher verwendet werden kann. Da die Speicherallokation aus dem Extended Memory (EM) für einen Benutzer in der Regel in Blöcken erfolgt (siehe Abschnitt 6.1), die deutlich größer sind als die Seiten einer internen Tabelle, reden wir hier von einer zweistufigen Freigabe. Freigeben bedeutet also zunächst, dass die Seiten innerhalb eines EM-Blocks freigegeben werden und dieser Platz vom *selben* Benutzer wiederverwendet werden kann. Erst wenn ein EM-Block komplett leer ist und keinerlei Daten (Variablen etc.) des Benutzers mehr enthält, wird dieser Block an die SAP-Speicherverwaltung zurückgegeben und steht dann wieder anderen Benutzern zur Verfügung.

Die ABAP-Anweisung `FREE itab` hingegen führt zur kompletten Deallokation des Tabellenkörpers, d. h, es werden *alle Seiten* und der Index (falls vorhanden) der internen Tabellen freigegeben. Darüber hinaus wird noch der Tabellenkopf in eine systeminterne »Freiliste« zur Wiederverwendung eingefügt.

Falls eine interne Tabelle also weiterverwendet werden soll, empfiehlt sich der Einsatz von `REFRESH` oder `CLEAR` anstelle von `FREE`, da auf diese Weise das erneute Anlegen der ersten Seite entfällt. Falls ein Großteil der Zeilen einer internen Tabelle per `DELETE` gelöscht wurde und der noch belegte Speicher freigegeben werden soll, empfiehlt es sich, die Tabellenzeilen umzukopieren. Eine einfache Zuweisung in eine andere interne Tabelle reicht dazu wegen des Tabellen-Sharings, das in Abschnitt 7.4 besprochen wird, nicht aus. Alternativ können Sie zum Umkopieren auf ABAP-Statements (`INSERT` oder `APPEND`) oder auf die `EXPORT/IMPORT`-Varianten (siehe Abschnitt 6.2.2) zurückgreifen. Die »Freigabe« von Speicher spielt in diesem Zusammenhang für die Performance eine untergeordnete Rolle (solange kein Speicherengpass auf dem System vorliegt). Fragmentierte interne Tabellen haben, im Gegensatz zu fragmentieren Datenbanktabellen, keine negativen Auswirkungen auf die Performance, da die Einträge immer effizient adressierbar sind, weil interne Tabellen zeilenweise verwaltet werden.

Hintergrund: Unterschied zwischen internen Tabellen und Datenbanktabellen

Interne Tabellen sind in vielerlei Hinsicht vergleichbar mit Datenbanktabellen, es gibt aber einen wesentlichen Unterschied:

Interne Tabellen werden immer zeilenorientiert verarbeitet, während Datenbanktabellen mengenorientiert verarbeitet werden. Das heißt, eine mengenorientierte Verarbeitung, wie sie mit Open SQL auf Datenbanktabellen möglich ist, ist auf internen Tabellen nicht möglich, da bei den internen Tabellen immer die einzelne Zeile das Hauptverarbeitungskriterium ist, während es bei Datenbanktabellen immer eine Menge an Datensätzen ist. Mengenorientierte Zugriffe auf interne Tabellen wie z. B. `LOOP ... WHERE` oder `DELETE ... WHERE` werden von der ABAP VM emuliert und können bei manchen Tabellentypen optimiert abgebildet werden (siehe Abschnitt 7.4). Komplexere mengenorientierte Operatoren wie z. B. Joins und Aggregate sind auf internen Tabellen nicht möglich. Diese müssen mit den vorhanden ABAP-Sprachmitteln ausprogrammiert werden.

Nachdem wir die Organisation interner Tabellen im Hauptspeicher besprochen haben, wenden wir uns nun der Organisation der internen Tabellen selbst zu. Der folgende Abschnitt behandelt die verschiedenen Typen interner Tabellen.

7.3 Die Tabellentypen

Interne Tabellen lassen sich in Indextabellen und Hash-Tabellen unterteilen. Die Indextabellen lassen sich wiederum weiter in Standardtabellen und Sorted-Tabellen differenzieren. Abbildung 7.3 zeigt die Tabellenarten in der Übersicht.

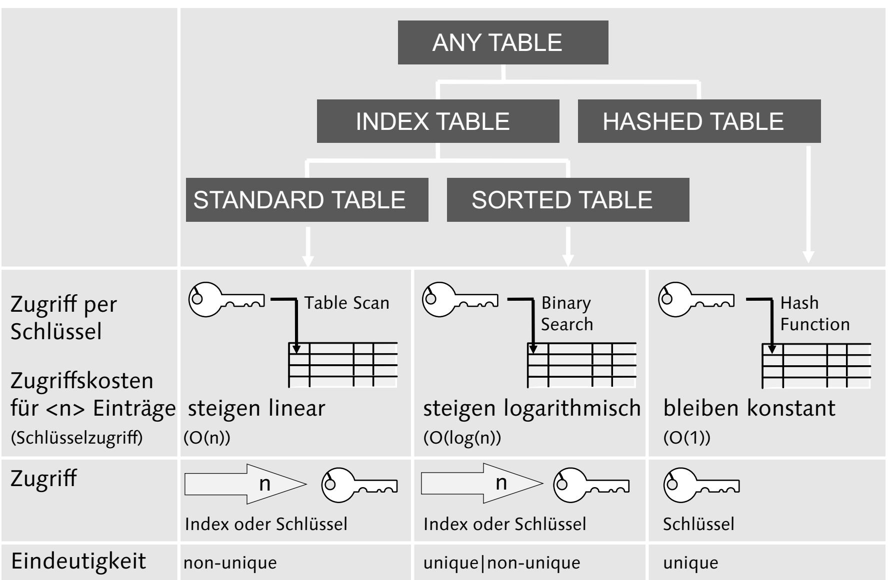


Abbildung 7.3 Die Tabellentypen im Überblick

Der Tabellentyp legt fest, wie man per ABAP auf einzelne Tabellenzeilen zugreifen kann.

Bei *Standardtabellen* kann der Zugriff über den Tabellenindex oder einen »Schlüssel« erfolgen. Beim Schlüsselzugriff hängt die Antwortzeit linear von der Anzahl der Tabelleneinträge ab, da der Lesezugriff (Read) einem linearen Scan der Einträge entspricht, der beim ersten Treffer abgebrochen wird. Der Schlüssel einer Standardtabelle ist immer nicht eindeutig (non-unique). Falls kein Schlüssel angegeben wird, erhält die Standardtabelle den so genannten *Default Key*, der sich aus allen zeichenartigen Feldern zusammensetzt.

Sorted-Tabellen sind immer nach dem Schlüssel sortiert. Der Zugriff kann über den Tabellenindex oder den Schlüssel erfolgen. Beim Schlüsselzugriff hängt die Zugriffszeit logarithmisch von der Anzahl der Tabelleneinträge ab, da der Lesezugriff (Read) über eine binäre Suche erfolgt. Der Schlüssel von sortierten Tabellen kann eindeutig (unique) oder nicht eindeutig (non-unique) sein. Auf Sorted-Tabellen können auch Teilschlüssel (Anfangsstücke des vollständigen

Schlüssels) optimiert verarbeitet werden. Auch ist eine Überspezifikation des Tabellenschlüssels möglich, es werden nur die Komponenten des Schlüssels für die Suche verwendet und die weiteren Komponenten anschließend für die Filterung genutzt.

Standardtabellen und sortierte Tabellen werden zusammenfassend auch als *Indextabellen* bezeichnet, weil auf beide Tabellen über den Tabellenindex zugegriffen werden kann.

Der Lesezugriff (Read) auf *Hash-Tabellen* ist nur über eine Schlüsselangabe möglich. Dabei ist die Antwortzeit konstant und hängt nicht von der Anzahl der Tabelleneinträge ab, da der Zugriff über einen Hash-Algorithmus erfolgt. Der Schlüssel von Hash-Tabellen muss eindeutig (unique) sein. Auf Hash-Tabellen sind weder explizite noch implizite Indexoperationen erlaubt. Wird auf eine Hash-Tabelle mit einem anderen »Schlüssel« als dem eindeutigen Tabellenschlüssel zugegriffen, wird die Tabelle wie eine Standardtabelle behandelt und linear nach den Einträgen durchsucht. Dies ist auch bei einem Teilschlüssel der Fall. Anders als bei der Sorted-Tabelle kann dieser bei der Hash-Tabelle nicht optimiert werden. Überspezifizierte Schlüssel werden optimiert verarbeitet.

Mithilfe der Anweisung `DESCRIBE TABLE <itab> KIND <k>` können Sie den aktuellen Tabellentyp zur Laufzeit ermitteln. Dies ist natürlich auch mit RTTI (Run Time Type Identification) möglich.

Zur effizienten Verwaltung bzw. Zugriffsoptimierung interner Tabellen gibt es einen Index oder eine Hash-Verwaltung. Welche Arten es gibt und wann diese angelegt werden, erfahren Sie im folgenden Abschnitt.

Indextabellen

Indizes werden bei Indextabellen erst dann angelegt, wenn die physische Reihenfolge nicht mehr der logischen Reihenfolge entspricht, d. h., wenn eine der Anweisungen `INSERT`, `DELETE` oder `SORT` auf der Tabelle ausgeführt wird und folgende Bedingungen zutreffen:

1. **`INSERT`**
   Der einzufügende Eintrag soll vor einem bereits bestehenden Eintrag eingefügt werden. (Eine `INSERT`-Anweisung, die *hinter* dem letzten Satz eingefügt wird, entspricht weitestgehend einer `APPEND`-Anweisung).
2. **`DELETE`**
   Der zu löschende Eintrag ist nicht der letzte Eintrag der Tabelle.
3. **`SORT`**
   Die Tabelle hat eine bestimmte Größe und wird sortiert.

Ein Index dient dem effizienten Indexzugriff in der »logischen Sortierreihenfolge« bzw. dem effizienten Auffinden »gültiger Zeilen«, wenn die Tabellenseiten Lücken durch das Löschen enthalten. Über den Index wird die logische Reihenfolge der Tabelle auf die physischen Speicheradressen der Einträge abgebildet.

Ein Index kann auf zwei Arten vorliegen:

1. als linearer Index
2. als baumartiger Index

Die Indexstruktur wird immer lückenlos gehalten, während in den Tabellenseiten Lücken durch das Löschen von Sätzen auftreten können. Die lückenlose Verwaltung der Tabellenseiten wäre, verglichen mit der lückenlosen Verwaltung des Index, bei größeren Tabellen zu zeitaufwändig.

Durch die lückenlose Verwaltung der Indexstruktur entstehen beim Einfügen und Löschen von Sätzen Schiebekosten, da die vorhandenen Einträge verschoben werden müssen. Genau genommen, handelt es sich dabei um Kopierkosten. Bei großen Indizes (ab ca. 5.000 Einträgen) nehmen diese überhand, weswegen für große Tabellen ein baumartiger Index angelegt wird.

Zusätzlich zum Index existiert eine Freiliste, die die Adressen der per DELETE gelöschten Einträge zur Wiederverwendung verwaltet.

In Abbildung 7.4 sehen Sie die schematische Darstellung eines linearen Index.

Ob ein baumartiger Index angelegt wird, hängt von systeminternen Regeln wie z. B. der Anzahl der (zu erwartenden) Einträge und weiteren Faktoren ab. In Abbildung 7.5 sehen Sie die schematische Darstellung eines baumartigen Index.

Beim baumartigen Index sind die Indexeinträge in so genannten *Blättern* organisiert. Die zuvor angesprochenen Schiebe- bzw. Kopierkosten fallen nur auf Blattebene an. Der Index muss nicht mehr an einem Stück allokiert werden, es wird nur auf Blattebene zusammenhängender Speicher am Stück benötigt. Im Gegenzug muss beim Zugriff auf den Index zunächst durch die Baumstruktur navigiert werden, um an den jeweiligen Indexeintrag zu gelangen.

Ansonsten sind baumartige Indizes auf Indextabellen mit den in Kapitel 5 vorgestellten Datenbankindizes vergleichbar. Ein baumartiger Index benötigt ca. 50 % mehr Platz als ein linearer Index.

Wenn bei der Verarbeitung von Indextabellen die logische Reihenfolge der Einträge der physischen Reihenfolge im Hauptspeicher entspricht, muss gar

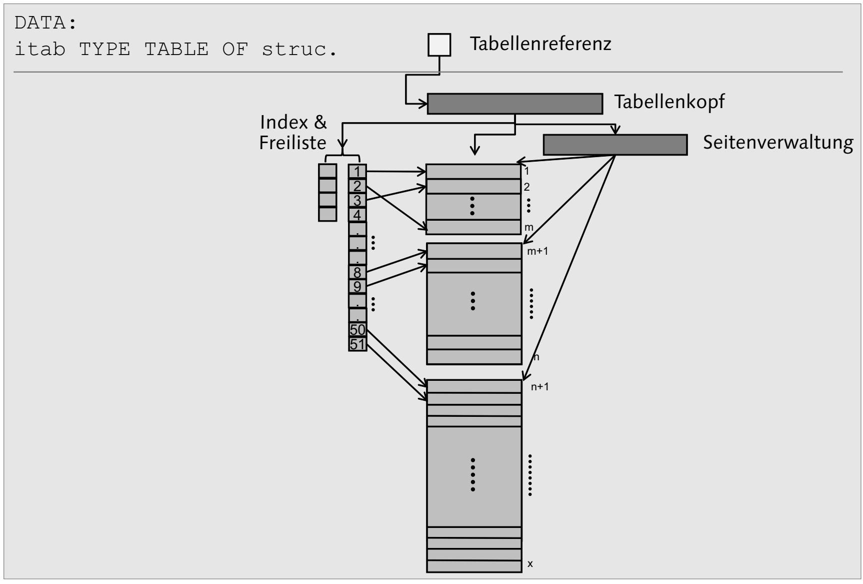


Abbildung 7.4 Schematische Darstellung eines linearen Index

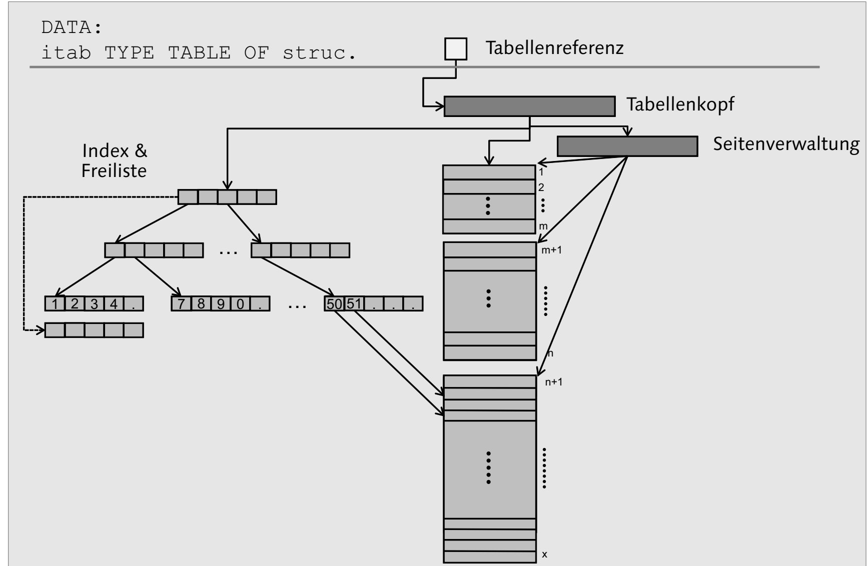


Abbildung 7.5 Schematische Darstellung eines baumartigen Index

kein Index angelegt werden. In diesem Fall entspricht die Einfügereihenfolge der physischen Reihenfolge, und die Tabelle wurde sortiert gefüllt und nicht gelöscht oder sortiert. Wenn kein Index benötigt wird, braucht die interne Tabelle etwas weniger Speicher.

Hash-Verwaltung

Die Hash-Verwaltung basiert auf dem eindeutigen Schlüssel der Tabelle. Der Hash-Verwaltung wird nur für Hash-Tabellen angelegt. Sie wird über den eindeutigen Schlüssel der internen Tabelle aufgebaut. Indexzugriffe (z. B. zweiter Eintrag der internen Tabelle) sind nicht möglich, auf Hash-Tabellen kann nur mit dem Schlüssel zugegriffen werden.

Bei der Hash-Tabelle wird jeder Schlüsselwert über eine Hash-Funktion einer eindeutigen Nummer zugewiesen. Zu dieser Nummer wird in einem entsprechenden Hash-Array die Speicheradresse des jeweiligen Datensatzes abgelegt.

Wenn auf einer Hash-Tabelle ein `DELETE` oder ein `SORT` ausgeführt wird, ist es notwendig, eine doppelt verlinkte Liste (vorheriger und nächster Zeiger) anzulegen, damit sequenzielle Zugriffe (`LOOP`) über die Daten gemäß Einfügereihenfolge (bzw. in einer durch `SORT` erzeugten Sortierreihenfolge) weiterhin möglich sind. Durch die doppelt verlinkte Liste wird etwa 50 % mehr Platz für die Hash-Verwaltung benötigt.

In Abbildung 7.6 sehen Sie die schematische Darstellung einer Hash-Verwaltung.

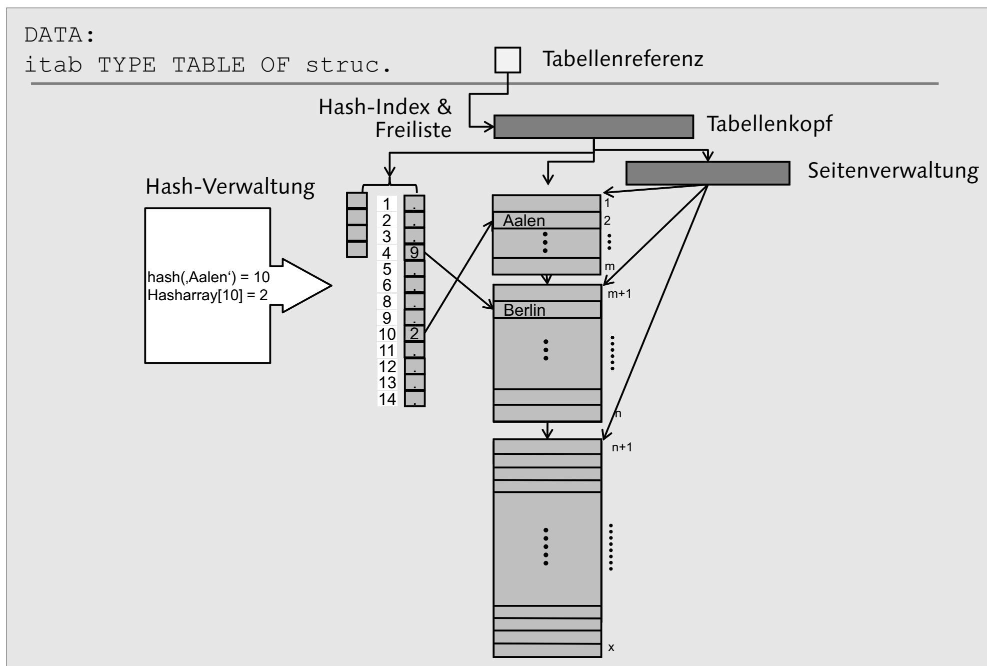


Abbildung 7.6 Schematische Darstellung eines Hash-Index

Beschränkungen

Für interne Tabellen gibt es neben dem für einen Benutzer zur Verfügung stehenden Speicher weitere Beschränkungen:

Eine Grenze für die Anzahl der Zeilen in internen Tabellen ergibt sich daraus, dass sie intern und in ABAP-Anweisungen über 4-Byte-Integers adressiert werden, was sie auf 2.147.483.647 Einträge beschränkt.

Die Größe von Hash-Tabellen wird zusätzlich durch den größten an einem Stück verfügbaren Speicherblock beschränkt. Dieser beträgt maximal 2 GB, wird in der Regel aber über den *Profilparameter* `ztta/max_memreq_MB` weiter eingeschränkt. Die maximale Zeilenzahl von Hash-Tabellen hängt von der erforderlichen Größe der Hash-Verwaltung ab, die dort untergebracht werden muss.

Die tatsächliche maximale Größe interner Tabellen wird in der Regel nochmals kleiner sein, als sie durch obige Grenzen gegeben ist, da der insgesamt verfügbare Speicher normalerweise nicht nur von einem String oder einer internen Tabelle genutzt wird (siehe ABAP-Dokumentation: MAXIMALE GRÖSSE DYNAMISCHER DATENOBJEKTE).

Zusammenfassung der Tabellentypen

Die folgende Tabelle stellt die wichtigsten Merkmale der Tabellentypen nochmals dar. Es folgt eine Empfehlung, wann welcher Tabellentyp eingesetzt werden sollte.

	Standardtabelle	Sorted-Tabelle	Hash-Tabelle
Mögliche Zugriffe	Indexzugriff oder Schlüsselzugriff	Indexzugriff oder Schlüsselzugriff	Schlüsselzugriff
Eindeutigkeit	non-unique	non-unique oder unique	unique
Optimaler Zugriff	Index oder Binärsuche (falls die Tabelle nach den Suchkomponenten sortiert ist)	Index oder Schlüssel	Schlüssel

Tabelle 7.1 Merkmale der Tabellentypen

Standardtabellen sollten dann eingesetzt werden, wenn nach dem Füllen alle Einträge sequenziell verarbeitet werden sollen oder wenn flexibel mit mehreren verschiedenen Schlüsseln effizient auf die interne Tabelle zugegriffen wer-

den soll. Dazu muss die Tabelle nach dem Suchfeld sortiert sein und mit der Binärsuche durchsucht werden. Achten Sie darauf, dass das Umsortieren nur so häufig wie nötig geschieht. Falls ein Umsortieren nur für einen oder wenige Lesezugriffe notwendig ist, würden die Sortierzeiten die Zeitersparnis beim Lesen bei Weitem überwiegen. Schlüsselzugriffe ohne Binärsuche sollten nur bei kleinen Tabellen verwendet oder besser ganz vermieden werden. Wenn nur über ein bestimmtes Feld gesucht wird, sollte eine Sorted- oder Hash-Tabelle verwendet werden.

Sorted-Tabellen bieten sich für die teilsequenzielle Verarbeitung an, wenn z. B. ein verhältnismäßig kleiner Teil einer Tabelle über Schlüsselzugriffe, bei denen nur der Anfang des Schlüssels benannt ist, verarbeitet werden soll. Auch Schlüsselzugriffe auf den Tabellenschlüssel können bei Sorted-Tabellen effizient abgewickelt werden.

Hash-Tabellen sind optimal, wenn nur über den Tabellenschlüssel zugegriffen wird. Wenn der Schlüssel eine hohe Linkssignifikanz hat, bietet sich auch eine eindeutige Sorted-Tabelle an, da sich in diesem Fall für die Binärsuche Performancevorteile beim Zugriff auf einzelne Zeilen ergeben. Mit *Linkssignifikanz* ist hier gemeint, dass sich der selektive Teil eines Schlüssels möglichst weit am Anfang (links) im Schlüssel befinden sollte.

7.4 Performanceaspekte

In diesem Abschnitt werden nun alle performancerelevanten Aspekte beim Umgang mit internen Tabellen behandelt. Dabei werden wir die wichtigsten Befehle für interne Tabellen betrachten. Die Beispiele sind mit einem Arbeitsbereich (`wa`) aufgeführt. Die Verarbeitung mit Kopfzeilen wird zwar noch unterstützt, soll aber nicht mehr verwendet werden, da die Kopfzeilen auf interne Tabellen obsolet und im OO-Kontext verboten sind.

7.4.1 Füllen

Wie auch schon bei den Datenbankzugriffen stehen Ihnen auch für die internen Tabellen Mengenoperationen und Einzelsatzoperationen zur Verfügung.

Mengenoperationen

Diese Art von Verarbeitung wird in der ABAP-Dokumentation generell als Blockverarbeitung beschrieben, während im Bezug auf die Datenbank bei der `SELECT`-Anweisung von Array-Operationen die Rede ist.

Wenn interne Tabellen aus Datenbanktabellen gefüllt werden, sorgt das Schlüsselwort `INTO TABLE itab` bei der `SELECT`-Anweisung dafür, dass die Datensätze en bloc in die interne Tabelle eingefügt werden (siehe Abschnitt 5.7).

Auch beim Füllen interner Tabellen aus anderen internen Tabellen steht ein Array-Interface zur Verfügung. Die entsprechenden ABAP-Anweisungen sind:

```
APPEND LINES OF itab1 TO itab2.
INSERT LINES OF itab1 INTO  TABLE itab2.
```

Für Hash-Tabellen kann nur das `INSERT` Statement verwendet werden, während für Indextabellen sowohl `APPEND` als auch `INSERT` infrage kommt. Beim Anhängen von Zeilen mittels `APPEND` muss bei Sorted-Tabellen aber sichergestellt sein, dass die Sortierreihenfolge der internen Tabelle gewahrt bleibt.

Ebenso gehören Zuweisungen per `MOVE` und = zu den Mengenoperationen auf internen Tabellen.

Es ergeben sich hier zwischen den Tabellentypen geringe Laufzeitunterschiede, die abhängig von der Einfügeposition und der Menge der einzufügenden Einträge sind. Für die Verwaltung der Indizes fallen relativ geringe Kosten an.

Sie sollten Blockoperationen auf internen Tabellen den Einzeloperationen (nächster Abschnitt) auf jeden Fall vorziehen, wo immer dies möglich ist, da Verwaltungsarbeiten (wie z.B. Speicherallokation) effizienter vom Kernel abgearbeitet werden können.

Es sollte aber folgendes Verhalten beachtet werden, da die Reihenfolge der Zeilen in internen Tabellen, im Gegensatz zu Datenbanktabellen, immer wohldefiniert ist:

- Bei Duplikaten und Non-unique-Schlüsseln wird man bei Blockoperationen in der Zieltabelle immer die gleiche Reihenfolge innerhalb der Duplikate wie in der Quelltabelle haben. Dies ist bei Einzelsatzoperationen nicht so, dort kann sich die Reihenfolge bei Duplikaten ändern.
- Bei Duplikaten und Unique-Schlüsseln kommt es bei Blockoperationen zu nicht abfangbaren Laufzeitfehlern, während bei Einzelsatzoperationen lediglich der Return-Code `sy-subrc` gesetzt wird.

Praxisbeispiel – SE30, Tipps & Tricks:

In der Transaktion SE30 finden Sie bei den Tipps & Tricks unter Internal Tables • Array Operations verschiedene Beispiele, deren Laufzeiten Sie vermessen können.

Einzelsatzoperationen

Auch bei den Einzelsatzoperationen stehen Ihnen die ABAP-Anweisungen APPEND und INSERT zur Verfügung:

```
APPEND wa TO itab.
INSERT wa INTO itab INDEX indx.
INSERT wa INTO TABLE itab.
```

Während eine APPEND- und ein eine INSERT-Anweisung mit dem Zusatz INDEX nur auf Indextabellen verwendet werden können, steht die dritte Variante für alle Tabellen zur Verfügung.

Bei Standardtabellen entspricht eine INSERT-Anweisung ohne INDEX weitestgehend dem APPEND-Statement. (Bei APPEND muss die anzufügende Zeile konvertibel sein, während die einzufügende Zeile bei INSERT kompatibel sein muss, siehe ABAP-Dokumentation.) Die Kosten für das APPEND-Statement sind konstant. Ein APPEND ist die schnellste Variante beim Einfügen von Einzelsätzen, da hier nur ein Eintrag ans Ende der Tabelle angehängt wird.

Das Einfügen an einer bestimmten Stelle (INSERT ... INDEX) erfordert je nach Einfügeposition Schiebekosten, die umso höher werden, je weiter »vorne« der Eintrag eingefügt wird (mehr Schiebekosten). Dagegen werden sie umso günstiger, je weiter »hinten« der Eintrag eingefügt wird (weniger Schiebekosten). Bis zu einer gewissen Grenze (derzeit 4.096) sind die Kosten für das Einfügen von der Einfügeposition und linear von der Anzahl der Einträge abhängig. Sobald eine Indextabelle mehr Einträge hat, wird intern auf einen baumartigen Index umgestellt, bei dem die Schiebekosten und die Einfügeposition nur noch auf Blattebene relevant sind. Sobald ein baumartiger Index vorliegt, skalieren die Kosten dann nicht mehr linear, sondern logarithmisch mit der Anzahl der Einträge.

Ein Einfügen mit Index auf Standardtabellen bietet sich an, um diese sortiert aufzubauen. Dazu muss die korrekte Einfügeposition zunächst ermittelt werden, wenn sie noch nicht bekannt ist. Dies erfolgt am besten über die Binärsuche (siehe nächster Abschnitt).

Bei Sorted-Tabellen können eine APPEND- und eine INSERT-Anweisung mit dem Zusatz INDEX nur dann verwendet werden, wenn die Sortierreihenfolge gewahrt bleibt. In diesem Fall muss geprüft werden, ob der Schlüssel des neuen Eintrags an die gewünschte Position in der Tabelle passt.

Beim generischen INSERT (ohne den Zusatz INDEX) erfolgt eine Binärsuche, über die intern die richtige Einfügeposition ermittelt wird. Die Kosten für das Auf-

finden der Position entsprechen einem Lesezugriff mit Schlüssel auf diese Tabelle und skalieren logarithmisch mit der Anzahl der Einträge. Wie bei der Standardtabelle kommen noch die Schiebekosten hinzu. Diese hängen von der Einfügeposition und dem Index (linear oder baumartig) ab.

Bei Hash-Tabellen wird anhand des Tabellenschlüssels eingefügt. Die Kosten hierfür sind konstant und damit unabhängig von der Anzahl der Einträge. Der Weg über die Hash-Verwaltung ist etwas aufwändiger als das Anhängen von Einträgen an die Standardtabelle.

Zusammenfassend lässt sich für das Einfügen also sagen, dass Mengenoperationen, wo immer möglich, vorzuziehen sind. Beachten Sie jedoch die zuvor genannten Verhaltensweisen dieser Operationen.

Die Kosten für die Einzelsatz-Anweisungen finden Sie in Tabelle 7.2 nochmals in der Übersicht: Bitte beachten Sie, dass die Kosten für Reorganisation der Index- bzw. Hash-Verwaltung bei interner Speichererweiterung bzw. bei der Verwaltung des baumartigen Index hier nicht berücksichtig sind.

	Standard	Sorted	Hashed
`APPEND`	O(1) konstant	O(1) konstant (höher als Standard, Check nötig)	–
`INSERT ... INTO ... INDEX`	linearer Index: O(1) – O(n) konstant – linear baumartiger Index: O(1) – O(log n) konstant – logarithmisch (abhängig von Position)	linearer Index: O(1) – O(n) konstant – linear baumartiger Index: O(1) – O(log n) konstant – logarithmisch (abhängig von Position) konstant (etwas höher, Check notwendig)	–
`INSERT ... INTO TABLE`	O(1) konstant	O(log n) logarithmisch	O(1) konstant (höher als Standard, Hash-Verwaltung)

Tabelle 7.2 Kosten von Einzelsatzoperationen beim Füllen interner Tabellen

7.4.2 Lesen

Bei den Lesezugriffen unterscheiden wir zwischen dem Lesen mehrerer oder einzelner Zeilen.

Mehrere Zeilen (LOOP)

Hier unterscheiden wir zwischen dem Lesen aller Zeilen und dem Lesen eines Teils der Zeilen.

Alle Zeilen werden mit dem ABAP-Statement `LOOP AT itab` gelesen. Hierbei werden alle Zeilen einer internen Tabelle gelesen. Die Kosten für das Lesen aller Datensätze skaliert linear mit der Anzahl der Datensätze. Diese Kosten sind unabhängig von der Tabellenart, da jeder Eintrag in der internen Tabelle bearbeitet werden muss. Ohne weitere Angabe wird dazu jeder Eintrag in den mit `INTO` angegebenen Arbeitsbereich *kopiert*.

Ein Teil der Zeilen einer internen Tabelle wird mit `LOOP ... FROM ix1 TO ix2` (bei Indextabellen) oder generell mit `LOOP ... WHERE` gelesen. Die Kosten für das Lesen eines Teilbereichs der internen Tabelle hängen davon ab, wie groß dieser Teil ist und ob der zu lesende Teil effizient gefunden werden kann. Es fallen die Kosten für das Bereitstellen der Ergebnismenge im Ausgabebereich (`LOOP ... INTO`) an. Weit wichtiger sind aber die Kosten für das Auffinden der passenden Einträge.

Bei Standardtabellen verhalten sich hier die Kosten linear zur Anzahl der Einträge.

Bei Hash-Tabellen kann, wenn der vollständige Schlüssel der Tabelle in der `WHERE`-Bedingung angegeben ist, eine Suche über die Hash-Verwaltung vorgenommen werden. Dann entspricht der `LOOP ... WHERE` dem Read eines eindeutigen Satzes. Die Kosten sind dann konstant. In allen anderen Fällen verhalten sich Lesezugriffe auf die Hash-Tabelle linear – abhängig von der Anzahl der Einträge, da die Tabelle komplett durchsucht wird.

Beim Zugriff auf Sorted-Tabellen kann aufgrund der Tatsache, dass die Tabelle per Definition sortiert vorliegt, auch ein unvollständiger Schlüssel vom Kernel optimiert werden. Dazu müssen folgende Bedingungen gegeben sein:

1. Die `WHERE`-Bedingung hat die Form `WHERE k1 = b1 AND ... AND kn = bn`.
2. Die `WHERE`-Bedingung deckt ein Anfangsstück des Tabellenschlüssels ab.

Im Gegensatz zu Hash-Tabellen werden bei Sorted-Tabellen so auch teilsequenzielle Zugriffe optimiert. Der Aufsatzpunkt für den gesuchten Bereich kann so effizient gefunden werden.

Wenn Standardtabellen nach dem Schlüssel sortiert sind, kann auch eine Optimierung erreicht werden, indem zunächst der erste passende Eintrag über eine Binärsuche gesucht wird und anschließend von dieser Position aus ein Loop gestartet wird. Dieser wird verlassen, sobald mittels einer IF-Anweisung festgestellt wird, dass die Suchbedingung nicht mehr zutrifft. Die Kosten für diese Vorgehensweise entsprechen etwa denen der Sorted-Tabelle und skalieren logarithmisch zu den Tabelleneinträgen. Das folgende Listing zeigt ein Pseudocode-Beispiel für diese Vorgehensweise:

```
READ TABLE itab INTO wa WITH KEY ... BINARY SEARCH.
   INDEX = SY-TABIX.
   LOOP AT itab INTO wa FROM INDEX.
   IF ( key <> search_key ).
      EXIT.
   ENDIF.
   ENDLOOP.
```

Beim Massenzugriff ergeben sich folgende Kosten wie in Tabelle 7.3 dargestellt. Bitte beachten Sie, dass die Kosten außer Suchkosten für das Auffinden der passenden Einträge (wie sie in der Tabelle dargestellt sind) auch Bereitstellungskosten der Treffermenge (z. B. im Arbeitsbereich oder einer Datenreferenz) enthalten. Die Bereitstellungskosten der Treffermenge sind bei kleinen Treffermengen von untergeordneter Bedeutung. Lediglich beim LOOP ... FROM ... TO, bei dem die Suchkosten konstant sind, kann die Bereitstellung der Treffermenge die Kosten dominieren.

Für große Treffermengen oder den Extremfall, dass aufgrund von Duplikaten bezogen auf den Schlüssel alle Zeilen der internen Tabelle die Treffermenge bilden, werden die Kosten auf Indextabellen von den Bereitstellungskosten dominiert, die linear mit der Anzahl der Treffer skalieren.

	Standard	**Sorted**	**Hashed**
LOOP ... ENDLOOP (alle Zeilen)	O(n) linear (Full Table Scan)	O(n) linear (Full Table Scan)	O(n) linear (Full Table Scan)
LOOP ... WHERE ENDLOOP (vollständiger Schlüssel)	O(n) linear (Full Table Scan)	O(log n) logarithmisch	O(1) konstant

Tabelle 7.3 Kosten beim Lesen mehrerer Zeilen aus internen Tabellen

	Standard	Sorted	Hashed
`LOOP ... WHERE ENDLOOP` (unvollständiger Schlüssel, Anfangsstück)	O(n) linear (Full Table Scan) Kann manuell optimiert werden durch sortierte Standardtabelle und Binärsuche O(log n).	O(log n) logarithmisch	O(n) linear (Full Table Scan)
`LOOP ... WHERE ENDLOOP` (unvollständiger Schlüssel, kein Anfangsstück)	O(n) linear (Full Table Scan)	O(n) linear (Full Table Scan)	O(n) linear (Full Table Scan)
`LOOP ... FROM ... TO`	O(1) konstant	O(1) konstant	–

Tabelle 7.3 Kosten beim Lesen mehrerer Zeilen aus internen Tabellen (Forts.)

Einzelne Zeilen

Um einzelne Zeilen aus internen Tabellen zu lesen, stehen folgende Anweisungen zur Verfügung:

```
READ TABLE itab INTO wa INDEX ...
READ TABLE itab INTO wa WITH [TABLE] KEY ...
READ TABLE itab INTO wa FROM wa1
```

Indexzugriffe können nur auf Indextabellen ausgeführt werden und haben konstante Kosten.

In der Regel wird man aber nicht mit dem Index, sondern mit dem Schlüssel auf eine interne Tabelle zugreifen wollen. In diesem Fall sind die Kosten wieder vom Aufwand, den richtigen Eintrag zu finden, abhängig.

Bei Standardtabellen sind die Kosten linear von der Anzahl der Einträge abhängig, da die Tabelle Eintrag für Eintrag gescannt wird, bis ein passender Eintrag gefunden wurde. Befindet sich der Eintrag am Anfang der Tabelle, ist die Suche früher beendet, als wenn sich der Eintrag am Ende der Tabelle befindet.

Eine Möglichkeit, die Suche in einer Standardtabelle zu beschleunigen, ist die Verwendung der Binärsuche. Dazu muss die Standardtabelle aber nach dem

Suchbegriff sortiert vorliegen und ein Anfangsstück des Sortierschlüssels vorhanden sein. Mit dem Statement `READ itab WITH KEY ... BINARY SEARCH` wird eine Binärsuche auf der Standardtabelle verwendet. Die Kosten skalieren in diesem Falle logarithmisch mit der Anzahl der Einträge.

Hintergrund: Binary Search

Die binäre Suche auf Standard- oder Sorted-Tabellen arbeitet mit dem Intervallhalbierungsverfahren. Die Tabelle muss dazu nach dem jeweiligen Schlüssel sortiert vorliegen. Dabei wird nicht am Anfang der Tabelle mit der Suche begonnen, sondern in der Mitte, und dann wird die Hälfte, in der sich der Eintrag befindet, wieder halbiert etc., bis ein Treffer vorliegt oder kein Satz gefunden wird. Falls Duplikate vorliegen, wird der erste Eintrag in der Duplikatliste zurückgeliefert.

Achten Sie darauf, dass die Standardtabelle nicht unnötigerweise sortiert wird, denn da es sich beim Sortieren einer Standardtabelle ebenfalls um eine teure Anweisung handelt (siehe Abschnitt 7.4.6), muss die Anzahl der Sortiervorgänge so klein wie möglich gehalten werden.

Praxisbeispiel – SE30, Tipps & Tricks:

In der Transaktion SE30 finden Sie bei den Tipps & Tricks unter Internal Tables • Linear search vs. Binary search ein Beispiel, dessen Laufzeit Sie vermessen können.

Die binäre Suche kann auch zur Optimierung teilsequenzieller Zugriffe verwendet werden, wie dies am Anfang dieses Abschnitts beim `LOOP ... WHERE` auf Standardtabellen gezeigt wurde. Auch um eine Standardtabelle sortiert aufzubauen, kann eine Binärsuche verwendet werden. Schauen Sie sich hierzu nochmals das Beispiel aus Abschnitt 6.2.1 an:

```
READ TABLE it_kunde INTO var_kunde
WITH KEY it_order_tab-kunnr BINARY SEARCH.
save_tabix = sy-tabix.
IF SY-SUBRC <> 0.
       SELECT *
       INTO var_kunde
       FROM db_kunden_tab
       WHERE kundennr = it_order_tab-kunnr.
          IF SY-SUBRC = 0.
            INSERT var_kunde INTO it_kunde INDEX save_tabix.
           ...
```

Die Tabelle `it_kunde` wird mit der Binärsuche auf einen passenden Eintrag durchsucht. Wenn kein passender Eintrag gefunden wird, steht der Tabellenindex `sy-tabix` auf der Zeilennummer, an der der Eintrag hätte stehen müssen. Dieser Index kann dann für das Einfügen des Eintrags an der richtigen Position

verwendet werden. So wird die Standardtabelle sortiert aufgebaut, ohne dass eine SORT-Anweisung notwendig ist.

Bei Lesezugriffen auf Sorted-Tabellen wird intern eine Binärsuche verwendet, wenn ein Anfangsstück des Tabellenschlüssels vorliegt. Die Kosten skalieren logarithmisch mit der Anzahl der Einträge.

Bei Hash-Tabellen wird bei einem vollständig spezifizierten Schlüsselzugriff die Hash-Verwaltung verwendet. Die Kosten dafür sind konstant. Falls mit einem nicht vollständig spezifizierten Schlüssel auf die Hash-Tabelle zugegriffen wird, sind die Kosten linear von der Anzahl der Einträge abhängig.

Bei allen Zugriffen ist es für die Performance unerheblich, ob mit dem Tabellenschlüssel (...WITH TABLE KEY...) oder mit einem freien Schlüssel (...WITH KEY...) zugegriffen wird. Für die Performance ist allein entscheidend, dass die genannten Schlüsselfelder mit dem Anfang bzw. dem vollständigen Tabellenschlüssel übereinstimmen. Es werden also auch überspezifizierte Suchzugriffe (mit mehr Feldern als den Schlüsselfeldern) auf interne Tabellen optimiert.

Beim Einzelsatzzugriff ergeben sich Kosten wie in Tabelle 7.4 dargestellt. Wie bereits beim LOOP ... WHERE erwähnt, kommt für Duplikate bei der Binärsuche bei den Indextabellen noch ein linearer Anteil hinzu, der im Extremfall (alle Einträge sind bezogen auf den Schlüssel Duplikate) ein lineares Laufzeitverhalten zeigen kann.

	Standard	Sorted	Hashed
READ ... INDEX	O(1) konstant	O(1) konstant	–
READ ... WITH KEY ... (vollständiger Schlüssel)	O(n) linear Binärsuche: O(log n) logarithmisch	O(log n) logarithmisch	O(1) konstant
READ ... WITH KEY ... (unvollständiger Schlüssel, Anfangsstück)	O(n) linear Binärsuche: O(log n) logarithmisch	O(log n) logarithmisch	O(n) linear
READ ... WITH KEY ... (unvollständiger Schlüssel, kein Anfangsstück)	O(n) linear	O(n) linear	O(n) linear

Tabelle 7.4 Kosten beim Lesen einzelner Zeilen aus internen Tabellen

7.4.3 Ändern

Interne Tabellen werden mit dem MODIFY-Befehl geändert. Beim MODIFY auf interne Tabellen handelt es sich *nur* um ein Ändern und nicht, wie beim MODIFY-Befehl auf eine Datenbanktabelle, um ein Ändern oder Einfügen.

Mehrere Zeilen einer internen Tabelle werden mit folgendem Statement geändert:

```
MODIFY itab FROM wa TRANSPORTING ... WHERE ...
```

Die Kosten sind die gleichen wie beim LOOP ... WHERE-Statement und hängen von der Anzahl der zu ändernden Einträge und dem Aufwand für das Auffinden der Einträge ab.

Einzelne Einträge in internen Tabellen können wie folgt geändert werden:

```
MODIFY itab [INDEX n] [FROM wa]
MODIFY TABLE itab [FROM wa]
```

Beim Zugriff auf Indextabellen über den Index (Variante 1) sind die Kosten konstant. Innerhalb von Loops kann diese Variante auch zur sequenziellen Änderung mehrerer Zeilen ohne INDEX benutzt werden. In diesem Fall wird die aktuelle Zeile, auf der sich der Loop gerade befindet, geändert. Hierbei handelt es sich um eine implizite Indexoperation, die nur auf Indextabellen zulässig ist.

Bei den Schlüsselzugriffen (Variante 2) mit vollständigem Schlüssel skalieren die Kosten bei Standardtabellen linear und bei Sorted-Tabellen logarithmisch mit der Anzahl der Einträge. Bei Hash-Tabellen sind die Kosten konstant. Da diese Variante eine eigene Suche der passenden Einträge beinhaltet, sollte sie nicht in der Loop-Schleife über dieselbe Tabelle eingesetzt werden. Dies könnte ein nicht-lineares Laufzeitverhalten zur Folge haben.

Die Kosten für den MODIFY entsprechen denen des LOOP, es gelten die gleichen Einschränkungen für die Duplikate (siehe Tabelle 7.5).

	Standard	Sorted	Hashed
MODIFY ... TRANSPORTING ... WHERE (vollständiger Schlüssel)	O(n) linear (Full Table Scan)	O(log n) logarithmisch	O(1) konstant

Tabelle 7.5 Kosten beim Ändern interner Tabellen

	Standard	Sorted	Hashed
MODIFY... TRANSPORTING... WHERE (unvollständiger Schlüssel, Anfangsstück)	O(n) linear (Full Table Scan)	O(log n) logarithmisch	O(n) linear (Full Table Scan)
MODIFY... TRANSPORTING... WHERE (unvollständiger Schlüssel, kein Anfangsstück)	O(n) linear (Full Table Scan)	O(n) linear (Full Table Scan)	O(n) linear (Full Table Scan)
MODIFY ... [INDEX n] FROM wa (Indexzugriff)	O(1)	O(1)	–
MODIFY TABLE... FROM wa (Suchaufwand wie bei WHERE)	O(n) linear (Full Table Scan)	O(log n)	O(1) konstant

Tabelle 7.5 Kosten beim Ändern interner Tabellen (Forts.)

7.4.4 Löschen

Um mehrere Einträge aus internen Tabellen zu löschen, stehen die folgenden Anweisungen zur Verfügung:

```
DELETE itab FROM ix1 TO ix2
DELETE itab WHERE...
```

Die Kosten werden vom Aufwand des Auffindens und der Menge der zu löschenden Zeilen bestimmt. Beim Indexzugriff sind die Kosten für das Auffinden konstant, beim Schlüsselzugriff die gleichen wie beim MODIFY.

Zugriffe auf einzelne Einträge werden über folgende Anweisungen realisiert:

```
DELETE itab [INDEX n].
DELETE TABLE itab WITH TABLE KEY .../DELETE TABLE itab FROM wa
```

Bei den Zugriffen auf einzelne Zeilen verhalten sich die Kosten auch wie beim LOOP bzw. MODIFY (siehe Tabelle 7.6).

	Standard	Sorted	Hashed
DELETE ... FROM ... TO	O(1)	O(1)	–
DELETE ... WHERE (vollständiger Schlüssel)	O(n) linear (Full Table Scan)	O(log n) logarithmisch	O(1) konstant
DELETE ... WHERE (unvollständiger Schlüssel, Anfangsstück)	O(n) linear (Full Table Scan)	O(log n) logarithmisch	O(n) linear (Full Table Scan)
DELETE ... WHERE (unvollständiger Schlüssel, kein Anfangsstück)	O(n) linear (Full Table Scan)	O(n) linear (Full Table Scan)	O(n) linear (Full Table Scan)
DELETE ... INDEX	O(1)	O(1)	–
DELETE FROM WA / DELETE TABLE WITH TABLE KEY	O(n) linear (Full Table Scan)	O(log n)	O(1) konstant

Tabelle 7.6 Kosten beim Löschen von Einträgen aus internen Tabellen

7.4.5 Verdichten

Mit dem Befehl `COLLECT` können verdichtete Datenbestände in internen Tabellen aufgebaut werden. Dabei werden die numerischen Daten aller Felder, die keine Schlüsselfelder sind, auf bereits vorhandene Werte mit demselben Schlüssel in der internen Tabelle aufsummiert. Bei Standardtabellen ohne explizite Schlüsselangabe werden alle nicht-numerischen Felder als Schlüsselfelder behandelt. Die Kosten des Befehls werden maßgeblich vom Aufwand, die betreffende Zeile zu finden, bestimmt.

Bei Standardtabellen wird eine temporäre Hash-Verwaltung angelegt, wenn eine Standardtabelle nur mit `COLLECT` gefüllt wird. Diese ist instabil gegenüber anderen modifizierenden Anweisungen (`APPEND`, `INSERT`, `DELETE`, `SORT`, `MODIFY`, Änderungen über Feldsymbole/Referenzen). Diese Optimierung ist aber durch die Einführung der Key-Tabellen (Sorted-Tabellen, Hash-Tabellen) obsolet geworden und daher auch der `COLLECT`-Befehl auf Standardtabellen.

Das Auffinden der Einträge ist bei intakter temporärer Hash-Verwaltung wie bei Hash-Tabellen konstant. Wenn die Hash-Verwaltung zerstört ist, ist der Aufwand für das Suchen der Einträge linear von der Anzahl der Einträge der internen Tabelle abhängig. Mit dem Funktionsbaustein `ABL_TABLE_HASH_STATE` kann überprüft werden, ob eine Standardtabelle eine intakte Hash-Verwaltung besitzt.

Bei Sorted-Tabellen wird der Eintrag intern mit einer Binärsuche bestimmt, wobei der Aufwand für das Suchen der Einträge logarithmisch von der Anzahl der Einträge in der internen Tabelle abhängt.

In Hash-Tabellen wird der Eintrag über die Hash-Verwaltung der Tabelle bestimmt. Die Kosten sind konstant und unabhängig von der Anzahl der Einträge.

`COLLECT` sollte hauptsächlich für Hash-Tabellen verwendet werden, da diese einen eindeutigen Tabellenschlüssel und eine stabile Hash-Verwaltung haben.

Praxisbeispiel – SE30, Tipps & Tricks:

In der Transaktion SE30 finden Sie bei den Tipps & Tricks unter Internal Tables • Building condensed tables ein Beispiel, dessen Laufzeit Sie vermessen können.

7.4.6 Sortieren

Standard- und Hash-Tabellen können nach einem beliebigen Feld der Tabelle mit dem Befehl `SORT` sortiert werden. Sorted-Tabellen können nicht mit dem Befehl `SORT` sortiert werden, da sie per Definition schon nach den Schlüsselfeldern sortiert sind und auch nicht nach anderen Feldern umsortiert werden können.

Beim Sortieren werden die Daten nach Möglichkeit im Hauptspeicher (im prozesslokalen Speicher eines Workprozesses) sortiert. Falls der Platz im Hauptspeicher nicht ausreicht, werden die zu sortierenden Komponenten im Dateisystem sortiert. Dazu werden zunächst Blöcke im Hauptspeicher sortiert und ins Dateisystem geschrieben. Anschließend werden diese sortierten Blöcke über einen Merge-Sort dann wieder eingelesen.

Beim Sortieren handelt es sich, egal, ob im Hauptspeicher oder im Dateisystem sortiert wird, um eine laufzeitintensive Anweisung. (Das Sortieren im Dateisystem ist natürlich noch teurer als das Sortieren im Hauptspeicher.) Es sollte also nur dann sortiert werden, wenn es von der Anwendung her zwingend erforderlich ist oder, wie im Falle der Standardtabelle, Laufzeitgewinne für das Lesen aus diesen Tabellen über die Binärsuche erzielt werden können. So ist es z. B. möglich, eine interne Standardtabelle zunächst nach einem und später nach einem anderen Schlüsselfeld zu sortieren und über die Binärsuche zu durchsuchen. Achten Sie in diesem Fall aber darauf, dass die erzielten Laufzeitgewinne durch die Binärsuche nicht durch den erhöhten Aufwand des Sortierens wieder zunichte gemacht werden. Die Sortierung lohnt sich nur, wenn dadurch eine große Anzahl nachfolgender Lesezugriffe optimiert werden kann.

Bei einer Tabelle mit etwa 1.000 Zeilen sollten mindestens 40–50 Lesezugriffe auf einen Sortiervorgang folgen.

Falls die interne Standardtabelle derart verarbeitet wird, dass immer ein Suchzugriff auf ein Feld wechselweise mit einem Suchzugriff auf ein anderes Feld erfolgt und somit pro Suchzugriff ein Sortiervorgang für das jeweilige Umsortieren notwendig wäre, wäre es kontraproduktiv, die Sortierung durchzuführen. In diesem Fall sollte nur einer der beiden Suchvorgänge über eine einmalige Sortierung und Binärsuche optimiert werden. Optional könnten Sie den Einsatz einer zweiten internen Tabelle, die einen Sekundärindex (siehe Abschnitt 7.4.8) darstellt, in Erwägung ziehen.

Hinweis

Bitte beachten Sie, dass bei Zuweisungen in sortierte Tabellen auch implizite Sortiervorgänge notwendig sein können, wenn diese einen anderen Schlüssel als die Quelltabelle haben. Diese Sortiervorgänge sind im ABAP-Trace nicht direkt ersichtlich, da Zuweisungen keinen Events zugeordnet sind und nicht separat erfasst werden. Die benötigte Zeit für diese Sortiervorgänge wird den Nettozeiten der aufrufenden Modularisierungseinheit zugerechnet.

7.4.7 Kopierkostenreduzierter bzw. -freier Zugriff

Bei den Anweisungen `LOOP ... WHERE` und `READ` werden die Ergebnisse in den Arbeitsbereich *kopiert*. Bei der Anweisung `MODIFY` werden die Änderungen aus dem Arbeitsbereich in die Tabelle zurückkopiert.

Die Kopierkosten können beim `READ` und beim `MODIFY` nur auf die tatsächlich benötigten Felder eingeschränkt werden. Dazu muss der Zusatz `TRANSPORTING f1 f2 ...` angegeben werden. Es werden dann nur die Felder, die hinter dem Zusatz genannt werden, kopiert. Darüber hinaus können beim `LOOP ... WHERE` und `READ` die Kopierkosten ganz vermieden werden, wenn ein `TRANSPORTING NO FIELDS` angegeben wird. In diesem Fall werden nur die zugehörigen Systemfelder gefüllt, und es wird kein Ergebnis in die Kopfzeile oder den Arbeitsbereich kopiert. Dies wird dazu verwendet, um zu prüfen, ob sich ein bestimmter Eintrag in einer internen Tabelle befindet. Beim `LOOP ... WHERE` entspricht dieser Zugriff dann eher einem Lesezugriff.

Die Kopierkosten lassen sich auch vermeiden, wenn stattdessen nur die Referenz auf die Tabellenzeile in eine Referenzvariable kopiert wird bzw. die Speicheradresse einer Zeile einem Feldsymbol zugewiesen wird. Dies ist in Abbildung 7.7 schematisch dargestellt.

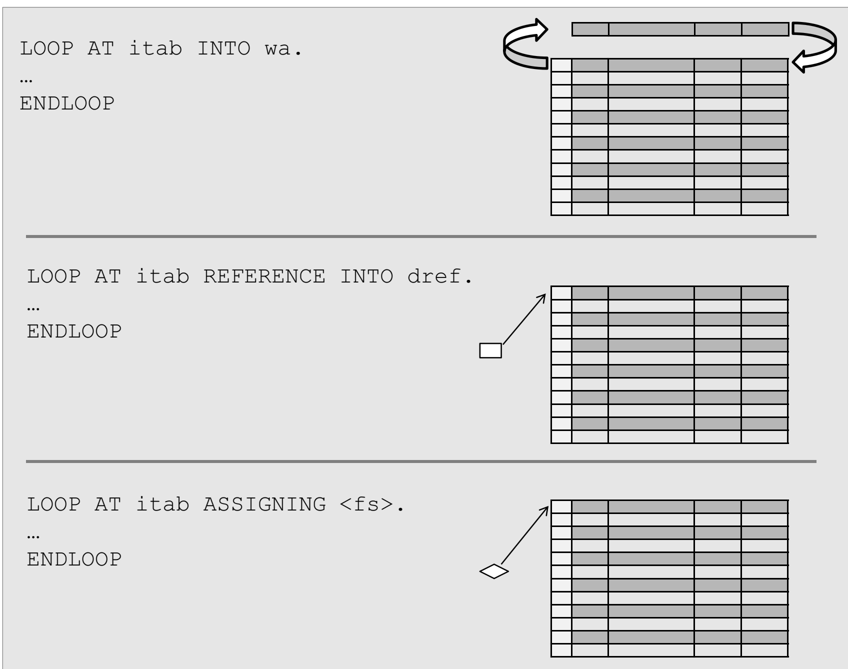


Abbildung 7.7 Schematische Darstellung der LOOP-Varianten

Die erste Variante `LOOP AT itab INTO wa` kopiert Zeile für Zeile der internen Tabelle `itab` in die Working Area `wa`. Falls die Zeile geändert werden soll, muss sie mit `MODIFY` (siehe Abschnitt 7.4.3) wieder zurückkopiert werden.

Die zweite Variante `LOOP AT itab REFERENCE INTO dref` stellt die Speicheradresse jeder Zeile – Zeile für Zeile – in die Datenreferenzvariable `dref`.

Die dritte Variante `LOOP AT itab ASSIGNING <fs>` weist dem Feldsymbol `<fs>` die Speicheradresse jeder Zeile zu, ebenfalls Zeile für Zeile.

Die Varianten zwei und drei sind durch den verringerten Kopieraufwand effizienter. Bei großen Datenmengen kann die Laufzeit durch diese Optionen reduziert werden. Bei sehr kleinen Datenmengen, wenn die interne Tabelle weniger als fünf Zeilen und keine übermäßig langen Zeilen (>5.000 Byte) hat, ist allerdings der normale Kopiervorgang schneller, da sowohl das Verwalten der Datenreferenzvariable als auch des Feldsymbols einen gewissen Overhead für das System bedeutet. Bei geschachtelten internen Tabellen (internen Tabellen, bei denen eine Spalte der Zeilenstruktur eine weitere Tabelle ist) lohnt sich ein Einsatz der kopierfreien Techniken immer. Auch wenn die Änderungen an der Zeile in die interne Tabelle zurückgeschrieben werden sollen, lohnt sich ein

Einsatz der kopierfreien Techniken, weil kein `MODIFY`-Befehl mehr benötigt wird.

Grundsätzlich gilt: Je größer die zu kopierende Datenmenge, desto lohnender ist sich ein Einsatz der kopierfreien Techniken.

Ein Zugriff auf *einen* Eintrag per `LOOP ... WHERE` oder `READ` lohnt sich bei breiten Zeilen (mehr als 1.000 Byte). Wenn die gelesene Zeile geändert und wieder in die Tabelle zurückgeschrieben werden soll (`MODIFY`), rentiert sich der kopierfreie Zugriff auch schon bei kürzeren Zeilen.

Praxisbeispiel – SE30, Tipps & Tricks:

In der Transaktion SE30 finden Sie bei den Tipps & Tricks unter Internal Tables • Using the Assigning Command • Modifying a set of lines directly ein Beispiel, dessen Laufzeit Sie vermessen können.

7.4.8 Sekundärindizes

Bis einschließlich Release 7.0 EhP1 können interne Tabellen keine Sekundärindizes haben. Daher werden, wenn effiziente Zugriffe über verschiedene Felder benötigt werden, oftmals Sekundärindizes in Form eigener interner Tabellen implementiert. Dabei wird eine zusätzliche interne Tabelle für jeden Sekundärschlüssel aufgebaut, die neben dem Feld, das den Sekundärschlüssel darstellt, eine Referenz auf die Haupttabelle enthält. Bei dieser Referenz kann es sich um die Position des Datensatzes in der Haupttabelle (nur bei Indextabellen) handeln oder um den Schlüssel in der Haupttabelle. Es kann aber auch eine eigene eindeutige Nummer dafür vorgesehen werden. Alle Lösungen bedeuten zusätzlichen Speicherbedarf, erlauben dafür aber einen effizienten Zugriff über mehrere Schlüsselfelder. Bei der Verarbeitung der internen Tabelle muss mit äußerster Genauigkeit darauf geachtet werden, dass die Sekundärindextabellen bei jeder Änderung der Haupttabelle mitgepflegt werden. Generell ist eine solche Vorgehensweise aufgrund ihrer Komplexität fehleranfällig und sollte nur in speziellen Situationen eingesetzt werden.

Praxisbeispiel – SE30, Tipps & Tricks:

In der Transaktion SE30 finden Sie bei den Tipps & Tricks unter Internal Tables • Secondary indices ein Beispiel, dessen Laufzeit Sie vermessen können.

Ab Release 7.0 EhP2 und 7.1 stehen Sekundärindizes zur Verfügung, die in Kapitel 10 beschrieben werden.

7.4.9 Kopieren

Ein weiterer Performanceaspekt, dessen man sich bewusst sein sollte, ist das so genannte *Tabellen-Sharing*. Bei Zuweisungen und Wertübergaben (Import und Export per Value) *gleichartiger* interner Tabellen, deren Zeilentypen selbst keine Tabellentypen enthalten, werden aus Performancegründen nur die internen Verwaltungsinformationen (Tabellenkopf) übergeben. Dies ist in Abbildung 7.8 schematisch dargestellt.

Hintergrund: Gleichartige interne Tabellen

Als gleichartige interne Tabellen werden Tabellen mit derselben Struktur bezeichnet. Das Tabellen-Sharing ist zwischen gleichartigen Tabellen möglich, wenn die Tabelle in die Zieltabelle den gleichen oder einen generischeren Typ hat wie die Quelltabelle. So funktionieren z. B. folgende Kombinationen:

`itab_standard = itab_sorted`

`itab_standard = itab_hashed`

`itab_sorted_with_nonunique_key = itab_sorted_with_unique_key`

Das Sharing funktioniert jeweils bei gleichem oder allgemeinerem Schlüssel der Zieltabelle (links vom =).

In folgenden Fällen funktioniert das Tabellen-Sharing nicht, weil die Zieltabelle nicht generischer ist als die Quelltabelle:

`itab_sorted = itab_standard` (bei gleicher Schlüsseldefinition)

`itab_sorted_with_unique_key = itab_sorted_with_nonunique_key` (bei gleicher Schlüsseldefinition)

Tabellen-Sharing ist mit beliebig vielen Tabellen möglich und kann vom ABAP-Entwickler nicht beeinflusst werden.

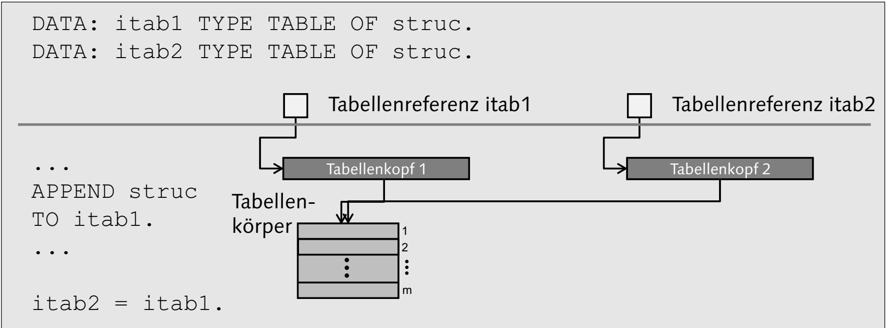


Abbildung 7.8 Tabellen-Sharing – Zuweisung

Das Tabellen-Sharing wird erst aufgehoben, wenn eine der am Sharing beteiligten internen Tabellen geändert wird. Erst dann findet der eigentliche Kopiervorgang statt. In Abbildung 7.9 ist der Zustand nach Aufhebung des Tabellen-Sharings schematisch dargestellt.

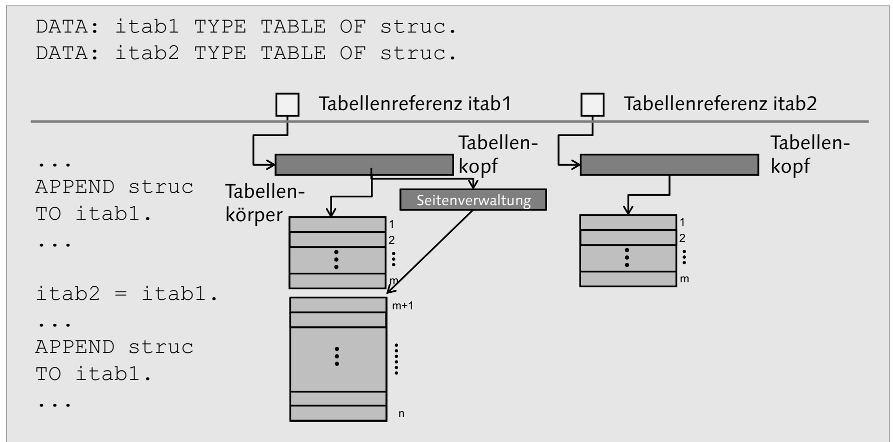


Abbildung 7.9 Aufgehobenes Tabellen-Sharing nach Änderung

Das Kopieren beim Aufheben des Tabellen-Sharings (auch *Copy on Write* oder *Lazy Copy* genannt) kann zu Situationen führen, die im ersten Moment »seltsam« aussehen:

So kann es vorkommen, dass nicht mehr genug Speicher zur Verfügung steht, wenn ein Eintrag einer internen Tabelle gelöscht werden soll, da das Tabellen-Sharing erst aufgehoben wird, wenn ändernd auf eine der beteiligten Tabellen zugegriffen wird. Dann findet der eigentliche Kopiervorgang statt. Wenn dann nicht genug Speicher für die Kopie zur Verfügung steht, werden Sie im Kurzdump die Information finden, dass bei der Ausführung der aktuellen Operation (DELETE) nicht genügend Speicher zur Verfügung stand.

Ein weiteres Beispiel ist, dass eine eigentlich schnelle Operation, beispielsweise eine APPEND-Anweisung, in der Laufzeitmessung mit einer deutlich höheren Zeit als vergleichbare Operationen auffällig werden kann. Dies kann auf das Aufheben des Tabellen-Sharings zurückzuführen sein.

Grundsätzlich sollten Sie sich also bewusst sein, dass jeder ändernde Zugriff auf eine interne Tabelle ein möglicherweise vorher bestehendes Tabellen-Sharing aufheben kann. Dies sind aber keine zusätzlichen, sondern lediglich aufgeschobene Kosten.

Zu den ändernden Zugriffen auf internen Tabellen gehören die Anweisungen APPEND, INSERT, MODIFY, DELETE, aber auch Zuweisungen auf Felder oder Zeilen der Tabelle, die über Datenreferenzen oder Feldsymbole gemacht werden. Auch ein DETACH bei Shared Objects führt zum Entsharen. Ebenso kann die Übergabe einer Tabelle per Value als Parameter einer Methode/Funktion/Form das Sharing aufheben, wenn der Parameter geändert wird.

Auch im Debugger oder im Memory Inspector wird das Tabellen-Sharing angezeigt. In Abbildung 7.10 sehen Sie unter den Speicherobjekten die jeweiligen Tabellenköpfe, die auf das Speicherobjekt zeigen. So sind in diesem Beispiel die internen Tabellen ITAB2A und ITAB1 gesharet.

Speicheranalyse
Speicherobjekte
Sicht: 1 Speicherobjekte — Einträge: 25

	Ra...	gebunden	Bele...	refer...
ABAP: 147072.672 bytes				
Speicherobjekte				
[293667x31(158)]	1	52.784.944	95%	52.784.!
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB				
[293667x31(158)]	2	47.733.552	100%	47.733.!
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB2				
[293667x31(158)]	3	46.491.264	100%	46.491.:
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB2A				
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB1				
[1x15(240)]	4	1.348	28%	1.:

Abbildung 7.10 Tabellen-Sharing im Debugger

Im Memory Inspector (siehe Abbildung 7.11) sieht man bereits neben den Tabellenrümpfen den Namen der jeweiligen internen Tabelle. Tabellenrümpfe ohne Namen (z. B. der zweite in Abbildung 7.11) weisen auf Tabellen hin, die gesharet sind. Auch in diesem Fall sind dies die Tabellen ITAB2A und ITAB1.

Speicherabzug (t_0) — Sicht Übersicht — Anzeigebegrenzung Programme 100

Speicherobjekt	Ra...	Wert 1	Wert 2	Wert 3	Wert 4
Rollbereich		ABAP_TOTAL	MM_TOTAL	delta MM_TOTAL (GC)	
·		147.073.084	147.796.728	0	
·		Anzahl Programme	Anzahl Klassen	Anzahl Instanzen	Anzahl Tabellen
·		29	3	0	5
29 Programme		Globale Daten	Anzahl Instanzen	Anzahl Tabellen	Anzahl Strings
3 Klassen (ABAP OO)		Anzahl Instanzen	Gebunden (allok.)	Gebunden (benutzt)	Referenziert (allok.)
4 Tabellenrümpfe		Zeilen (allokiert)	Zeilen (benutzt)	Belegung Zeilen (%)	Gebunden (allok.)
Summe		881.019	881.002		147.011.108
[293667x158] : ITAB	1	293.680	293.667	100	52.784.944
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB	1				
[293667x158]	2	293.667	293.667	100	46.491.264
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB2A	1				
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB1	2				
[293667x158] : ITAB2	3	293.667	293.667	100	47.733.552
\PROGRAM=Z_GAHM_TEMP\DATA=ITAB2	1				
[1x240] : SCREEN_PROGS[]	4	5	1	20	1.348
1 Strings		Gebunden (allok.)	Gebunden (benutzt)	RefCount	

Abbildung 7.11 Tabellen-Sharing im Memory Inspector

7.4.10 Geschachtelte Schleifen und nicht-lineares Laufzeitverhalten

Ineffiziente Zugriffe auf interne Tabellen wirken sich besonders drastisch bei großen Datenmengen aus. Das folgende kleine Beispiel zeigt eine geschachtelte Schleife, bei der die jeweiligen Aufträge von Kunden verarbeitet werden:

```
LOOP AT it_customers REFERENCE INTO dref_customer.
  LOOP AT it_orders REFERENCE INTO dref_order
  WHERE cnr = dref_customer->nr.
...
  ENDLOOP.
ENDLOOP.
```

Nehmen wir an, die interne Tabelle `it_customers` hat 1.000 Einträge. Für jeden Kunden existieren im Mittel zwei Aufträge, die interne Tabelle `it_orders` hat also 2.000 Einträge. Wenn es sich jeweils um Standardtabellen handelt, müssen beide internen Tabellen vollständig abgearbeitet werden: Die äußere Tabelle `it_customers`, weil keine Einschränkung vorhanden ist und semantisch alle Datensätze verarbeitet werden sollen; die innere Tabelle `it_orders` wird zwar eingeschränkt, die zugehörigen Einträge für jeden Kunden können aber nicht effizient gesucht werden, es muss also auch für die innere Tabelle die ganze Tabelle `it_orders` durchsucht werden. Dies geschieht für jeden Eintrag der äußeren Tabelle in unserem Beispiel also 1.000 Mal.

Nehmen wir an, der äußere Loop benötigt ungefähr 200 µs und der innere insgesamt 140.000 µs. Wenn die Datenmengen nun verdoppelt werden, verdoppeln sich auch die Laufzeiten für *jeden* Loop, da die beiden Loops linear mit der Anzahl der Einträge skalieren. Es ergäben sich also bei 2.000 Einträgen in der äußeren Tabelle (`it_customers`) ~400 µs und bei der inneren Tabelle (`it_orders`) für 4.000 Einträge ~560.000 µs für alle 2.000 Durchläufe. Zwar muss die innere Tabelle für jeden Eintrag der äußeren durchlaufen werden, dabei müssen aber nicht für jeden äußeren Eintrag alle inneren, sondern nur die zwei Einträge, die zu einem Kunden gehören, verarbeitet werden.

Die Laufzeit ist bei doppelter Datenmenge also viermal so lang. Das Laufzeitverhalten ist nicht linear, sondern quadratisch. (Der innere Loop dauert doppelt so lange wie zuvor – skaliert mit n – und wird zweimal so häufig wie zuvor ausgeführt.)

In diesem Fall ist der Grund also ein ineffizienter Zugriff auf die innere Tabelle. Um dies zu vermeiden, muss der Zugriff auf die innere Tabelle optimiert werden. Für ein lineares Laufzeitverhalten müsste der Zugriff auf die innere Tabelle konstant sein, damit sich bei einer Verdoppelung der Zugriffshäufigkeit die Laufzeit verdoppelt. Da im vorherigen Beispiel kein eindeutiger Schlüssel möglich ist, kann über eine Sorted-Tabelle ein logarithmisches Laufzeitverhalten für den inneren Zugriff erreicht werden. Die Sorted-Tabelle erlaubt eine Binärsuche auf der inneren Tabelle und sorgt so für ein effizientes Auffinden der beiden passenden Einträge in der inneren Tabelle für jeden Eintrag der äußeren Tabelle. Für das gesamte Codefragment ergibt sich dann `O(n x log n)`.

An dieser Stelle ein kurzer Vergleich zum Nested Loop Join bei Datenbanken (siehe Abschnitt 5.4.5): Wie auch bei den Nested Loop Joins auf Datenbanken sind für die Optimierung geschachtelter Schleifen die Anzahl und die Effizienz des Zugriffs auf die innere Tabelle von Bedeutung.

Nicht-lineares Laufzeitverhalten liegt nicht immer an ineffizienten Zugriffen auf interne Tabellen, sondern kann z. B. auch durch einen quadratischen Anstieg der Aufrufhäufigkeit eines effizienten Zugriffs auf eine interne Tabelle hervorgerufen werden.

Die Auswirkungen nicht-linearer Programmierung können generell mit kleineren Datenpaketen reduziert werden. Allerdings sollten die Pakete dabei nicht zu klein gewählt werden, um nicht zu großen Overhead an anderen Stellen zu erzeugen (siehe auch Abschnitt 4.1.3).

Da bei der Entwicklung von Programmen auf dem Entwicklungssystem meistens nur ein kleiner Testdatenbestand zur Verfügung steht, kommt es vor, dass nicht-lineares Laufzeitverhalten nur schwer zu entdecken ist, da geschachtelte Schleifen mit kleinen Datenmengen oft nur einen kleineren Teil der gesamten Programmlaufzeit ausmachen. Bei kleinen Testdatenbeständen sieht es dann oft so aus, als würde sich das Programm linear zur Anzahl der verarbeiteten Mengen verhalten.

Um bereits während der Entwicklung mit kleinen Datenmengen nicht-lineares Laufzeitverhalten zu entdecken, muss das Laufzeitverhalten auf ABAP-Statement-Ebene verglichen werden. Dabei werden die Zeiten für die Zugriffe auf interne Tabellen mit zwei Varianten – z. B. einmal mit zehn und einmal mit 100 zu verarbeitenden Datensätzen – mit den Transaktionen SE30 oder ST12 vermessen und miteinander verglichen. So kann schon mit kleinen Datenmengen nicht-lineares Laufzeitverhalten entdeckt werden.

Im Release 7.0 EhP1 gibt es kein Werkzeug, mit dem Sie diesen Vergleich automatisiert durchführen können. Sie finden jedoch im SDN unter folgenden Links ein Werkzeug und eine Beschreibung, wie sich so ein Vergleich automatisieren lässt:

- Nonlinearity: The problem and background
  https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/5804
- A Tool to Compare Runtime Measurements: Z_SE30_COMPARE:
  https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/8277
- Report Z_SE30_COMPARE:
  https://www.sdn.sap.com/irj/sdn/wiki?path=/display/Snippets/Report%2bZ_SE30_COMPARE

- Nonlinearity Check Using the Z_SE30_COMPARE:
https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/8367

Im Release 7.00 EhP2 kann der Vergleich von Trace-Dateien mit SAP-Standardmitteln durchgeführt werden (siehe Kapitel 10).

7.4.11 Zusammenfassung

Bei der Arbeit mit internen Tabellen ist die Wahl des richtigen Tabellentyps und der Zugriffsart entscheidend.

Standardtabellen sollten nur für kleine Tabellen oder Tabellen, die mit Indexzugriffen verarbeitet werden können, verwendet werden. Bei größeren Standardtabellen sollten Sie auf jeden Fall auf eine effiziente Verarbeitung mit der Binärsuche achten, wenn nur Teile der Tabelle verarbeitet werden sollen. Diese kann sowohl für Einzelsatz- als auch für Massenzugriffe verwendet werden. Standardtabellen sollten dafür nach Möglichkeit gleich sortiert aufgebaut werden oder aber nur so häufig sortiert werden, wie dies unbedingt erforderlich ist. Bei Zugriffen auf verschiedene Schlüsselfelder mit der Binärsuche, die ein Umsortieren notwendig machen würden, muss in jedem Fall geklärt werden, ob der Aufwand für das Sortieren gerechtfertigt ist (von den verbesserten Lesezugriffen amortisiert wird).

Sorted-Tabellen können eindeutig oder nicht-eindeutig für die meisten Anwendungsfälle benutzt werden. Sie bieten sich besonders für die teilsequenzielle Verarbeitung an, bei der z. B. ein Anfangsstück des Tabellenschlüssels benutzt wird.

Hash-Tabellen sollten nur da verwendet werden, wo der eindeutige Schlüsselzugriff die einzige Art des Zugriffs ist, insbesondere dann, wenn sehr große Tabellen verarbeitet werden müssen.

Generell sollten interne Tabellen, wenn es möglich ist, mit Block-Operationen gefüllt werden, um den Overhead von Einzelsatzzugriffen zu vermeiden.

Wo es sinnvoll ist, sollten Kopierkosten für die Bereitstellung der Ergebnisse mit `TRANSPORTING fieldlist/NO FIELDS` reduziert oder mit `ASSIGNING` bzw. `REFERENCE INTO` ganz vermieden werden. Dies ist besonders für geschachtelte Tabellen wichtig.

Interne Tabellen sollten nach Möglichkeit generell nicht zu groß werden, um den Speicherplatz des SAP-Systems zu schonen.

Index

/SDF/E2E_TRACE 105, 108

A

ABAP Array Interface 220
ABAP Central Services (ASCS) 24
ABAP Database Connectivity (ADBC) 161, 233
ABAP Dump 126
ABAP Load 23
ABAP Memory 243, 249
ABAP Paging Area 249
ABAP Virtual Machine 79
ABAP Workbench 254
ABAP-Debugger 31
ABAP-Stack 23, 24
ABAP-Trace 30, 79, 330
ABAP-Tuning 17
abgebrochene Pakete 145
Active Key Protection 339
Advanced List Viewer (ALV) 322, 326
Aggregat 207
Aggregated Table Summary 70
Aggregatfunktion 270
Allokation 278
Antwortzeit 17
Anwendungsstatistik 125
Anwendungs-Tuning 18
APPEND SORTED BY 279
Applikationsschicht 21, 23, 26
Architektur, Performanceaspekte 25
Array Interface 133, 221, 323
asynchrone Verbuchung 317
asynchroner RFC 148, 311
Aufrufhierarchie 90
Aufrufstelle des SQL-Statements 222
ausführliche Trace-Liste 61
Ausführungshäufigkeit 219
Ausführungsplan 68, 162, 343

B

Background RFC (bgRFC) 312
Batchjob 147
Batchjob-API 148
Batchservergruppe 148
baumartiger Index 283
benutzerbezogener Speicher 243
Benutzersitzung 244
benutzerübergreifender Speicher 244
Binary Search 294
Blatt 283
Blockgröße 158
Bottom-up-Analyse 100
Buffer Pool 157
BYPASSING BUFFER 270

C

Calendar-Puffer 251
CALL FUNCTION ... DESTINATION... 244
Call Stack 222, 327
Call Tree 102
CLIENT SPECIFIED 183, 219, 271
Client-Server-Architektur 21
Clustered Index Scan 182
Clustered Index Seek 181, 182
Code Inspector 32
  Performanceprüfungen 36
COLLECT 298
COMMIT WORK 61, 70, 104, 140, 163, 231, 317
Copy on Write 304
Cost-based Optimizer 179
COUNT 270
Covering Index 172, 200, 219
CUA-Puffer 251
Cursor-Cache 156

D

Data Manipulation Language (DML) 140, 214
Data Sharing 243
Datenbank 27
  Aggregate 207
  API für Datenbankabfragen 232
  Ausführungspläne 181
  Blöcke 157

Datenbank (Forts.)
Datenbank Hints 238
Datenbank-Interface 160
Datenbankprozess 156
Datenbank-Thread 156
Daten-Cache 157
DBI-Hints 237
Ergebnismenge 197
Existenzchecks 209
Explain Plan 181, 343
FOR ALL ENTRIES 227
Full Table Scan 174
geschachtelte SELECTs 223
Hash Join 194
Hauptspeicher 156
Heap-Tabellen 165
Identical SELECTs 223
Index Fast Full Scan 174
Index Full Scan 173
Index Range Scan 170
Index Unique Scan 169
Indexdesign 211
indexorganisierte Tabelle 165
Indizes als Suchhilfe 167
Join 191, 226
Join-Methoden 191
Kompilieren 162
logische Strukturen 165
NATIVE SQL 233
Nested Loop Join 192
OPEN SQL 232
Operatoren 177
Optimizer 179
Paketgrößen 199
Parsen 162
passender Zugriffspfad 211
physikalischer I/O 158
Pool- und Cluster-Tabellen 235
Selektivität und Verteilung 185
Softwarearchitektur 155
Sort Merge Join 193
Sortieren 234
SQL-Cache 157
Statistiken 180
Systemstatistiken 180
unpassender Zugriffspfad 196
Views 224
zentrale Ressource 155
Zugriffsstrategien 164
Datenbank-Interface 160
Datenbankprozess 156
Datenbankschicht 21, 23
Datenbanksperre 140
Datenblock 157
Daten-Cache 157
DB File Scattered Read 176
DB File Sequential Read 176
DB02 195, 196
DB05 31, 32, 33, 34, 41, 190, 272
Ergebnisbildschirm 42
DB2 for iSeries 156
DBI Array Interface 220
DBI-Hint 237
Deadlock 141
Deallokation 279
Debugger 48
Speicherabzug 50
Speicheranalyse-Werkzeug 49
Default Key 281
Delayed Index Update 340
DELETE 297
DELETE FROM SHARED BUFFER 251
DELETE FROM SHARED MEMORY 252
Dequeue-Baustein 139
DISTINCT 270
Double Stack 23, 24
Dreischichtenarchitektur 21, 22
Durchsatz 17, 136
dynamische Verteilung 145

E

E2E-Trace 105
Analyse 109
Durchführung eines Traces 107
Voraussetzungen 105
Einzelsatzoperation 289
Einzelsatzpufferung 259
Einzelsatzstatistik 30
Einzelsatzzugriff 134
End-to-End-Trace 29, 31, 105
Enqueue-Service 26, 323
Enqueue-Trace 74
Ergebnismenge 197
erweiterte Schreibsperre 323
Event 79
Existenzcheck 209
Explain Plan 181, 190

EXPORT TO MEMORY 249
EXPORT TO SHARED BUFFER 251
EXPORT TO SHARED MEMORY 252
Extended Memory 242
Extended Trace List 61
externer Modus 245

F

Fehlerbehandlung, Paketverarbeitung 134
Filesystem-Cache 158
Filterwerkzeug 334
FLUSH_ENQUEUE 323
FOR ALL ENTRIES 191, 227, 228, 238, 271
FOR UPDATE 270
Fragmentierung 262
Frontend 26
Frontendressource 322
Full Table Scan 174, 177, 182, 344, 348, 351, 354, 356, 359

G

Gebiet 254
gebündelter Zugriff 134
generische Pufferung 259
geschachtelte Schleifen 36, 305
geschachtelte SELECT-Anweisung 191, 223
geschachtelte Tabellen 37
GET PARAMETER ID 250
GROUP BY 270

H

Hash Join 191, 194
Hash-Tabelle 194, 282, 285, 286
Hash-Verwaltung 285
Hauptmodus 245
Heap-Speicher 243
Heap-Tabelle 165
Hints 237
horizontale Verteilung 25
HTTP 309
HTTP-Plug-in 107
HTTP-Trace 330

I

IBM DB2 iSeries 347
IBM DB2 UDB 350
IBM DB2 zSeries 344
Identical Selects 68
IMPORT FROM SHARED BUFFER 251
IMPORT FROM SHARED MEMORY 252
Index Fast Full Scan 174
Index Full Scan 173, 346, 349, 352, 355, 358, 361
Index Only 169
Index Range Scan 170, 177, 182, 345, 348, 351, 354, 357, 360
Index Skip Scan 177
Index Unique Scan 169, 177, 181, 346, 349, 352, 355, 357, 360
Indexdesign 211
indexorganisierte Tabellen 165
Indexstatistik 180
Indexstruktur 167
Indextabelle 282, 321
Indizes als Suchhilfe 167
INITIAL SIZE 278
Inner Join 226
Inspektion 35
Inter Process Communication (IPC) 156
interne Tabellen 275, 276
APPEND 288
baumartiger Index 283
Beschränkungen 286
Binary Search 294
COLLECT 298
DELETE 297
füllen 287
geschachtelte Schleifen 305
gleichartige 303
Hash-Tabellen 282
Hash-Verwaltung 285
Index 282
Indextabellen 282
INITIAL SIZE 278
INSERT 288
kopieren 303
Kopierkosten 300
Lesen 291
linearer Index 283
LOOP 291
MODIFY 296

interne Tabellen (Forts.)
Organisation im Hauptspeicher 277
Performanceaspekte 287
READ TABLE 293
Sekundärindizes 302
Sekundärschlüssel 336
SORT 299
Sorted-Tabellen 281
Standardtabellen 281
Tabellen-Sharing 303
Tabellentypen 281
interner Modus 245
IS NULL 271

J

Java-Stack 23, 24
Jobzustandsabfrage 148
Join 191, 226, 347, 350, 353, 355, 358, 361
Join-Methode 191, 194
Journal 163

K

Kiwi Approach 18
Kommunikation
direkte Kommunikation 309
indirekte Kommunikation 309
Protokolle 309
kompilieren 157, 162
Konsistenzcheck 129
Kopierkosten 264

L

Lastverteilung 146
Latenzzeit 322
Laufzeitverhalten 130
Lazy Copy 304
Lazy Index Update 338
Least Frequently Used 163, 261
Least Recently Used 162, 252, 261
Left Outer Join 226
lesende Verarbeitung 214
lesende Zugriffe 162
linearer Index 283
Lock-Eskalation 141
Logical Row ID 168
logische Strukturen 165
lokale Verbuchung 317, 318
LOOP 291

M

Mapping Area 242
Massendaten 130
Memory Inspector 32, 49, 50, 305
Speicherabzüge erstellen 50
Memory Snapshot 49
Mengenoperation 287
Merge Scan Join 191
Message-Service 26, 323
Messdatenübersicht 335
Microsoft SQL Server 359
MODIFY 296
Modularisierungseinheit 99
Multiblock I/O 175

N

Nametab-Puffer 243, 251
Native SQL 233
dynamisch 233
Nested Loop Join 191, 192
Netzwerkpaketgröße 199

O

Objektmenge 35
Open SQL 232
dynamisch 233
Operator 177
Optimierung 130
Optimizer 179
Oracle 356
ORDER BY 234, 270
OTR-Puffer 251

P

Package Size 220
Paketgröße 133, 142, 199
Paketierung 133
Paketverarbeitung 133
Array Interface 133
Fehlerbehandlung 134
Paketgröße 133

parallele Verarbeitung 136
Parallelisierung 133, 135
abgebrochene Pakete 145
asynchroner RFC 148
Batchjob 147
Batchservergruppe 148
Deadlock 141
Dynamische Verteilung 145
gleichmäßige Ausnutzung der Hardware 146
Herausforderungen 137
Kapazitätsgrenzen der Hardware 147
Lastverteilung 146
Paketgröße 142
Parallelisierungskriterium 136
Sperre 138
statische Verteilung 144
Status der Verarbeitung 145
Synchronisationszeitpunkt 137
Techniken zur Parallelisierung 147
Verteilung der Pakete 144
Wiederaufsetzbarkeit 145
Parameter Memory 250
Parameterübergaben 320
parsen 157, 162
passender Zugriffspfad 211
Passport 105
Performance 17
ABAP-Tuning 17
Antwortzeit 17
Anwendungs-Tuning 18
Durchsatz 17
Hardware 18
Skalierbarkeit 17
System-Tuning 18
Performanceanalyse 29, 325
Performancemanagement 18
Performance-Trace 30, 55, 72, 325
aktivieren 55
anzeigen 56
deaktivieren 56
sichern 57
PERFORMING ... ON END OF TASK 149
physikalischer I/O 158
Pool- und Cluster-Tabellen 235
Post-Runtime-Analyse 114
Präsentationsschicht 21, 22
PRIV-Mode 243
Profilwerkzeug 334
Programmpuffer 243, 251
Prozessanalyse 45
globale Prozessübersicht 47
lokale Prozessübersicht 47
Prozessdetails 47
Status 45
Prozesskette 143
Prozessmonitor 115
Prüfvariante 35, 39
Puffer 243
Pufferschlüssel 269
Puffer-Trace 272
Pufferung 241
im ABAP Memory 249
im internen Modus 245
im SAP Memory/Parameter Memory 250
im Shared Buffer 251
im Shared Memory 252
im Tabellenpuffer 256
in internen Tabellen 247
in Shared Objects 253
in Variablen 246
Wiederverwendbarkeit 248
Pufferungsstatus 55

Q

queued RFC (qRFC) 152, 311
Quota 242

R

Random I/O 175
Raw Device 158
Read Ahead 174
READ TABLE 293
Request Tree 112
RETURNING-Parameter 320
RFC 309, 310
Datentransfer 314
Roundtrips 314
RFC-Kommunikation 310
RFC-Overhead 313
RFC-Servergruppe 149
Konfiguration 152
Ressourcenparameter 152
RFC-Trace 72
Roundtrip 322

RSPRFC02 149
Rule-based Optimizer 179
Run Time Type Identification 282
RZ12 149

S

S_MEMORY_INSPECTOR 31, 50
SAP Business Explorer 22
SAP Code Inspector 33, 34, 35, 270
  Ad-hoc-Inspektion 37
  Ergebnisbildschirm 39
  Grenzen 37
  SQL-Trace 329
  Tests 35
SAP EarlyWatch Check 195
SAP GoingLive Check 195
SAP GUI 22
SAP MaxDB 353
SAP Memory 243, 250
SAP NetWeaver Application Server 23
SAP NetWeaver Business Client 22
SAP NetWeaver Portal 23
SAP-Enqueue 139
SAPHTTPPlugIn.exe 107
SAP-Systemarchitektur 21
SAP-Tabellenpuffer 245
SAT 330
  Filterwerkzeug 334
  Messdatenübersicht 335
  Profilwerkzeug 334
  Zeitwerkzeug 334
schreibende Verarbeitung 214
schreibende Zugriffe 163
SCI 31, 32, 33, 34, 35
SCII 36, 37
Screen-Puffer 251
SE11 224, 237
SE12 237
SE16 190, 195, 272
SE17 272
SE24 36
SE30 30, 47, 79, 185, 201, 204, 209, 210, 223, 226, 230, 288, 294, 299, 302, 307, 321, 328
  Aggregation, ohne 84
  Aggregation, pro Aufrufstelle 84
  Aggregation, vollständig 84
  Anweisungen 83
SE30 (Forts.)
  Aufrufhierarchie 90
  Brutto- und Nettozeit 88
  Dauer und Art 84
  im aktuellen Modus 85
  im parallelen Modus 85
  Messvariante definieren 82
  Programmteile 83
  Trace auswerten 86
  Trace erstellen 85
  Trace-Dateien verwalten 92
SE37 36
SE38 36
SE80 254
Seite 277
Sekundärindex 168, 302
  eindeutiger 168
Sekundärschlüssel 336
SELECT – Code-Inspector-Prüfungen 36
SELECT in Schleifen 222
Selektivität 185
Selektivitätsanalyse 33, 34, 41
Sequential I/O 175
sequenzielle Verarbeitung 136
SET PARAMETER ID 250
Shared Buffer 243, 251
Shared Memory 243, 252
Shared Objects 243, 244, 253
SHMA 254
SHMM 254
Single Block I/O 175
Single Statistical Record 111
Single Transaction Analysis 92
Skalierbarkeit 17
SM37 41
SM50 31, 45, 47, 115, 181, 182, 249
SM61 148
SM66 31, 45, 47
SMD 105
SMTP 310
Solution Manager Diagnostics 105
SORT 299
Sort Merge Join 191, 193
Sorted-Tabelle 281, 286
Sortieren 234
Spalten reduzieren 198, 200
Spaltenstatistik 180
Speicherabzug 50

Speicherarchitektur 241
benutzerbezogener Speicher 243
benutzerübergreifender Speicher 244
Speicherbereiche 241
Speicherbereich 241
Sperre 138
SQL 155
Ausführung 160
effizientes 164
SQL Statement Summary 62
SQL-Cache 156, 157
SQL-Trace 30, 58, 72, 183, 185, 195, 201
Aggregated Table Summary 70
Aufrufstelle im ABAP-Programm 68
Datenbank-Interface 58
DDIC-Informationen 68
Details des ausgewählten Statements 66
EXPLAIN 68
Identical SELECTs 68
Statement Summary 62
Table Summary 70
Trace-Liste 60
SSR 111
ST02 243, 252, 253, 268
ST04 46, 176
ST05 30, 46, 55, 181, 183, 185, 195, 197, 201, 211, 223, 237, 267, 272, 325
Enqueue-Trace 74
HTTP-Trace 330
Performance-Trace 72
RFC-Trace 72
SQL-Trace 58
Stack-Trace 327
Tabelllenpuffer-Trace 76
ST10 31, 52, 263, 267, 272
Status der gepufferten Tabellen 53
ST12 30, 47, 92, 93, 223, 307, 328
Bottom-up-Analyse 100
gruppiert nach Modularisierungseinheiten 98
SQL-Trace 104
Top-down-Analyse 102
Trace auswerten 97
Trace erstellen 94
Traces einsammeln 96
Übersicht 93
ST22 31, 125
Stack-Trace 327
STAD 30, 31, 32, 34, 41, 114
Auswertung 117
Selektion 116
Standalone-Enqueue-Server 24
Standardtabelle 281, 286
statische Verteilung 144
Statistiksatz 114
Swap 268
synchrone Verbuchung 318
synchroner RFC 310
Synchronisationszeitpunkt 137
Systemstatistik 180
System-Tuning 18

T

Tabellenpuffer 241, 243
Analysemöglichkeiten 272
Architektur 257
Einzelsatzpufferung 260
generische Pufferung 260
Größe der Tabellen 262
Kriterien zur Pufferung 263
lesender Zugriff 258
Performanceaspekte 264
schreibender Zugriff 258
SQL-Statements, die am Puffer vorbeigehen 269
Synchronisation 259
Typen 259
Verdrängung und Invalidierung 268
vollständige Pufferung 260
Zugriffe, die am Puffer vorbeigehen 267
Tabellenpuffer-Trace 76
Tabellenpufferung 256
Tabellen-Sharing 303
Tabellenstatistik 180
Tabellentyp 281
Tabellenübersicht 71
Tabellenzugriffsstatistik 54
Table Body 277
Table Header 277
Table Reference 277
Table Summary 70
TABLES-Parameter 320
Time Split Hierarchy 103
Top Down Time Split Hierarchy 102
Top-down-Analyse 102
Trace-Level 106, 107

Trace-Liste 60
Traces 34
Transaction Log 163
transaktionaler RFC (tRFC) 311
Typkonvertierung 321

U

Unicode 262
unpassender Zugriffspfad 196
Unterabfrage 270
UP TO n ROWS 206
Update 210
UPDATE dbtab FROM... 215
UPDATE SET... 215
User Session 244

V

varchar 262
Verbuchung
  asynchrone 317
  lokale 317, 318
  synchrone 318
Verbuchungstabelle 317
Verdichten 298
verdichtete Tabellenübersicht 70
Vermeidung 129
Verteilung 185
vertikale Verteilung 25
View 224
vollständige Pufferung 259

W

Web Dynpro Java 23
Werkzeuge 29
  ABAP-Trace 79, 330
  Debugger 48
  Dump-Analyse 125
  E2E-Trace 105
  Einsatzzeitpunkte 32
  Einzelsatzstatistik 114
  Enqueue-Trace 74
  Memory Inspector 50
  Performance-Trace 55, 325
  Prozessanalyse 45
  RFC-Trace 72
  SAP Code Inspector – SCI 35
  Selektivitätsanalyse 41
  Single Transaction Analysis 93
  SQL-Trace 58
  Tabellenpuffer-Trace 76
  Table Call Statistics 52
  Traces 34
  Übersicht 30
WHERE 36
Wiederaufsetzbarkeit 145
Workprozess-Monitor 32

Z

Zeilen reduzieren 198, 203
zeitbasierte Analyse 129
Zeitwerkzeug 334
zentrale Ressourcen 26
Zugriffspfad 179
Zugriffsstatistiken 52