Deutsch

FUJITSU Software BS2000

SESAM/SQL-Server V9.0

Performance

Benutzerhandbuch

Ausgabe Oktober 2016

Kritik… Anregungen… Korrekturen…

Die Redaktion ist interessiert an Ihren Kommentaren zu diesem Handbuch. Ihre Rückmeldungen helfen uns, die Dokumentation zu optimieren und auf Ihre Wünsche und Bedürfnisse abzustimmen.

Sie können uns Ihre Kommentare per E-Mail an manuals@ts.fujitsu.com senden.

Zertifizierte Dokumentation nach DIN EN ISO 9001:2008

Um eine gleichbleibend hohe Qualität und Anwenderfreundlichkeit zu gewährleisten, wurde diese Dokumentation nach den Vorgaben eines Qualitätsmanagementsystems erstellt, welches die Forderungen der DIN EN ISO 9001:2008 erfüllt.

cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Copyright und Handelsmarken

Inhalt

1 Einleitung

Das Datenbanksystem SESAM/SQL-Server erfüllt durch seine Funktionen und seine Architekturmerkmale alle Anforderungen, die heute an einen leistungsfähigen Datenbankserver gestellt werden. Diese Eigenschaft drückt sich auch im Produktnamen SESAM/SQL-Server aus.

SESAM/SQL-Server gibt es als Standard Edition mit Singletask-Betrieb und als Enterprise Edition, die den Multitask-Betrieb beinhaltet.

Der Einfachheit halber ist im Folgenden von SESAM/SQL die Rede, wenn das Datenbanksystem SESAM/SQL-Server gemeint ist.

Folgende einleitenden Beschreibungen befinden sich zentral im „Basishandbuch“:

- Kurzbeschreibung des Produkts
- Konzept der SESAM/SQL-Server-Dokumentation
- Beispieldatenbank
- Readme-Datei
- Änderungen gegenüber den Vorgänger-Handbüchern

1.1 Zielsetzung und Zielgruppen des Handbuchs

Das vorliegende Handbuch richtet sich an den erfahrenen Anwender von SESAM/SQL.

Es beschreibt, wie der Anwender Performance-Engpässe im Leistungsverhalten von SESAM/SQL erkennen, und mit welchen Maßnahmen er das Systemverhalten beeinflussen kann.

1.2 Konzept des Handbuchs

Das vorliegende Handbuch soll Hinweise dazu geben, wie der Anwender die Performance von SESAM/SQL beeinflussen kann:

- Wie kann der Anwender das Vorliegen eines Performance-Problems erkennen, und wie kann er ein Performance-Problem analysieren?
- Welche Eigenschaften des SESAM/SQL-Gesamtsystems können geändert werden?
- Wie kann der Anwender ein Performance-Problem beseitigen?

Im Einzelnen behandelt das Handbuch folgende Themen:

- Anschließend an diese Einleitung beschreibt Kapitel 2, welche Mess- und Analyse-Tools zum Tuning bzw. zur Engpassanalyse zur Verfügung stehen.
- Kapitel 3 beschreibt performance-relevante Aspekte von Anwendungen: Das Vorgehen des Optimizers bei der Erzeugung von SQL-Zugriffsplänen, Synchronisationsmechanismen und Ressourcenverbrauch.
  Das Verständnis der Vorgehensweise des Optimizers ist eine Voraussetzung für Anwender, die mit Hilfe von Pragmas und zusätzlichen Indizes ihre Anwendung beschleunigen wollen.
- Kapitel 4 behandelt performance-relevante Aspekte, die schon beim Datenbankentwurf berücksichtigt werden können: Wie wird der Plattenspeicherbedarf für Primär-, Sekundär- und Metadaten ermittelt? Welche Auswirkungen auf die Performance haben die Prüfung von Integritätsbedingungen und die Verteilung der Daten?
- Kapitel 5 behandelt performance-relevante Aspekte des Datenbankbetriebs.
- Kapitel 6 enthält gezielte Hinweise zum Tuning von SQL- und CALL-DML-Anwendungen.
- Kapitel 7 erläutert die Ausgabe der Erklärungskomponente des Optimizers.
- Kapitel 8 behandelt performance-relevante Aspekte von Utility-Funktionen und DDL-Anweisungen.
- Kapitel 9 behandelt performance-relevante Aspekte von Administrationsanweisungen.

Schnittstellen, Abläufe und Berechnungen, die ausschließlich in diesem Handbuch beschrieben sind, beziehen sich auf die aktuelle Version von SESAM/SQL-Server und können sich in Folgeversionen ändern.

1.3 Darstellungsmittel

In diesem Handbuch verwenden wir folgende Darstellungsmittel:

GROSS	SQL-Schlüsselwörter
unterstrichen	Voreinstellungen
fett	Hervorhebung im Fließtext
kursiv	Parameter in Syntaxdefinitionen und im Fließtext
`dicktengleich`	Programmtext in Beispielen
,...	In Syntaxdefinitionen bedeutet diese Schreibweise, dass die vorausgehende Angabe beliebig oft, durch Komma getrennt, wiederholt werden kann. Wird keine Wiederholung angegeben, fällt auch das Komma weg.
...	In Syntaxdefinitionen bedeuten die Punkte, dass die vorausgehende Angabe beliebig oft wiederholt werden kann. In Beispielen bedeuten die Punkte, dass die restlichen Teile für das Beispiel ohne Bedeutung sind. Die Punkte sind Metazeichen, die in einer SQL-Anweisung nicht angegeben werden.
⇒	Die Ausdrücke oder Operationen auf der rechten Seite des Zeichens bzw. über dem Zeichen resultieren in derselben Ergebnistabelle wie die Ausdrücke oder Operationen auf der linken Seite des Zeichens bzw. unter dem Zeichen.
	Hinweis auf besonders wichtige Informationen
	Warnhinweise

Die Darstellungsmittel der Erklärungskomponente des Optimizers werden gesondert im Kapitel „Erklärungskomponente des Optimizers“ auf Seite 131 erläutert.

2 Werkzeuge zur Performance-Analyse

Dieses Kapitel beschreibt:

- welche Mess- und Analyse-Tools für das Tuning und die Engpassanalyse zur Verfügung stehen
- welche Informationen der SESAM/SQL-Performance-Monitor SESMON liefert
- welche Informationen die Erklärungskomponente des Optimizers zur Verfügung stellt
- wie mit der SESAM/SQL-Auftragsprotokollierung SESCOS Transaktionen und Anweisungen protokolliert und analysiert werden können
- wie mit einem View des SYS_INFO_SCHEMA „teure“, d.h. besonders viele Betriebsmittel verbrauchende, SQL-DML-Anweisungen ermittelt werden können.

2.1 Mess- und Analyse-Tools

Zum Tuning bzw. zur Engpassanalyse stehen dem DB-Administrator mehrere Mess- bzw. Analyse-Tools zur Verfügung. Dies sind:

- der BS2000-Monitor openSM2
- der SESAM/SQL-Performance-Monitor SESMON
- der SQL-Optimizer mit seiner Erklärungskomponente
- die SESAM/SQL-Auftragsprotokollierung SESCOS mit dem Auswertungsprogramm SESCOSP
- der View SYS_DML_RESOURCES

Jedes der oben genannten Messinstrumente hat sein spezifisches Anwendungsszenario.

openSM2 und SESMON sind Monitore, die statistische Daten sammeln und ausgeben, was den „normalen“ Betrieb wenig beeinflusst (2-3% der CPU-Leistung).

Typische Größen, die mit Hilfe von openSM2 bestimmt werden können, sind Überlastung von einzelnen Kanälen bzw. Platten, Auslastung des DAB (Disk-Access-Buffer) sowie Auslastung der CPU durch die einzelnen Programme
(näheres siehe Handbuch „openSM2 (BS2000)“).

Jedoch erhält man über openSM2 keine Information über die Auslastung der SESAM/SQL-Puffer und anderer DBH-Betriebsmittel. Hier setzt der Performance-Monitor SESMON an. Mit Hilfe von SESMON kann die Fragestellung beantwortet werden: „Sind meine DBH-Optionen „vernünftig“ ?“. Die Fragestellung „Ist meine Plattenverteilung unter den gegebenen Randbedingungen „optimal“ ?“ kann dagegen nur openSM2 beantworten.
Neben der reinen Auslastungsproblematik gibt SESMON auch noch Auskunft über die aktuellen Locksituationen (siehe Handbuch „Datenbankbetrieb“).

SESCOS erlaubt es, Abläufe im System zu analysieren. An wohldefinierten Punkten (z.B Auftragseingang, -ausgang, Threadwechsel usw.) werden Messgrößen ermittelt und für jeden Auftragsabschnitt verwaltet. So kann mit Hilfe des Auswertungsprogramms SESCOSP der Anteil der Kosten eines Verarbeitungschrittes an den Gesamtkosten ermittelt werden. In Verbindung mit der Erklärungskomponente des SQL-Optimizers kann so die Abarbeitung einer Anweisung untersucht und optimiert werden (siehe Handbuch „Datenbankbetrieb“). Das Einschalten von SESCOS bedeutet eine erhebliche Belastung des DBH und sollte deshalb nur ganz gezielt zum Tunen einzelner Anweisungen auf einem Test-DBH eingesetzt werden.

SESAM/SQL protokolliert automatisch in einem internen Puffer besonders „teure“ SQL-DML-Anweisungen. Eine DML-Anweisung gilt als teuer, wenn die Menge der von ihr verwendeten Betriebsmittel verglichen mit anderen SQL-DML-Anweisungen sehr hoch ist. Mit dem View SYS_DML_RESOURCES des SYS_INFO_SCHEMA können Informationen über besonders „teure“ Anweisungen ermittelt werden. Die gesammelten Daten beinhalten pro Anweisung z.B. die Anwendung, den Auftraggeber, den Anweisungstyp und die Menge der verbrauchten Betriebsmittel.

2.2 SESMON

Der Performance-Monitor SESMON ermittelt aktuelle Betriebsdaten des SESAM/SQL-DBH und von SESAM/SQL-DCN und gibt diese wahlweise auf dem Bildschirm, auf SYSLST oder auf Datei aus.

SESMON kann im Dialog oder im Batch gestartet werden.

Der SESAM/SQL-DBH, SESAM/SQL-DCN und das Konnektionsmodul DBCON pflegen Statistikzähler (auch wenn der Monitor nicht hochgefahren wird). SESMON läuft in einer eigenen Task und übernimmt die Auswertung und Ausgabe der statistischen Daten.

Die Performance des SESAM/SQL-DBH und von SESAM/SQL-DCN wird durch den Betrieb des Monitors nicht beeinträchtigt.

Die Ausgabe auf SYSLST oder Datei bietet die Möglichkeit, über einen längeren Zeitraum Statistikdaten zu protokollieren und später hinsichtlich Tuning-Maßnahmen auszuwerten.

Mit Hilfe eines Anwenderprogramms können die auf Datei protokollierten Daten ausgewertet werden.

Um sich über die momentane Situation zu informieren, kann der Benutzer den Monitor im Dialog starten und die Bildschirmausgabe wählen.

Bei Ausgabe auf Bildschirm kann der Benutzer die Statistikdaten differenziert auswählen, da die Daten auf verschiedene Masken aufgeteilt sind. Eine differenzierte Auswahl ist auch bei Ausgabe auf SYSLST möglich.

Die Daten über den SESAM/SQL-DBH sind folgendermaßen in Masken aufgeteilt:

- Die Maske „OPTIONS“ informiert über die gegenwärtig eingestellten DBH-Optionen.
- Die Maske „SYSTEM INFORMATION“ informiert über die Größe und Belegung der Work- und Transfer-Container, Anzahl der Cursor, Anzahl der vorhandenen bzw. benutzten logischen Dateien sowie über den Fortschritt eines Wiederanlaufs der DBH-Session.
- In der Maske „SQL INFORMATION“ werden Informationen über SQL-DML-Anweisungen, Pläne und Aufrufe spezieller SQL-Komponenten ausgegeben.

- In der Maske „SERVICE TASKS“ werden Informationen über die Service-Tasks und die zu bearbeitenden Aufträge ausgegeben.
- In der Maske „SERVICE ORDERS“ sind Informationen über die gerade in den Service-Tasks bearbeiteten DDL- bzw. Utility-Anweisungen sowie die User-Id zum Auftrag enthalten. Zusätzlich wird der Fortschritt einer RECOVER/REFRESH-Anweisung dargestellt.
- In der Maske „I/O“ sind Informationen über die I/O-Zugriffe auf die Spaces und die Cursor-Dateien enthalten. Außerdem sind Informationen über das I/O-Verhalten der übrigen vom SESAM-DBH verwendeten Dateien (DA-LOG, CAT-LOG, TA-LOG und CAT-REC) enthalten.
- Die Maske „TRANSACTIONS“ informiert über die Anzahl der Transaktionen, gegliedert nach verschiedenen Transaktionszuständen, Anzahl der Anweisungen (CALL-DML und SQL) und Anzahl der Vorgänge.
- Die Maske „STATEMENTS“ gibt eine Übersicht über alle laufenden Anweisungen. Es werden sowohl Anweisungen, die innerhalb einer Transaktion, als auch solche, die außerhalb einer Transaktion ablaufen, gezeigt.
- Die Maske „SYSTEM THREADS“ gibt Statistikdaten zu den Schreibthreads aus.
- Die Maske „TASKS“ stellt Statistikdaten für das Multitaskingsystem zur Verfügung, die auf die Auslastung der einzelnen DBH-Tasks schließen lassen.

Die Informationen über den SESAM/SQL-DCN Betrieb sind folgendermaßen aufgeteilt:

- In der Maske „OVERVIEW“ werden allgemeine Informationen zum DCN-Betrieb ausgegeben. Das betrifft Werte aus der Parameterdatei sowie Werte, die sich auf die ganze DCN-Session beziehen.
- In der Maske „CAPACITY“ werden Werte ausgegeben, die das Nachrichtenaufkommen und die Pool-Belegung betreffen. Außerdem sind hier die Angaben über Betriebsmittel-Engpässe enthalten.
- Die Maske „TRANSACTIONS“ informiert über den Zustand und das Verhalten von Transaktionen.
- Die Maske „APPLICATIONS“ enthält Informationen zu den einzelnen Applikationen.

Außerdem bietet SESMON zu jeder Konfiguration die Maske „PREFETCH-BUFFERS“. Sie enthält Informationen über Nutzung und Auslastung des Prefetch-Puffers, der zur Unterstützung der Schubmodus-Funktionalität an der SQL-Schnittstelle verwendet wird.

Wie der Datenbankbetrieb mit Hilfe von SESMON optimiert werden kann, ist den Hinweisen im Kapitel „Gezielte Hinweise zum Tuning von Anwendungen“ auf Seite 113 zu entnehmen.

2.3 SQL-Optimizer

SESAM/SQL erstellt für die meisten SQL-Anweisungen intern eine Auswertungsvorschrift, den SQL-Zugriffsplan. Der SQL-Optimizer sorgt bei SQL-DML-Anweisungen dafür, dass ein besonders effizienter Zugriffsplan erstellt wird, bei dem möglichst wenig Systemressourcen beansprucht werden.

Mit der Erklärungskomponente des SQL-Optimizers kann man untersuchen, in welchen Einzelschritten eine SQL-DML-Anweisung (INSERT, UPDATE, DELETE, SELECT, MERGE und CALL) abgearbeitet wird. Wenn man ein Pragma EXPLAIN in die SQL-Anweisung einfügt, erhält man eine lesbare Darstellung des internen Zugriffsplans auf Datei (siehe Handbuch „SQL-Sprachbeschreibung Teil 1: SQL-Anweisungen“).

Das Pragma EXPLAIN hat folgende Form:

```
--%PRAGMA EXPLAIN INTO file
```

Das Kapitel „Erklärungskomponente des Optimizers“ auf Seite 131 des vorliegenden Handbuchs beschreibt die Ausgabe der Erklärungskomponente und die darin enthaltene Information. Insbesondere erhalten Sie Informationen über die genutzten Indizes, über Join-Reihenfolgen und die verwendeten Join-Algorithmen.

Der Abschnitt „Aufbau und Steuerung des Optimizers“ auf Seite 24 beschreibt die Kriterien für die Auswahl von Indizes und relationalen Operationen.

Folgende Faktoren beeinflussen den internen Zugriffsplan des SQL-Optimizers:

- Pragmas und Annotationen
- Catalog-Inhalt
- Datenbankinhalt

Pragmas

Für den Optimierungsprozess sind folgende Pragmas von Bedeutung (siehe auch Abschnitt „Wirkung optimierungsrelevanter Pragmas“ auf Seite 39):

- OPTIMIZATION LEVEL *n*
- SIMPLIFICATION ON/OFF
- JOIN
- KEEP JOIN ORDER
- IGNORE/USE INDEX *indexname*
- IGNORE/USE SORT_INDEX *indexname*

Catalog-Inhalt

Die Optimierung orientiert sich an den vorhandenen Indizes und Integritätsbedingungen.

Datenbankinhalt

Die Optimierung orientiert sich an geschätzten Treffermengen. Diese Schätzung hängt maßgeblich von Statistik-Informationen ab. Daher sollte unbedingt darauf geachtet werden, dass die Statistik-Information die tatsächliche Werteverteilung in der Datenbank beschreibt. Gegebenenfalls kann mit der Anweisung REORG STATISTICS FOR INDEX die Statistik-Information zu einem Index aktualisiert werden.

Wenn Sie den Catalog-Inhalt oder den Datenbankinhalt modifizieren und insbesondere über Pragmas versuchen, den Zugriffsplan zu optimieren, können Sie mit Hilfe der Erklärungskomponente nachvollziehen, in welcher Form sich der Plan verändert hat.

Da auch der Datenbankinhalt eine Rolle spielt, sollten Versuche zur Einstellung der Pragmas und der Indizes am Original oder an einer realitätsnahen Testdatenbank erfolgen.

2.4 SESCOS und SESCOSP

SESCOS ist ein Mess- und Diagnosemittel, mit dem in einem gewissen Zeitraum Transaktionen, Anweisungen oder Bearbeitungsschritte einer Anweisung protokolliert und analysiert werden können. Damit kann festgestellt werden, wie lange eine Transaktion, eine Anweisung oder ein Bearbeitungsschritt dauert und welche Betriebsmittel dabei verbraucht werden.

SESCOSP ist ein Dienstprogramm für die Ausgabe der protokollierten Aufträge.

2.4.1 Abgrenzung gegen andere Werkzeuge zur Performance-Analyse

SESCOS eignet sich nicht zur Analyse des Systemverhaltens und der Systemauslastung des SESAM/SQL-DBH. Dazu ist der Performance-Monitor SESMON das geeignete Hilfsmittel.

Ebenso kann mit SESCOS nicht festgestellt werden, in welche Bearbeitungsschritte eine Anweisung vom Optimizer aufgeteilt wurde. Diese Aufteilung kann jedoch mit Hilfe der Erklärungskomponente ermittelt werden. Zur Verifizierung, welche Kosten ein solcher Bearbeitungsschritt verursacht, kann SESCOS sehr wohl als Ergänzung zur Erklärungskomponente verwendet werden.

2.4.2 Empfohlenes Vorgehen bei der Analyse mit Hilfe von SESCOS und SESCOSP

Bei der Analyse mit Hilfe von SESCOS und SESCOSP empfiehlt sich folgendes Vorgehen:

- Protokollieren der SESAM-Aufträge mit SESCOS über einen Zeitraum, der hinreichend groß ist, um das Performance-Problem in der Auswertung auch sichtbar zu machen. Ein Protokollieren über einen zu großen Zeitraum erfordert unnötig viel Platz für die CO-LOG-Datei und macht die Auswertung unnötig lang und aufwendig. Eine Protokollierung über einen größeren Zeitraum macht nur dann Sinn, wenn mittels USER-Selektion die Angabe auf die CO-LOG-Datei entsprechend eingeschränkt wird.
  Außerdem ist zu beachten, dass die Auftragsprotokollierung den SESAM-DBH zusätzlich belastet. Insbesondere schlägt sich diese zusätzliche Belastung auch im Messergebnis nieder, was aber die generelle Aussage nicht stört.
- Auswerten der Daten mit dem Auswertungsprogramm SESCOSP.
- Falls nicht bekannt ist, welcher Benutzer oder welche Anweisung das Performance-Problem ausmacht, empfiehlt es sich, zuerst die Transaktions-Statistik über den gesamten Mitschnitt-Zeitraum auszuwerten. Daraus ist dann ersichtlich, welche Transaktionen die meisten Betriebsmittel verbrauchen und zu welchem Benutzer sie gehören. Daraus kann dann ein Zeitraum ermittelt werden, der näher untersucht werden soll.
- Besonders „teure“ SQL-DML-Anweisungen können mit dem View SYS_DML_RESOURCES des SYS_INFO_SCHEMA ermittelt werden, siehe Abschnitt „View SYS_DML_RESOURCES“ auf Seite 21.
- Falls eine einzelne Anweisung oder ein hinreichend kleiner Zeitraum bekannt ist, der näher untersucht werden soll, kann für diese Anweisung bzw. für diesen Zeitraum die Anweisungsstatistik ausgewertet werden. Dabei kann mit dem Aufbereitungs-Format „*STRING-FORMAT“ die Anweisung abdruckbar und mit dem Aufbereitungs-Format „*IO-STATISTICS“ das IO-Verhalten jeder Anweisung ausgegeben werden.
- Falls eine einzelne Anweisung oder einige wenige Anweisungen analysiert werden sollen, können für diese Anweisungen die Aufbereitungs-Formate „*STEP-IO-STATISTICS“ bzw. „*STEP-COMPLEXITY“ ausgegeben werden, die das IO-Verhalten bzw. die Komplexität eines jeden Bearbeitungsschritts wiedergeben.

2.5 View SYS_DML_RESOURCES

Performance-Verbesserungen können durch die Analyse, Beurteilung und Verbesserung besonders „teurer“ SQL-DML-Anweisungen erzielt werden.

Eine SQL-DML-Anweisung gilt als teuer, wenn die Menge der von ihr verwendeten Betriebsmittel verglichen mit anderen SQL-DML-Anweisungen sehr hoch ist.
Betrachtete Betriebsmittel einer Anweisung sind die Anzahl der logischen und physikalischen Ein-/Ausgaben, die abgelaufene Zeit (Elapsed Time) und die Aktivitätszeit dieser Anweisung in DBH und in den Service-Tasks. Die Aktivitätszeit liefert treffendere Aussagen als die Elapsed Time, da die Zeiten, in denen die SQL-DML-Anweisung deaktiviert war (wegen Transaktionssperre oder Warten auf Aktivierung eines Service-Task-Auftrags), nicht mitberechnet werden. SESAM/SQL bestimmt aus diesen Betriebsmitteln eine Maßzahl (`MEASURE_OF_COSTS`) für die Kosten der Anweisung.

SESAM/SQL protokolliert die SQL-DML-Anweisungen mit einer besonders hohen Maßzahl in einem internen, zyklisch überschreibbaren Puffer. Das Protokollieren erfolgt automatisch und erfordert keine vorbereitenden Maßnahmen. Es beeinträchtigt die Performance nicht.

Der View SYS_DML_RESOURCES des SYS_INFO_SCHEMA liefert Informationen über besonders „teure“ SQL-DML-Anweisungen, siehe das Handbuch „SQL-Sprachbeschreibung Teil 1: SQL-Anweisungen“. Er gibt im Prinzip den Inhalt des beschriebenen Puffers aus. Die gesammelten Daten beinhalten pro Anweisung z.B. die Anwendung, den Auftraggeber, den Anweisungstyp und die Menge der verbrauchten Betriebsmittel.

Die betreffenden SQL-DML-Anweisungen können z.B. über ihren Startzeitpunkt, den System- und Anwendungsnamen sowie den Auftraggeber identifiziert werden. Wenn eine kritische Anweisung erkannt wird, dann können z.B. durch Einsatz von SESCOS detailliertere Informationen ermittelt werden.

3 Performance-relevante Aspekte von Anwendungen

Dieses Kapitel beschreibt wichtige Anwendungsaspekte, die performance-relevant sind, und auf die der Anwender ggf. durch geeignete Maßnahmen einwirken kann:

- Aufbau und Steuerung des Optimizers
- Synchronisationsmechanismen
- Ressourcenverbrauch
- Einsatz von Routinen

3.1 Aufbau und Steuerung des Optimizers

In diesem Abschnitt wird beschrieben, wie der Optimizer bei der Erzeugung von Zugriffsplänen für SQL-Anweisungen vorgeht und welche Optimierungsschritte und -techniken dabei angewendet werden. Der Abschnitt wendet sich speziell an Leser, die mit Hilfe von Pragmas und zusätzlichen Indizes die Auswertung von SQL-Anweisungen beschleunigen möchten, bzw. verstehen wollen, warum vom Optimizer ein bestimmter Zugriffsplan gewählt wurde.

CALL-DML-Anweisungen sind nicht Gegenstand dieses Abschnitts, da hier andere Optimierungstechniken als bei SQL-Anweisungen zum Tragen kommen.

Für das Verständnis dieses Abschnitts wird vorausgesetzt, dass der Leser fundierte Kenntnisse über die Optimierung von Abfrage-Ausdrücken hat, wie sie z.B. in den einschlägigen Lehrbüchern über SQL-Systeme vermittelt werden.

3.1.1 Pläne

Die Hauptaufgabe des Optimizers besteht darin, aus der textuellen Darstellung einer SQL-Anweisung einen, bzgl. Antwortzeit- und Durchsatzverhalten, möglichst günstigen SQL-Zugriffsplan zu erzeugen. Ein SQL-Zugriffsplan, kurz Plan, ist eine Auswertungsvorschrift für (Teile von) SQL-Anweisungen. Ein Plan beschreibt Typ und Reihenfolge einzelner Auswertungsschritte wie beispielsweise „Lies nächsten Primärdatensatz“, „Lies nächsten Index-Eintrag“ oder „Erzeuge Spaltenbeschreibung“.

Folgendes ist zu beachten:

- Nicht für jede SQL-Anweisung wird ein Plan erzeugt. So wird z.B. für die Anweisungen COMMIT, ROLLBACK, SET TRANSACTION, SET SCHEMA usw. kein Plan erzeugt.
- Für DDL-, SSL- und Utility-Anweisungen werden zwar Pläne erzeugt, jedoch ist hier eine Optimierung des Plans nicht erforderlich. Daher werden diese Anweisungen im vorliegenden Abschnitt „Aufbau und Steuerung des Optimizers“ nicht weiter betrachtet.

3.1.2 Phasen der Planerzeugung

Die Erzeugung eines Plans umfasst im Allgemeinen die lexikalische, syntaktische und semantische Analyse der SQL-Anweisung, die Generierung und Bewertung mehrerer alternativer Auswertungsvorschriften und schließlich die Auswahl des (gemäß Kostenmodell) günstigsten Plans. Die Optimierung eines Plans erfolgt in mehreren unabhängigen Phasen.

3.1.2.1 Phase 1: Front-End

Das Front-End nimmt die syntaktische und semantische Analyse der SQL-Anweisung vor. Es erzeugt für die Anweisung einen sog. Standardplan, der bereits eine vollständige und gültige - wenn auch wenig effiziente - Auswertungsvorschrift darstellt. In den nachfolgenden Phasen wird dieser Standardplan schrittweise verbessert.

3.1.2.2 Phase 2: Algebraische Optimierung

Die algebraischen Optimierungstechniken sind noch unabhängig von den vorhandenen Zugriffspfaden (Indizes), Auswertungsmethoden (wie z.B. spezielle Join-Algorithmen), Kosten- und Trefferabschätzungen. Die algebraischen Optimierungstechniken untergliedern sich selbst wieder in die folgenden Phasen:

- Normalization (Vereinheitlichung)
- Simplification (Vereinfachung)
- Nesting (Schachtelung).

Normalization (Vereinheitlichung)

Ziel der Normalization ist es, textuell verschiedene, aber semantisch äquivalente SQL-Anfragen in eine einheitliche, normalisierte interne Darstellung zu transformieren, so dass die nachfolgenden Optimierungsphasen von dieser einheitlichen Darstellung ausgehen können.

Beispiele zur Normalisierung

1. NOT, IN, BETWEEN Eliminierung:

```
NOT (R.a = 5)                ⇒  R.a <> 5
R.a IN (10,20,30)            ⇒  R.a = 10 OR R.a = 20 OR R.a = 30
R.a BETWEEN :var1 AND :var2 ⇒  R.a ≥ :var1 AND R.a ≤ :var2
```

2. Prädikat-Normalisierung ⇒ konjunktive Normalform (CNF):

```
R.a > 10 OR (R.b = 20 AND R.c = 5) ⇒
CNF (R.a > 10 OR R.b = 20 AND R.a > 10 OR R.c = 5)
```

3. View-Substitution:

```
DECLARE VIEW V AS SELECT R.a, R.b, R.c
                  FROM   R
                  WHERE  R.c > 20

SELECT V.a FROM V WHERE V.b = :var1  ⇒
SELECT R.a FROM R WHERE R.b = :var1 AND R.c > 20
```

Simplification (Vereinfachung)

Ziel der Simplification ist es, sämtliche Suchbedingungen (von geschachtelten Abfrage-Blöcken und Views) an einer Stelle zusammenzufassen, nämlich im äußersten Abfrage-Block und daran anschließend Prädikate und Ausdrücke zu vereinfachen (z.B. Erkennung und Eliminierung von Widersprüchen oder Tautologien). Dies ist insbesondere vorteilhaft bei Join-Ausdrücken und Anfragen auf Views.

Beispiele zur Simplification

1. Eliminierung überflüssiger Prädikate:

   ```
   WHERE R.a > 20 AND R.a > 10  ⇒  WHERE R.a > 20
   ```

2. Erkennung von Widersprüchen (leere Treffermenge):

   ```
   WHERE R.a > :var1 AND R.a IS NULL  ⇒  FALSE
   ```

3. Auswertung konstanter Ausdrücke:

   ```
   a)  WHERE R.a > 3 + S.b + 4            ⇒ WHERE R.a > S.b + 7
   b)  WHERE R.a > 2 * :varx + 3 * :vary  ⇒ WHERE R.a > p
   ```

   Der rechte Vergleichsoperand p wird einmal vorab berechnet, statt neu für jeden Satz von R.

4. Konstanten-Propagierung bei lokalem und Join-Prädikat

   ```
   SELECT R.a, S.b
   FROM   R, S
   WHERE  R.c = 10
   AND    R.c = S.c
   ⇒
   SELECT R.a, S.b
   FROM   R, S
   WHERE  R.c = 10  (1)
   AND    S.c = 10  (2)
   ```

   Der Join wird eliminiert zugunsten eines kartesischen Produkts zwischen den beiden kleineren Treffermengen der beiden lokalen Prädikate (1) und (2).
   (Eine analoge Propagierung von Parameterwerten erfolgt nicht, da deren Werte zum Optimierungszeitpunkt nicht bekannt sind. Auch bei den anderen Vergleichsoperanden (<>, >, ≥, <, <=) erfolgt keine Propagierung.)

5. Umformung eines Join in ein IN-Prädikat mit einer Unterabfrage
(nur für den Spezialfall, dass die Select-List ausschließlich DISTINCT-Spalten eines einzigen Join-Operanden enthält):

```
SELECT DISTINCT R.a, R.b
FROM   R, S
WHERE  R.a = S.a
AND    R.b > 10
⇒
SELECT DISTINCT R.a, R.b
FROM   R
WHERE  R.b > 10
AND    R.a IN (SELECT S.a FROM S)
```

Der Vorteil dieser Darstellung ist, dass nicht alle S.a-Werte untersucht werden müssen, sondern der Scan über S nach dem ersten Treffer mit S.a=R.a gestoppt werden kann.

Beeinflussen der Simplification
Solange der Standardplan keine Views und keine überflüssigen Ausdrücke oder Prädikate enthält, und außerdem kein Join-Ausdruck vorkommt, bei dem die angesprochenen Optimierungsschritte greifen, führt die Simplification zu keiner Verbesserung des Standardplans. Um Optimierungszeit zu sparen, kann die Phase per Pragma „SIMPLIFICATION OFF“ abgeschaltet werden.
Bei besonders „einfachen“ Abfrage-Ausdrücken schaltet das System die Simplification-Phase automatisch ab, weil angenommen wird, dass sich der Optimierungsaufwand nicht lohnt. Die Simplification kann per Pragma „SIMPLIFICATION ON“ erzwungen werden.

Nesting (Schachtelung)

Ziele des Nestings sind:

- Prädikate frühzeitig auszuwerten, um damit die Größe von Zwischenergebnissen zu reduzieren
- teure Auswertungsschritte wenn möglich zu eliminieren oder nur einmal vorab zu berechnen

Dies soll an einigen Beispielen verdeutlicht werden.

Beispiele zum Nesting

1. Vorziehen von Restriktionen

```
SELECT R.a, S.a
FROM   R, S
WHERE  R.b = S.b   (1)
AND    R.c > 20    (2)
AND    S.c > 30    (3)
```

   Die lokalen Teilprädikate (2) und (3) werden zuerst auf R bzw. S ausgewertet, anschließend wird das Joinergebnis nach (1) ermittelt (nicht auf den ganzen Relationen R und S, sondern auf den kleineren Treffermengen von (2) und (3)).

2. Bildung von vorab berechenbaren Ausdrücken („Precomputables“)

   Precomputables sind teure Ausdrücke, die nur einmal ausgewertet werden.

   a)
```
SELECT R.a  FROM R
WHERE  R.b >(SELECT MAX(S.b)
             FROM   S
             WHERE  S.c = :var1)
```

   Die unkorrelierte Unterabfrage bezüglich S wird zum Precomputable, d.h. der Unterabfrage-Ergebniswert wird nur einmal zu Beginn der Verarbeitung ermittelt, nicht erneut für jedes Tupel in R.

   b)
```
WHERE  R.b > 3 * :varx + :vary
```

   Der rechte Vergleichswert wird zum Precomputable und damit für den DBH-Kern zum atomaren Vergleichswert. Wenn ein Index auf R.b existiert, so wird erst dadurch die Indexnutzung möglich (der Vergleichswert darf kein arithmetischer Ausdruck sein).

3. Erkennung von vorab auswertbaren Prädikaten („Preconditions“)

```
SELECT R.a FROM R
WHERE  R.a   > :var1   (1)
AND    :var2 > 30      (2)
```

Die Bedingung (2) ist unabhängig von den Datensätzen in R und kann deshalb vorab berechnet werden:
- falls FALSE herauskommt, so ist die Treffermenge leer (unabhängig von R, d.h. der Zugriff auf die einzelnen Datensätze ist nicht mehr erforderlich).
- falls TRUE herauskommt, so muss für jeden Datensatz in R nur noch Bedingung (1) geprüft werden.

Preconditions sind nur dann möglich, wenn das Teilprädikat mit dem Rest ausschließlich AND-verknüpft ist (wenn im Beispiel AND durch OR ersetzt wird, kann keine Precondition extrahiert werden).

4. DISTINCT-Eliminierung via Eindeutigkeitsbedingung

```
SELECT DISTINCT R.a FROM R
```

Falls eine Eindeutigkeitsbedingung auf R.a existiert, wird die sonst notwendige Duplikateliminierung (incl. Sortierung) eingespart.

3.1.2.3 Phase 3: Zugriffspfadauswahl

Die Optimierungstechniken zur Zugriffspfadauswahl berücksichtigen speziell die im System vorhandenen Zugriffspfade (Indizes) und Auswertungsmethoden. Im Gegensatz zu den anderen Phasen genügt es bei der Zugriffspfadauswahl nicht, immer nur eine Planvariante („straight forward“) zu erzeugen. In vielen Situationen müssen mehrere Planvarianten untersucht und bewertet werden, um den günstigsten Plan zu finden. Bei der Bewertung spielen Kosten- und Trefferabschätzungen, die anhand von Statistik-Informationen über die Werteverteilung in der Datenbank vorgenommen werden, eine entscheidende Rolle. Daher sollte mit der Anweisung REORG STATISTICS FOR INDEX von Zeit zu Zeit dafür gesorgt werden, dass die Statistik-Information aktuell ist.

Im Allgemeinen Fall besteht eine SQL-Anfrage aus mehreren ineinander geschachtelten Abfrage-Blöcken (Unterabfragen). In jedem Abfrage-Block sind mehrere Tabellen über Joins miteinander verbunden. Auf jeder einzelnen Tabelle gibt es zusätzliche lokale Restriktionsbedingungen.

An dieser generellen Struktur einer Anfrage orientiert sich die Vorgehensweise in der Zugriffspfadauswahl. Eine (komplexe) Anfrage wird schrittweise in immer einfachere Teilanfragen zerlegt. Während der Zerlegung in immer einfachere Teilanfragen werden der Reihe nach folgende Zwischenschritte durchlaufen:

- Unterabfrage-Optimierung
- Join-Optimierung
- Optimierung von 1-Relationen-Anfragen
- Minimierung von Sortiervorgängen
- Zugriff auf Basistabellen

Optimierung einer geschachtelten Anfrage (Unterabfrage-Optimierung)

Eine geschachtelte Anfrage wird in ihre einzelnen Abfrage-Blöcke zerlegt. Jeder Abfrage-Block wird wieder für sich optimiert. Dabei wird versucht, innere Abfrage-Blöcke möglichst selten auszuwerten. Hierbei ist zwischen korrelierten und unkorrelierten Unterabfragen zu unterscheiden.

Beispiele für Unterabfrage-Optimierung

1. Unkorrelierte Unterabfrage:

   Isolierung als Precomputable

   ```
   SELECT R.a  FROM R
   WHERE  R.b > (SELECT MAX(S.b)
                 FROM   S
                 WHERE  S.c = :var1)
   ```

   Die unkorrelierte Unterabfrage bezüglich S wird zum Precomputable, d.h. der Ergebniswert wird nur einmal zu Beginn ausgewertet (nicht erneut für jedes Tupel in R).

   Eine solche Optimierung erfolgt stets und kann nicht durch ein Pragma beeinflusst werden.

2. Korrelierte Unterabfrage:

   Minimierung der Auswertung durch Sortierung der „äußeren Werte“

   ```
   SELECT R.a  FROM R
   WHERE  R.b > (SELECT S.b
                 FROM   S
                 WHERE  S.c > R.c)
   ```

   Falls die Auswertung der Unterabfrage bzgl. S „teuer“ ist, so werden die Tupel von R nach R.c sortiert angeliefert, so dass das Ergebnis der korrelierten Unterabfrage zwischengespeichert werden kann und nur bei einem neuen R.c-Wert neu berechnet werden muss.

   Diese Optimierung erfolgt nur für Außenreferenzen in dem direkt umgebenden Abfrage-Block und nur, wenn diese Außenreferenzen keine Spalten mit Eindeutigkeitsbedingung sind.

   Der Anwender kann die Optimierung einer korrelierten Unterabfrage unterdrücken, indem er bei PRAGMA OPTIMIZATION LEVEL n einen niedrigen Wert (z.Z. $n \leq 4$) angibt. Ohne Angabe des Pragmas wird die Optimierung durchgeführt, d.h. die Auswertungskosten der Unterabfrage werden abgeschätzt, und es wird genau dann eine Sortieroperation eingeführt, wenn die Kosten der Unterabfrage einen bestimmten Grenzwert überschreiten.

Optimierung eines Abfrage-Blocks (Join-Optimierung)

Es werden mehrere Join-Reihenfolgen betrachtet, um die günstigste Reihenfolge festzulegen. Die einzelnen Operanden des Joins (1-Relationen-Anfragen) werden wieder für sich optimiert. Die Join-Reihenfolge wird nach folgender Strategie festgelegt:

1. Anwendung aller elementaren Equi-Joins
   (d.h. Bedingungen der Art R.a = S.b, nicht jedoch R.a+1 = 2*S.a)
2. gegebenenfalls Anwendung elementarer Theta-Joins
   (d.h. Bedingungen R.a OP S.a, OP in {>,≥,<,≤}).

Alle sonstigen Join-Prädikate werden nicht weiter optimiert und sollten daher vermieden werden.

Beispiel

```
SELECT R.d, S.d, T.d, U.d
FROM   R, S, T, U
WHERE  R.a = S.a
AND    S.b = T.b
AND    T.c > U.c
```

1. Zuerst wird der Join von (R, S, T) geplant:
   Aus den beiden Reihenfolgen JOIN (R, JOIN(S,T)) und JOIN (JOIN(R,S), T) wird die kostengünstigere und der jeweils anzuwendende Join-Algorithmus ausgewählt.
2. Das Kompositum (R, S, T) wird äußere Relation eines Nested-Loop-Join (und damit nur einmal berechnet); U wird als innere Relation gewählt (gegebenenfalls wird dabei ein vorhandener Index auf U.c genutzt).

Nur wenn das Join-Prädikat mehr als eine elementare Equi-Join-Bedingung enthält, lässt diese Strategie noch Freiheitsgrade für die Festlegung der Join-Reihenfolge offen. Die tatsächliche Join-Reihenfolge kann dann auf drei Arten bestimmt werden:

- Ohne Pragma:
  Der Optimizer generiert einige/alle möglichen Alternativen, führt für jede Planvariante eine Kostenabschätzung durch und wählt die Planvariante mit den geringsten Kosten aus.
- Durch PRAGMA OPTIMIZATION LEVEL n (n ≤ 8):
  Der Optimizer generiert (quasi nichtdeterministisch) nur eine Join-Reihenfolge (die natürlich der obigen Strategie folgen muss) und ignoriert von vornherein alle weiteren Alternativen. Hierdurch kann der Anwender den Optimierungsaufwand zu Lasten der Ablauf-Performance beschränken.

- Durch PRAGMA SIMPLIFICATION OFF:
  Hierdurch kann der Anwender explizit eine bestimmte Join-Reihenfolge vorgeben, an die sich der Optimizer halten muss. Der Einsatz dieses Pragmas setzt eine gute Kenntnis des Mengengerüsts und der Werteverteilung in der Datenbank voraus.
- Durch PRAGMA KEEP JOIN ORDER:
  Es veranlasst den Optimizer, sich bei der Join-Reihenfolge soweit möglich an die in der SQL-Anweisung vorgegebene Reihenfolge der Join-Operanden zu halten.

Außerdem wird für jede binäre Join-Operation der zu verwendende Join-Algorithmus festgelegt:

- für elementare Equi-Joins: Sort-Merge-Join oder Nested-Loop-Join
- für elementare Theta-Joins: Nested-Loop-Join
- für sonstige Join-Prädikate: Bildung des Kreuzprodukts, anschließende Selektion

Beim (empfehlenswerten, weil performanten) elementaren Equi-Join kann der Anwender durch Pragma-Spezifikation eines niedrigen OPTIMIZATION LEVEL n ($n \leq 5$) einen Sort-Merge-Join erzwingen; darüber hinaus hat er keine direkten Einwirkungsmöglichkeiten. Die Auswahl erfolgt auf Grund von Treffer- und Kostenabschätzungen für die beiden Join-Operanden und anhand von Heuristiken, wobei die vorliegende Schemainformation (insbesondere Existenz von Indizes) ausgenutzt wird.

Beispiel für die Problematik der Wahl des Join-Algorithmus

```
SELECT R.a, R.b, S.a, S.b
FROM   R, S
WHERE  R.a = S.a
AND    R.b = :var1
```

- Sort-Merge-Join unter Einbeziehung der Sort-Minimierung
  Falls R.a und S.a beide invertiert oder Primärschlüssel sind, so werden die Indizes durchlaufen und im Merge die Join-Treffer ermittelt, ohne dass die Treffermengen physikalisch sortiert werden müssen (teuer); fehlt ein Index, so muss der betreffende Join-Operand vorher physikalisch sortiert werden.
- Nested-Loop-Join mit Index auf innerer Relation
  Falls S.a invertiert oder Primärschlüssel ist, so kommt ein Nested-Loop-Join mit R als äußerer Relation in Betracht. Für jeden Treffer in R dient R.a als Index-Vergleichswert in S.a, um die Join-Treffer in S zu bestimmen.

Optimierung von 1-Relationen-Anfragen

Von einer Teilanfrage, deren Prädikat sich nur noch auf eine Basisrelation bezieht, wird der Teil des Prädikats separiert, der für die Zugriffspfadplanung infrage kommt.

Der DBH-Kern kann alleTeilprädikate auswerten, die folgende Konstrukte **nicht** enthalten:

- Korrelierte skalare Unterabfragen (nicht precomputable)
- EXISTS-, ANY-, ALL-Prädikate
- IS CASTABLE-Prädikate
- LIKE_REGEX-Prädikate
- IN-Prädikate mit Unterabfrage als Vergleichsoperand
- Konkatenation von Zeichenketten
- CASE, CAST und alle weiteren Funktionen, sofern sie unabhängig vom Datenbankinhalt sind, d.h. wenn sie vorübersetzt werden können

Auf dem separierten Teilprädikat werden einige weitere Optimierungen durchgeführt:

- Suchbereiche auf einem Index zusammenfassen und minimieren

  ```
  WHERE R.a > 10 AND R.a < 20
  ```
  Indexsuche auf Range-Bereich (10,20)
  (dies ist nur dann möglich, wenn keine OR-Verknüpfung mit weiteren Prädikaten auf anderen Spalten existiert; diese sollten daher sparsam verwendet werden.)

- Schemareduktion: nur benötigte Spalten einer Tabelle ausgeben

  ```
  SELECT R.a, R.b
  FROM   R
  WHERE  R.c > :var1                  (1)
  AND    R.d = :var2                  (2)
  AND    R.a IN (SELECT S.a FROM S    (3)
                 WHERE  S.b = R.b)
  ```

  Hier werden R.c und R.d nur vom DBH-Kern zur Auswertung der Prädikate (1) und (2) gelesen. An den DML-Interpreter, der noch das Prädikat (3) prüft, werden nur die dort benötigten Werte der Spalten R.a und R.b übertragen und so der Datentransport minimiert.

  Bei multiplen Spalten werden nur die tatsächlich benötigten Ausprägungen an den DML-Interpreter übergeben, nicht die gesamte Spalte.

Minimierung von Sortiervorgängen

Verzahnt mit den vorher beschriebenen Optimierungstechniken wird gezielt versucht, physikalische Sortieroperationen zu eliminieren und dennoch die gewünschte Sortierordnung einer (Zwischen-) Treffermenge zu garantieren. Hierzu wird bei der Optimierung einer Teilanfrage das geforderte Sortierkriterium als Nebenbedingung berücksichtigt.
Die Sort-Minimierung ist mit hohem Optimierungsaufwand verbunden, da hier oft gegensätzliche Kriterien (Tradeoffs) zu berücksichtigen sind und daher mehrere Planalternativen analysiert werden müssen. Wenn dieser Aufwand im Einzelfall nicht gerechtfertigt erscheint, so kann durch einen niedrigen Wert im Pragma OPTIMIZATION LEVEL n ($n \leq 6$) die Sort-Minimierung und dadurch der Optimierungsaufwand vom Anwender eingeschränkt werden.

Beispiele zur Eliminierung von Sortieroperationen

1. Vererben der Sortierordnung durch die Zwischenergebnis-Relation

```
SELECT    R.a, SUM(R.b)
FROM      R
WHERE     R.a < :varx
GROUP BY  R.a
ORDER BY  R.a
```

   – Für die ORDER BY-Klausel wird nicht neu sortiert, da das GROUP-Ergebnis bereits richtig sortiert angeliefert wird.
   – Falls R.a NOT NULL und außerdem invertiert (Sekundärindex) ist, wird R durch Index-Scan gelesen, d.h. es muss überhaupt nicht sortiert werden.

2. Reduzieren von Sort-Operationen durch geschickte Reihenfolge von Join-Operationen

```
SELECT    *
FROM      R, S, T
WHERE     R.a = S.a
AND       S.b = T.b
ORDER BY  R.a
```

   Durch die Join-Reihenfolge Sort-Merge-Join (R ORDER BY R.a, JOIN(S,T) ORDER BY S.a) wird die abschließende Sortierung nach R.a eingespart, da schon als Vorbereitung für den Sort-Merge-Join sortiert wurde.

3. Nutzung von Indizes zur Sortierung

```
SELECT    R.a, R.b
FROM      R
WHERE     R.a > 10
ORDER BY  R.a
```

   – Ist R.a Sekundärindex-invertiert (oder der Primärschlüssel bzw. dessen erste Komponente), so wird durch einen Index-Scan über R.a die WHERE-Klausel ausgewertet, gleichzeitig wird dadurch die Treffermenge richtig sortiert.
   – Fehlt dagegen die Bedingung R.a > 10, und sind darüber hinaus NULL-Werte erlaubt, so kann der Index nicht zur Sortierung genutzt werden, da sonst die nicht-signifikanten Sätze verlorengehen würden. Im Sinne der Sort-Minimierung sind Indizes daher besonders für signifikante Spalten empfehlenswert.

Zugriff auf Basistabellen

Anhand der Verteilung von Spaltenwerten und den in der Anfrage benutzten Vergleichswerten wird die günstigste Methode für den Zugriff auf eine einzelne Basistabelle festgelegt.

Zur Auswahl stehen folgende Zugriffsmethoden (siehe auch Beispiele auf Seite 37):

- sequenzielles Lesen der gesamten Tabelle
- sequenzielles Lesen eines eingeschränkten Primärschlüsselbereichs der Tabelle
- Nutzung von ein oder mehreren Sekundärindizes, um die Treffer (bzw. eine Obermenge davon) zu bestimmen, mit anschließendem Lesen der Treffersätze in der Tabelle
- sequenzielles Lesen eines Sekundärindex. Diese Methode wird angewendet, um die Sätze der Tabelle sortiert nach dem Index zu erhalten, oder um auf einem zusammengesetzten Index Suchbedingungen auszuwerten, die sich nicht auf die erste(n) Spalte(n) des Index beziehen. In beiden Fällen wird der Sekundärindex u.U. noch mit weiteren Indizes verschnitten, bevor die Treffersätze (bzw. eine Obermenge davon) in der Tabelle gelesen werden.

Voraussetzungen für die Nutzung eines Index:

1. Durch die Indexauswertung muss eine Obermenge der echten Treffermenge ermittelt werden. Dies ist in folgenden Fällen möglich:

   – das durch Index auszuwertende Teilprädikat ist mit dem Restprädikat nur durch AND verknüpft

   – das durch Index auszuwertende Teilprädikat ist nur mit solchen anderen Teilprädikaten OR-verknüpft, die selbst prinzipiell index-auswertbar sind.

2. Bei Vergleichsbedingungen muss das Prädikat von folgender Form sein:
   `spalte op wert`
   D.h. die invertierte Spalte muss als Vergleichsoperand auftreten und darf nicht in einem arithmetischen Ausdruck „versteckt“ sein (solche Prädikate sollten daher unbedingt umformuliert werden)
3. Bei NULL-Bedingungen: Der Operator muss „IS NOT NULL“ sein
4. Die Statistikabschätzung muss eine ausreichende Selektivität des Teilprädikats (≤ 75%) prognostizieren.
5. Der Index darf nicht vom Anwender durch PRAGMA IGNORE INDEX ausgeschlossen worden sein.

Falls die Anfrage Parameter oder, was auf dasselbe hinausläuft, Unterabfragen enthält, sind die Vergleichswerte noch nicht zum Zeitpunkt der Planerzeugung, sondern erst während der Planausführung bekannt. Hier wird die Zugriffspfadauswahl zum Ausführungszeitpunkt teilweise wiederholt, um die aktuellen Vergleichswerte zu berücksichtigen.

Die Schätzung der Trefferquote wird anhand der statistischen Verteilung der Indexwerte vorgenommen. Sollte diese Statistik nicht mehr der aktuellen Werteverteilung in der Tabelle entsprechen, empfiehlt es sich, die Statistik mit der Anweisung REORG STATISTICS FOR INDEX (bzw. REORG SPACE) zu aktualisieren.

Beispiele zur Indexverarbeitung

Relation R mit Spalten: a (Sekundärindex); b,c (zusammengesetzter Index); d (ohne Index) und der Spalte 'key' als Primärschlüssel

```
(1) WHERE R.key = :var1 AND ...
```

Indexverarbeitung:
Direkter Zugriff über den Primärschlüssel, gegebenenfalls werden anschließend noch die restlichen Prädikate ausgewertet (Methode 2).

```
(2) WHERE R.a = :var1
```

Indexverarbeitung:
Vollständige Prädikatauswertung mit dem Sekundärindex auf a (Methode 3).

```
(3) WHERE R.a = :var1 AND R.d > :var2
```

Indexverarbeitung:
Bestimmung einer Obermenge der Treffer mit dem Sekundärindex auf a, Prüfung von d auf Primärdaten (Methode 3).

```
(4) WHERE R.b > :var1
    WHERE R.b = :var1 AND R.c > :var2
```

Indexverarbeitung:
In beiden Fällen Nutzung des zusammengesetzten Sekundärindex (Methode 3).

```
(5) WHERE R.c > 10
```

Indexverarbeitung:
Auch ohne weitere Bedingung auf R.b Nutzung des zusammengesetzten Sekundärindex (Methode 4).

```
(6) WHERE R.a > 10  AND/OR  R.b > 20
```

Indexverarbeitung:
AND/OR-Verschneidung der beiden Index-Treffermengen (Methode 3).

```
(7) WHERE R.gebdatum > 1900 AND ...
```

Indexverarbeitung:
Index wird nicht genutzt, wenn die Statistikauswertung eine Trefferquote von mehr als 75% prognostiziert (Entscheidung wird zum Zeitpunkt der Planerzeugung getroffen). Daher: sequenzielle Suche in der Tabelle (Methode 1).

```
(8) WHERE R.gebdatum < 1900 AND R.plz BETWEEN 80000 AND 81999
```

Indexverarbeitung:
Index wird nicht genutzt, wenn das bisher ermittelte Zwischenresultat wenige Treffer enthält (Entscheidung wird zum Zeitpunkt der Planausführung getroffen).
Daher: nur Index auf „gebdatum“ verwenden, zweites Teilprädikat auf den Primärdaten prüfen (Methode 3).

3.1.3 Wirkung optimierungsrelevanter Pragmas

Für den eigentlichen Optimierungsprozess sind folgende Pragmas von Bedeutung:

- SIMPLIFICATION
- IGNORE/USE INDEX
- IGNORE/USE SORT_INDEX
- JOIN
- KEEP JOIN ORDER
- OPTIMIZATION LEVEL

Diese Pragmas können auch in Routinen und beim Aufruf von Prozeduren mit der SQL-Anweisung CALL eingesetzt werden, siehe Handbuch „SQL-Sprachbeschreibung Teil 1: SQL-Anweisungen“.

```
--%PRAGMA SIMPLIFICATION ON/OFF
```

Die kompletten Optimierungstechniken zur Simplification (Teil der algebraischen Optimierung) werden ein- bzw. abgeschaltet, d.h. die dort skizzierten Optimierungsschritte werden entweder alle durchlaufen (`ON`) oder keiner von ihnen (`OFF`).

```
--%PRAGMA IGNORE/USE INDEX index
```

Der spezifizierte Index wird bei der Festlegung der Join-Reihenfolge, des Join-Algorithmus und bei der Auswahl des optimalen Zugriffspfads (auf Basisrelationen) ignoriert (`IGNORE`) oder genutzt (`USE`).

```
--%PRAGMA IGNORE/USE SORT_INDEX index
```

Der spezifizierte Index wird bei der Auswahl des Sortier-Algorithmus (auf Basisrelationen) ignoriert (`IGNORE`) oder genutzt (`USE`).

```
--%PRAGMA JOIN [SORT MERGE | NESTED LOOP]
```

Durch Angabe des Pragmas wird die verwendete Join-Methode (Sort Merge Join oder Nested Loop Join) ausgewählt.
Wird durch das Pragma OPTIMIZATION LEVEL die Anzahl der betrachteten Planalternativen so eingeschränkt, dass kein Nested Loop Join mehr betrachtet wird (OPTIMIZATION LEVEL < 6), so kann auch durch das Pragma JOIN kein Nested Loop Join erzwungen werden.
Bei Mehrfachjoins wird das Pragma auch bei den anfallenden Zwischenjoins berücksichtigt.

Ist das Pragma JOIN nicht angegeben, so wählt der Optimizer die zu verwendende Join-Methode aus.

```
--%PRAGMA KEEP JOIN ORDER
```

Das Pragma gibt vor, wie ein mehrfacher Join abgearbeitet werden soll.

Es veranlasst den SQL-Optimizer, sich bei der Join-Reihenfolge soweit möglich an die in der SQL-Anweisung vorgegebene Reihenfolge der Join-Operanden zu halten. Dies gilt für explizite Joins (z.B. *table_1* JOIN *table_2* ON *suchbedingung*), soweit die JOIN-Annotation keinen anderen Algorithmus vorgibt.

Dieselbe Wirkung kann auch mit JOIN und SIMPLIFICATION OFF erreicht werden. Dabei werden aber auch andere Optimierungen ausgeschaltet.

Ist das Pragma KEEP JOIN ORDER nicht angegeben, so wählt der Optimizer die zu verwendende Join-Reihenfolge aus, sofern nicht eine bestimmte Reihenfolge über die Pragmas JOIN oder SIMPLIFICATION OFF erzwungen wird.

Das Pragma kann bei CALL, bei DML-Anweisungen und in Routinen eingesetzt werden.

```
--%PRAGMA OPTIMIZATION LEVEL n
```

Die Option '*n*' steuert die Anzahl der Planalternativen, die im Verlauf der Zugriffspfadauswahl erzeugt und bewertet werden.
Es werden zunächst alle Planvarianten untersucht und mittels Heuristiken nur die aussichtsreichsten Varianten, die im allgemeinen eine Verbesserung der Auswertungskosten versprechen, ausgewählt. Die Anzahl Planvarianten, die weiterverfolgt werden, hängt von '*n*' ab. Der Wert '*n*' kann zwischen 1 und 10 gewählt werden; die Voreinstellung ist 9.
Ab einem Wert $n \leq 5$ wird in jedem Optimierungsschritt nur noch eine Planvariante untersucht.

Im Einzelnen werden folgende Stufen unterschieden:

- $n \geq 9$
  Es werden verschiedene Join-Reihenfolgen betrachtet.
- $n \geq 8$
  Beim Nested-Loop-Join wird untersucht, ob die Vertauschung der beiden Join-Partner vorteilhaft ist.
- $n \geq 7$
  Durchführung der Sort-Minimierung
- $n \geq 6$
  Bei der Join-Optimierung wird neben dem Sort-Merge auch der Nested-Loop-Join betrachtet.
  Bei der Zugriffsauswahl werden alle Möglichkeiten betrachtet, eine geforderte Sortierung zu erreichen (physikalische Sortierung im DBH-Kern, Sort durch Index-Scan)

- $n \geq 5$
  Durchführung der Unterabfrage-Optimierung und Abspeicherung von mehrfach benötigten Zwischenergebnisrelationen.
- $n \geq 4$
  Durchführung der Range-Konstruktion; d.h. mehrere atomare Prädikate auf derselben Spalte werden zu einem Indexzugriff zusammengefasst.

Darüber hinaus hat dieses Pragma Auswirkungen auf die Vollständigkeit der Normalisierung von Prädikaten. Da die Normalisierung für komplexe Prädikate sehr CPU-aufwendig werden kann, führt ein niedriger Wert für 'n' bei komplexen Prädikaten dazu, dass keine vollständige konjunktive Normalform generiert wird. Diese Einsparung von CPU-Aufwand während der Optimierung führt i.a. zu erhöhten Abarbeitungskosten des Abfrage-Ausdrucks, da bei degenerierten Normalformen bestimmte Optimierungsschritte ausgelassen werden müssen.

Konkrete Hinweise zur Verwendung von Pragmas und anderen Tuning-Möglichkeiten finden Sie im Abschnitt „Optimierungsmöglichkeiten bei SQL-Anwendungen“ auf Seite 114.

Die genannten Pragmas sind ein sehr sensibles Werkzeug, das nur in Ausnahmefällen eingesetzt werden sollte. Mit Pragmas wird die Zahl der vom Optimizer betrachteten Varianten für den günstigsten Zugriffsplan eingeschränkt. In der Regel entstehen dadurch, insbesondere nach umfangreicheren Änderungen in der Datenbank (Mengengerüst, Werteverteilung, Indizes), weniger günstige Zugriffspläne.

3.1.4 Annotationen

Wie im vorigen Abschnitt beschrieben, stehen Pragmas am Anfang einer Anweisung und ihre Wirkung erstreckt sich über die gesamte Anweisung. So kann z.B. ein mit dem Pragma IGNORE INDEX ausgeschlossener Index in der gesamten Anweisung nicht verwendet werden.

Für einen lokal begrenzten Wirkungsbereich von Optimierungshinweisen gibt es so genannte Annotationen. Annotationen sind Hinweise, die innerhalb einer SQL-Anweisung an bestimmten Stellen stehen und nur bei der Optimierung der betroffenen Operationen verwendet werden.

Annotation können auch in der Definition eines Views angegeben werden. Damit können bereits in der Viewdefinition Optimierungshinweise für SQL-Anweisungen gegeben werden, die diesen View benutzen.

Wenn in einer Operation eine Annotation vorkommt, die vom Optimizer ausgewertet werden kann, dann hat diese Annotation Vorrang vor einem evtl. angegebenen optimierungsrelevantem Pragma. Dies gilt auch dann, wenn die Annotation implizit über die Definition eines Views in die SQL-Anweisung integriert wird.

3.1.4.1 JOIN-Annotation

SESAM/SQL bietet eine Annotation, um die Join-Optimierung von Inner Joins und Outer Joins zu beeinflussen:

```
T1 ... JOIN /*% {SORT MERGE | NESTED LOOP} [{LEFT|RIGHT} FIRST] %*/ T2 ON ...
```

Diese Annotation teilt dem Optimizer mit, welcher Join-Algorithmus und (optional) welche Reihenfolge der Join-Operanden für die Auswertung des Joins zwischen T1 und T2 bevorzugt wird. Wenn die vorgeschlagene Strategie möglich ist, dann wird sie vom Optimizer auch entsprechend geplant.

Annotationen können ignoriert werden, wenn sie im Kontext nicht ausführbar sind.
Z.B. ist die Annotation RIGHT FIRST bei einem LEFT OUTER JOIN sinnlos, da hier auf die LEFT-Tabelle als so genannte dominante Tabelle immer zuerst zugegriffen wird.
Auch durch die Angabe eines niedrigen Optimierungslevels kann die Anzahl der betrachteten Planvarianten so eingeschränkt werden, dass gewisse Join-Methoden oder -Reihenfolgen nicht mehr betrachtet werden und folglich eine Annotation ignoriert wird.

Mehrere Annotationen können in einer Anweisung verwendet werden, z.B. um die Planung vom Mehrfach-Joins zu unterstützen.

3.1.4.2 CACHE-Annotation

Mit dieser Annotation können Sie das Ergebnis der Tabellenfunktion CSV() in einer temporären Datei zwischenspeichern (caching). Dies beschleunigt einen mehrfachen Zugriff auf dieselbe Tabelle. CSV-Tabellen können so auch parallel bearbeitet werden.

Beispiele

Wenn in einer SQL-Anweisung eine CSV Tabelle mehrfach gelesen wird (z.B. als Unterabfrage oder als Operand in einem Join), so kann durch das Caching der CSV-Tabelle die Ausführungzeit reduziert werden.

Folgendes Beispiel ermittelt zu jedem Namen `nam` der Namenstabelle `namtab` die Summe `namscore` der Punkte `score` aus der CSV-Datei `csv.scoretab`. Die CSV-Datei enthält zu jeden Namen keine, einen oder mehrere Zeilen mit den erreichten Punkten.

```
SELECT nam, (SELECT SUM(CAST(S.score AS INT))
             FROM TABLE(CSV /*%CACHE%*/ ('csv.scoretab' DELIMITER ',' ,
                                         CHAR(20), VARCHAR(10))
                  AS S(nam, score)
             WHERE N.nam = S.nam)
             AS namscore
FROM namtab AS N
....
```

Durch das Caching einer CSV-Datei kann auch die Einschränkung bei der Verwendung von CSV-Dateien umgangen werden, dass eine CSV-Datei nicht gleichzeitig parallel gelesen werden kann. Eine CSV Datei kann zwar auch weiterhin zu einer Zeit nur von einer Transaktion gelesen werden. Wenn aber die Tabellenfunktion die Annotation CACHE enthält, so beschränkt sich die Zeit, in der paralleles Lesen nicht möglich ist, auf die Zeit, die für das Lesen der Datei in den Cache erforderlich ist. In der übrigen Verarbeitungszeit ist das parallele Lesen der CSV-Datei dann möglich.

Folgendes Beispiel wählt aus der CSV-Datei `csv.scoretab` alle Zeilen aus, in denen die Punktzahl `score` kleiner ist als die durchschnittliche Punktzahl der anderen Namen `nam` der CSV-Datei.

```
SELECT name, cast(score as int)
       FROM TABLE(CSV /*%CACHE%*/ ('csv.scoretab' DELIMITER ',',
                                   CHAR(20), VARCHAR(10)))
            AS S(nam, score)
       WHERE CAST(score AS INT) < (SELECT AVG(CAST(score AS INT))
                                   FROM TABLE('csv.scoretab' DELIMITER ',',
                                              CHAR(20), VARCHAR(10)))
                                        AS X(nam, score)
                                   WHERE X.nam <> S.nam)
```

3.1.4.3 IMMUTABLE-Annotation

Die Berechnung von Funktionen mit konstanten Eingabewerten (unkorrelierte Funktionen) hängt vom jeweiligen Aufrufkontext ab. Insbesondere werden diese Funktionen z.B. in SELECT-Listen für jeden Satz neu berechnet. Siehe die Beschreibung im Handbuch „SQL-Sprachbeschreibung Teil 1: SQL-Anweisungen, Kapitel „Routinen“.

Bei der Gestaltung der Anwendung kann durch Verwendung der Annotation IMMUTABLE dafür gesorgt werden, dass der Funktionswert unkorrelierter Funktionen nur einmal berechnet wird. Es muss aber sichergestellt sein, dass sich der Funktionswert bei gleichbleibenden Eingabewerten nicht ändert.

Bei unkorrelierten Funktionen, deren Berechnung teuere Datenbankzugriffe erfordert, kann dies zu einer spürbaren Performance-Verbesserung führen.

Beispiel

```
CREATE FUNCTION maxs( ) RETURNS INTEGER READS SQL DATA
       BEGIN
          RETURNS (SELECT max(c) FROM S);
       END
```

Die Funktion `maxs()` ist unkorreliert, würde aber innerhalb einer SELECT-Liste stets neu berechnet werden. Dies kann durch die Annotation IMMUTABLE verhindert werden:

```
SELECT x - maxs  /*% IMMUTABLE %*/ ( )
       FROM   T
       WHERE  col = ...
```

3.2 Synchronisation

Dieser Abschnitt behandelt folgende Themen:

- Phänomene und Isolationslevel
- Synchronisation von DDL und DML
- Synchronisation der Utilities

3.2.1 Phänomene und Isolationslevel

In der Definition der SQL2-Norm werden die folgenden drei Phänomene verwendet:

- P1 (Dirty read)
- P2 (Non-repeatable read)
- P3 (Phantom)

P1 (Dirty read)

(SQL-)Transaktion T1 ändert einen Satz. (SQL-)Transaktion T2 liest diesen Satz bevor T1 ein COMMIT ausführt. Wenn jetzt T1 zurückgesetzt wird, hat T2 einen Satz gelesen, der nie „commited“ war und so als nie existent betrachtet werden darf.

P2 (Non-repeatable read)

(SQL-)Transaktion T1 liest einen Satz. (SQL-)Transaktion T2 ändert oder löscht diesen Satz und führt ein COMMIT aus. Wenn T1 dann versucht, diesen Satz erneut zu lesen, bekommt sie den geänderten Satz oder erkennt, dass der Satz gelöscht wurde.

P3 (Phantom)

(SQL-)Transaktion T1 liest eine Menge N von Sätzen, die einen bestimmten Abfrage-Ausdruck erfüllen. (SQL-)Transaktion T2 führt dann (SQL-)Anweisungen aus, die einen oder mehrere Sätze erzeugen, die den Abfrage-Ausdruck von T1 erfüllen. Wenn dann T1 das ursprüngliche Lesen mit demselben Abfrage-Ausdruck wiederholt, erhält sie eine andere Menge von Sätzen.

Aufbauend auf diese 3 Phänomene werden die Isolations- bzw. Konsistenzlevel wie folgt definiert:

SQL		CALL-DML		Phänomene		
Isolationslevel	**Konsistenzlevel**	**RNL[1]**	**RNW[2]**	**P1**	**P2**	**P3**
READ UNCOMMITTED	0	x	x	x	x	x
-	1		x	x	x[3]	x
READ COMMITTED	2	x		-	x	x
REPEATABLE READ	3			-	-	x
SERIALIZABLE	4			-	-	-

Tabelle 1: Isolationslevel, Konsistenzlevel und zugeordnete Phänomene

[1] Lesen ohne zu sperren

[2] Ignorieren der Sperre

[3] Das Phänomen non-repeatable read kann auftreten, wenn vorher ein Satz mit dirty read gelesen wurde.

Zur Realisierung dieser Isolationslevel gibt es die Objekt-Sperrverwaltung, die die folgenden Sperrobjekte und ihre Hierarchie kennt:

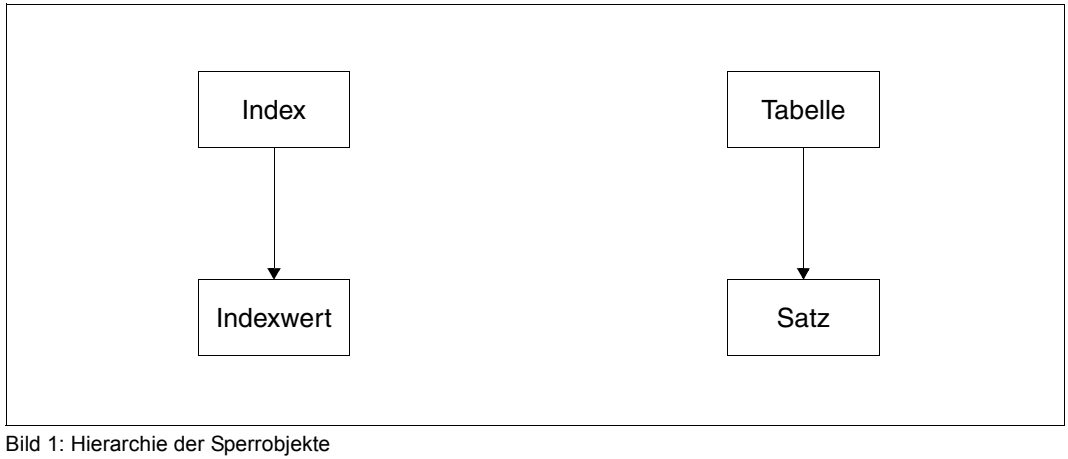


Bild 1: Hierarchie der Sperrobjekte

Dabei gelten folgende Bedingungen:

- Eine Sperre auf ein Objekt impliziert eine Sperre auf alle hierarchisch darunter liegenden Objekte.
- Gibt es sehr viele gesperrte Objekte, die alle zum selben „Vater-Objekt“ gehören, werden diese Sperren zum „Vater-Objekt“ hochgezogen.

Die Sperrverwaltung kennt Shared- und Exclusive-Sperren. Für ein Objekt kann entweder eine Exclusive-Sperre oder $n \geq 1$ Shared-Sperren aktiv sein. Sobald eine Exclusive-Sperre vorangemeldet ist, kann keine weitere Shared-Sperre mehr eingetragen werden.

Mit dieser Objekt-Sperrverwaltung werden die Isolationslevel wie folgt realisiert:

1. Sätze und Sekundärindex-Blöcke (SIS0), die geändert werden, werden stets mit einer Exclusive-Sperre versehen. Falls dies nicht möglich ist, wird der Auftrag so lange deaktiviert, bis die sperrende Transaktion beendet ist.
   Der Update arbeitet also unabhängig vom Isolationslevel.
2. Die Datenwiedergewinnung arbeitet in der Regel mit Shared-Sperren. Es gelten folgende Ausnahmen:
   – CALL-DML:
   In der Open-Anweisung kann festgelegt werden, dass mit Exclusive-Sperren gearbeitet werden soll.
   – SQL:
   Sätze werden beim Lesen mit einer Exclusive-Sperre versehen, wenn sie in derselben Anweisung geändert oder gelöscht werden oder wenn sie zu einem „FOR UPDATE“-Cursor gehören oder die SQL-Anweisung enthält %PRAGMA LOCK MODE EXCLUSIVE.
3. In den Isolationsleveln READ UNCOMMITED, READ COMMITED und REPEATABLE READ, sowie im Konsistenzlevel 1 wird beim Lesen nur mit Satz-Sperren gearbeitet:
   – Isolationslevel „READ UNCOMMITTED“ (Konsistenz-Level 0):
   Treffer-Sätze werden nicht gesperrt; Sätze, die von einer anderen Transaktion gesperrt sind, werden mit einem entsprechenden Status ausgegeben (SQLSTATE 01SA1, CALL-DML-Status 9S).
   – Konsistenz-Level 1
   Treffer-Sätze werden - sofern möglich - gesperrt; Sätze, die von einer anderen Transaktion gesperrt sind, werden mit einem entsprechenden Status ausgegeben (siehe oben).
   – Isolationslevel „READ COMMITTED“ (Konsistenz-Level 2):
   Treffer-Sätze werden nicht gesperrt. Falls ein Satz von einer anderen Transaktion gesperrt ist, wird der Auftrag so lange deaktiviert, bis sich die sperrende Transaktion beendet.

- Isolationslevel „REPEATABLE READ“ (Konsistenzlevel 3):
  Treffer-Sätze werden gesperrt; Falls ein Satz von einer anderen Transaktion gesperrt ist, wird der Auftrag so lange deaktiviert, bis sich die sperrende Transaktion beendet.
- Isolationslevel „SERIALIZABLE“:
  Hier genügt es nicht mehr einzelne Sätze zu sperren, man muss vielmehr ganze „Räume“ sperren.
  - Suche über Sekundärindex: Es werden alle ausgewerteten SIS0-Blöcke gesperrt.
  - Zusätzlich werden nicht nur die Treffer-Sätze, sondern auch die Nicht-Treffer-Sätze gesperrt.
  - Bei sequenzieller Suche wird immer auch der 1. Satz, der nicht mehr zum Primärschlüssel-Intervall gehört, gesperrt. Mit jedem Satz ist auch der logische „Raum“ zu seinem Vorgänger gesperrt.
  - Bei Tabellen ohne Primärschlüssel muss die ganze Tabelle gesperrt werden.

3.2.2 Synchronisation von DDL und DML

Die Synchronisation von DDL und DML erfolgt auf zwei Ebenen, auf der Ebene der von der DDL betroffenen Pläne und auf der Ebene der betroffenen Spaces.

SQL-DML-Pläne enthalten Catalog-Information über die in der DML angesprochenen Tabellen und die verwendeten Zugriffswege (Indizes). DDL ändert die Catalog-Information. Wenn eine Tabelle oder ein Index der Tabelle mit DDL/SSL geändert wird, müssen deshalb Pläne, die Catalog-Information über die Tabelle enthalten, invalidiert werden; die Tabelle darf nicht geändert werden, solange solche Pläne existieren. Die entsprechende Synchronisation erfolgt über eine Transaktionssperre auf dem Satz in der Catalog-Tabelle TABLES, der die Tabelle beschreibt. Für die Generierung und Ausführung eines Plans wird eine Shared-Sperre auf dem Satz angefordert. Für die DDL wird eine Exclusive-Sperre auf dem Satz angefordert. Wenn diese Exclusive-Sperre erworben ist, d.h. wenn alle Transaktionen beendet sind, in denen eine Shared-Sperre angefordert wurde, werden die bestehenden Pläne invalidiert und solange die Sperre besteht, also die DDL-Transaktion nicht beendet ist, können keine neuen Pläne generiert werden.

DDL bewirkt nicht nur eine Änderung der Definitionen im Catalog-Space, sondern auch Änderungen in den Strukturen des Anwender-Space (Datei), auf dem das von der DDL betroffene Objekt - Tabelle oder Index - liegt. Insbesondere wegen der Freiplatzverwaltung müssen auch DML-Zugriffe auf andere Objekte des Space mit der DDL-Strukturänderung koordiniert werden. Die Synchronisation dafür erfolgt über Transaktionssperren auf dem Space. Diese Synchronisation wirkt auch bei CALL-DML, für die es die oben genannten Pläne nicht gibt.

Vor der Ausführung der DDL-Anweisung auf einem Space wird eine exklusive Transaktionssperre auf dem Space der Tabelle angefordert. Wenn ein Index betroffen ist (CREATE INDEX oder DROP INDEX) und der Index auf einem anderen Space liegt wie die Tabelle, wird die exklusive Sperre auf beiden Spaces angefordert. Die Anforderung der Sperre bewirkt, dass gewartet wird, bis alle Transaktionen, die Shared- oder Exclusive-Sperren auf dem Space halten, beendet sind. Neue Sperranforderungen auf einen Space, für den die exklusive Sperre angefordert ist, werden in einen Wartezustand versetzt, d.h. die Sperre wird erst zugeteilt, wenn die DDL-Transaktion beendet ist. Für die von der DDL betroffene Tabelle werden auch Betriebsmittel freigegeben, die transaktionsübergreifend gehalten werden (CALL-DML-Open auf die Tabelle und SQL-Cursor, die mit STORE gesichert wurden). Das führt bei Folgezugriffen in den Anwendungen, die diese Betriebsmittel belegt hatten, zu CALL-DML-Status bzw. SQLSTATE.
Per Administrationskommando kann eine langlaufende, eine DDL-Anweisung blockierende Transaktion gegebenenfalls zurückgesetzt werden. Die sperrende Transaktion kann mit dem Performance-Monitor SESMON oder mit SESADM ermittelt werden.

Für die Ausführung einer DDL-Anweisung wird Information aus den Metadaten des Catalog gelesen, um die Autorisierung zu prüfen und um Parameter der Anweisung zu ergänzen und zu plausibilisieren. Nach der Ausführung werden die Metadaten im Catalog aktualisiert. Die dazu nötigen Catalog-Zugriffe werden im höchsten Konsistenzlevel ausgeführt. Hier kann es auch zu Wartesituationen auf Transaktionssperren anderer DDL- oder DML-Anweisungen kommen und möglicherweise auch zu Deadlocks. Bei Deadlock wird eine konkurrierende DML-Anweisung zurückgesetzt. Bei Deadlock zwischen zwei DDL-Anweisungen wird eine der beiden DDL-Anweisungen zurückgesetzt.
Es wird empfohlen, DDL parallel zu DML-Hochlast möglichst zu vermeiden, da Pläne, Cursor und CALL-DML-Betriebsmittel dadurch verloren gehen. Desgleichen sollte parallele DDL auf dieselben Spaces vermieden werden, da sie im besten Fall serialisiert werden, im ungünstigen Fall aber eine der Anweisungen zurückgesetzt wird.

3.2.3 Synchronisation der Utilities

Zunächst zur Synchronisation der Utilities, bei denen nur Catalog-Zugriffe erforderlich sind: MODIFY und CREATE / DROP / ALTER MEDIA DESCRIPTION. Diese Anweisungen werden mit DML-Plänen für den Catalog-Space abgewickelt. Es werden die Synchronisations-Mechanismen der DML verwendet.
Die Synchronisation zwischen den übrigen Utilities einerseits und DDL und DML andererseits erfolgt über Transaktionssperren auf den Spaces (Space Usage Sperre).

Aus Sicht der Anwendung laufen Utility-Anweisungen außerhalb von Transaktionen. Intern werden Transaktionen eröffnet, um die Synchronisation der Utilities über Transaktionssperren zu ermöglichen. Nach Abarbeitung der Utility-Anweisung ist die Transaktion wieder geschlossen.

Die Transaktionssperre kann shared oder exklusiv sein. Eine Shared-Sperre wird bei den Utilities gesetzt, bei denen nur lesend auf den Space zugegriffen wird, das sind: CHECK FORMAL, COPY OFFLINE und UNLOAD. Shared-gesperrt sind außerdem die Tabellen und Indizes eines Space, die bei LOAD OFFLINE und MIGRATE nicht von der Anweisung betroffen sind. Exklusiv ist die Sperre bei REFRESH, RECOVER und REORG und für die Tabelle, die bei LOAD OFFLINE oder MIGRATE angegeben ist. Eine andere Anweisung, die beim Zugriff auf einen Space eine Transaktionssperre vorfindet, wird in einen Wartezustand versetzt. Für die von der Utility betroffenen Tabellen werden auch Betriebsmittel freigegeben, die transaktionsübergreifend gehalten werden (CALL-DML-Open auf die Tabellen und SQL-Cursor, die mit STORE gesichert wurden). Das führt bei Folgezugriffen in den Anwendungen, die diese Betriebsmittel belegt hatten, zu CALL-DML-Status bzw. SQLSTATE.

Für CHECK CONSTRAINTS wird ein SQL-DML-Plan generiert. Dieser Plan wird innerhalb einer Transaktion abgearbeitet. Die Synchronisation erfolgt bei CHECK CONSTRAINTS *integritätsbedingungsname*,... über die Synchronisationsmechanismen der DML. Bei CHECK CONSTRAINTS ON TABLE / ON SPACE wird zuvor eine exklusive Transaktionssperre auf den Spaces angefordert, die in der Anweisung angegeben sind oder auf denen die angegebenen Tabellen liegen.

Bei COPY ONLINE kann auch parallel auf dem Space mit DML geändert werden. Für geänderte Blöcke werden Before Images geschrieben, die nach Beendigung des COPY auf die Sicherung eingespielt werden. DDL/SSL und Utilities, von denen der Space betroffen ist, werden in einen Wartezustand versetzt. Zu einem Zeitpunkt kann nur eine Online-Kopie für einen Catalog aktiv sein.

Bei LOAD ONLINE oder UNLOAD ONLINE können die Tabelle und die ggf. dazu gehörenden Indizes parallel mit DML (oder einem weiteren (UN)LOAD ONLINE) gelesen und geändert werden. LOAD ONLINE verhält sich diesbezüglich wie eine DML-Änderungsanweisung. UNLOAD ONLINE verhält sich diesbezüglich wie eine lesende DML-Anweisung.

Wie bei DDL kann es bei Utilities in den Transaktionsphasen zu Deadlocks kommen. Bei ihrer Auflösung werden die Utilities bevorzugt. Wenn jedoch der Deadlock zwischen zwei Utility-Anweisungen besteht, wird eine Utility abgebrochen, da eine Utility-Anweisung insgesamt nicht rücksetzbar ist. Die von dieser Utility-Anweisung betroffenen Spaces müssen mit RECOVER wiederhergestellt werden. Bei der Überlegung, ob es zu einem Deadlock kommen kann, muss man auch die Catalog-Zugriffe zur Autorisierungsprüfung und Informationsbeschaffung bzw. -aktualisierung berücksichtigen. Parallele Utilities auf denselben Spaces sollten deshalb möglichst vermieden werden.

3.3 Ressourcenverbrauch

In diesem Abschnitt ist beschrieben:

- Welche vorgangs- bzw. auftragsspezifischen Ressourcen es auf Seiten des SQL-Anwenderprogramms gibt.
- Welche Faktoren die Belegung dieser Ressourcen beeinflussen.
- Welche Maßnahmen ergriffen werden können, um einen Ressourcenengpass zu beheben.

3.3.1 Auftragsspezifischer Platzbedarf im Kommunikationspuffer

Anwenderprogramm und DBH verständigen sich (bei Independent-Betrieb) über einen Kommunikationspuffer, der max. 64 KB groß ist. Der aktuell belegte Platz muss also kleiner sein als diese Obergrenze; da außerdem die Sendelänge Einfluss auf die Performance hat, sollte jeweils so wenig wie möglich belegt werden.

Die Größe des benötigten Platzes im Kommunikationspuffer zwischen Anwendungsprogramm und DBH hängt im Wesentlichen ab von:

- Anzahl und Datentyp verwendeter Parameter in der Anweisung
  Zum Ausführungszeitpunkt belegen die Werte der Ein- bzw. Ausgabeparameter (plus Verwaltungsinformation) bei der Übertragung zum bzw. vom DBH Platz im Kommunikationspuffer.
  Durch Modifikation der Länge des Datentyps von Parametern (z.B. CHAR(20) statt CHAR(200)) oder Reduktion der Anzahl von Parametern (z.B. gezielte Angabe von Spalten in der SELECT-Liste statt „SELECT *“) kann gegebenenfalls ein Engpass bei der Ressource Kommunikationspuffer verhindert werden.

- Anweisungstyp
  Abhängig vom Anweisungstyp wird zum Zeitpunkt der Vorübersetzung bzw. Ausführung auch der komplette Anweisungstext (plus Verwaltungsinformation) über den Kommunikationspuffer an den DBH übergeben.
  Zum Zeitpunkt der Vorübersetzung sind dies die DML-Anweisungen SELECT, INSERT, UPDATE, DELETE, MERGE und CALL, die Cursor-Spezifikation bei DECLARE CURSOR und sämtliche DDL-, SSL- und Utility-Anweisungen.
  Zum Zeitpunkt der Ausführung sind dies die Anweisungen SELECT, INSERT, UPDATE, DELETE, MERGE und CALL, die Cursor-Spezifikation bei OPEN, sämtliche DDL-, SSL- und Utility-Anweisungen sowie die zu präparierende Anweisung bei PREPARE bzw. EXECUTE IMMEDIATE.

  Durch Verkürzung des Anweisungstextes (z.B. durch Verwendung permanenter Views, Vermeidung von Kommentaren, Verwendung von Korrelationsnamen, Aufspaltung einer CREATE SCHEMA-Anweisung etc.) kann gegebenenfalls ein Engpass bei der Ressource Kommunikationspuffer verhindert werden.

3.3.2 Vorgangsspezifischer Platzbedarf im VGM

Bei jeder Vorübersetzung und jeder Ausführung eines SQL-Programms wird ein vorgangsspezifischer Speicherbereich (VGM) verwendet, dessen Größe begrenzt ist.

Unter openUTM wird zum PEND-Zeitpunkt der belegte Bereich der VGMs gesichert. Die Größe des Bereichs, den openUTM maximal sichern kann, wird durch den Parameter VGM-SIZE=number der KDCDEF-Steueranweisung MAX bei der Generierung der UTM-Anwendung festgelegt.

Belegung des VGM zur Vorübersetzungszeit

Zum Zeitpunkt der Vorübersetzung einer Übersetzungseinheit werden im VGM im Wesentlichen folgende Informationen abgelegt:

- Spezifikation (Anweisungstext) und Verwaltung statischer Cursor
- Verwaltung dynamischer Cursor und präparierbarer Anweisungen

Durch Verkürzung der Anweisungstexte statischer Cursor (z.B. durch Vermeidung von Einrückungen und Kommentaren, Verwendung von Korrelationsnamen etc.), Verwendung permanenter Views oder Aufteilung in mehrere Übersetzungseinheiten kann gegebenenfalls ein Engpass bei der Ressource VGM verhindert werden.

Belegung des VGM zur Laufzeit

Zum Zeitpunkt der Ausführung einer Anwendung werden im VGM neben diversen Verwaltungsinformationen im wesentlichen folgende Informationen abgelegt:

- SQL-Deskriptorbereiche (Verwaltung und Inhalt):
  Der Platzbedarf für einen Deskriptorbereich ist **nur** vom Wert der WITH MAX-Klausel der ALLOCATE DESCRIPTOR-Anweisung abhängig. Für diese Anzahl von Deskriptor-Einträgen wird sowohl Verwaltungsinformation als auch Platz für Werte durchschnittlicher Länge reserviert.
  Durch gezielte Freigabe oder Wiederverwendung von Deskriptoren kann gegebenenfalls ein Engpass bei der Ressource VGM verhindert werden.
  Durch Reduktion der Anzahl Einträge in einem SQL-Deskriptorbereich (WITH MAX-Klausel) und durch Modifikation der Länge der Datentypen (z.B. CHAR(20) statt CHAR(200)) kann der Platzbedarf für einen SQL-Deskriptorbereich verringert werden.
- Präparierte Anweisungen (Verwaltung):
  Bei präparierten Anweisungen wird nicht der Anweisungstext sondern eine interne Ausführungsvorschrift gespeichert. Der Platz für diesen Code ist vom Typ der präparierten Anweisung sowie der Anzahl der dynamischen Parameter in der Anweisung abhängig. Durch Reduktion der Anzahl von Parametern (z.B. gezielte Angabe von Spalten in der SELECT-Liste statt „SELECT ∗“) oder gezielte Wiederverwendung von Anweisungsnamen kann gegebenenfalls ein Engpass bei der Ressource VGM verhindert werden.

Anmerkung

Bei UTM-Anwendungen könnte ein Engpass auch durch Vergrößerung des VGM (Parameter VGMSIZE bei UTM-Generierung) beseitigt werden. Dadurch werden jedoch alle Teilprogramme der UTM-Anwendung beeinträchtigt (openUTM sichert den belegten Teil des VGM bei jedem Dialogschrittwechsel). Deshalb sollten zunächst teilprogramm-spezifisch die oben genannten Aspekte untersucht werden.

3.3.3 Platzbedarf im Prefetch-Puffer

Arbeiten SQL-Anwender mit Schubmodus, so werden alle Teilergebnismengen (Schübe) von Cursorn, die im Schubmodus eröffnet sind, im Prefetch-Puffer zwischengespeichert. Die Größe des Prefetch-Puffers legt der Anwender in der Konfigurationsdatei der Anwendung über den Parameter PREFETCH-BUFFER fest.

Belegung des Prefetch-Puffers

Der Platz im Prefetch-Puffer wird durch folgende Daten belegt:

- Nutzdaten (Schübe)
- Daten der Speicherverwaltung (Verwaltung der Speicherbereiche für die Nutzdaten)

Der für einen Schub benötigte Speicherbereich hängt von folgenden Faktoren ab:

- Anzahl der Spalten im Cursor
- Datentypen der einzelnen Spalten
- aktuelle Werte der zwischengespeicherten Ergebnissätze
- Anzahl von Ergebnissätzen im Schub.

Der für die Speicherverwaltung benötigte Platz ist abhängig von der Zahl der gleichzeitig zu verwaltenden Schübe (siehe auch Abschnitt „Größe des Prefetch-Puffers“ auf Seite 92).

Informationen über die Auslastung des Prefetch-Puffers bietet der Performance-Monitor SESMON in der Maske „PREFETCH-BUFFERS“ (siehe Handbuch „Datenbankbetrieb“).

Ist die Dimensionierung des Prefetch-Puffers zu gering ausgefallen, so kann der Anwender den Wert des Parameters PREFETCH-BUFFER in der Konfigurationsdatei vergrößern. Der neue Wert wird beim nächsten Neustart der Anwendung wirksam.

3.4 Einsatz von Routinen

Routinen (Prozeduren (Stored Procedures) und User Defined Functions (UDF)) enthalten Folgen von SQL-Anweisungen, die in einer Datenbank gespeichert und verwaltet werden.

Routinen und ihre Verwendung in SESAM/SQL sind detailliert im Handbuch „SQL-Sprachbeschreibung Teil 1: SQL-Anweisungen“ beschrieben.

In diesem Abschnitt werden die performance-relevanten Aspekte des Einsatzes von Routinen erläutert.

Durch den Einsatz von Routinen kann eine Folge von SQL-Anweisungen im DBH ohne Kommunikation mit dem Anwenderprogramm durchgeführt werden.

Dies erspart den **Kommunikationsaufwand** für die einzelnen Anweisungen und führt zu verkürzten **Laufzeiten**. Dieser Effekt tritt besonders dann auf, wenn die Anwendung nicht auf dem gleichen Rechner wie der DBH abläuft.

Bei kurzen Anweisungen ist die Kommunikation ein wesentlicher Anteil an der Laufzeit einer Anweisung. Wenn z.B. ein Cursor abgearbeitet wird, so sind die FETCH-Anweisungen zumeist von relativ kurzer Dauer, erfordern aber trotzdem die Kommunikation mit dem Anwenderprogramm. Die Einschalung einer Cursor-Verbeitung in eine Routine wirkt positiv auf die Performance sofern keine Kontakte zum Anwenderprogramm erforderlich sind.

Zu beachten ist, dass in Routinen kein COMMIT WORK (und auch kein ROLLBACK) möglich ist und die Transaktion deshalb den gesamten Ablauf der Routine umfasst. Dies ist bei der Definition von Routinen zu berücksichtigen.

Durch den Einsatz von Routinen ergeben sich auch Einsparungen an **CPU-Zeit**. Privilegprüfungen müssen nur für das EXECUTE-Privileg der Routine durchgeführt werden. Es wird nur ein Plan bereitgestellt. Ein-/Ausgabewerte werden nur einmal pro Routine aufbereitet.

Zu beachten ist, dass durch die Verwendung von Routinen zu Gunsten der Anwendung ein zusätzlicher Aufwand im DBH entsteht. Die Logik (Kontrollanweisungen) eines Vearbeitungsprozesses läuft mit der Routine im DBH ab. Ohne Routine würde sie in der Anwendung ablaufen.

Ein performance-neutraler, aber sinnvoller Aspekt von Routinen ist die Einschalung von Arbeitsabläufen. Solange sich die Schnittstelle einer Routine nicht ändert, kann die darin enthaltene Verarbeitung beliebig geändert werden, ohne dass das Anwendungsprogramm geändert, übersetzt oder gebunden werden muss.

4 Performance-relevante Aspekte des Datenbankentwurfs

Dieses Kapitel befasst sich mit dem Plattenspeicherbedarf von Primär-, Sekundär- und Metadaten. Es beschreibt außerdem performance-relevante Aspekte, die bei der Verwendung von Integritätsbedingungen und bei der Datenverteilung und -allokation zu beachten sind.

4.1 Spaces mit einer Größe von mehr als 64 GByte

Anwender-Spaces (und auch der Catalog-Space) können bis zu 4 TByte groß werden („großer Space“), wenn sie mit mit SESAM/SQL ab V7.0 auf einem Pubset, das „große Dateien“ unterstützt, angelegt werden. Dies geschieht mit mit CREATE SPACE, CREATE CATALOG, REORG SPACE oder RECOVER TO.
Sonst können Spaces bis zu 64 GByte groß werden.

Auf großen Spaces können Sie Tabellen oder Indizes mit sehr großem Datenvolumen speichern. Eine nicht-partitionierte Tabelle kann so bis zu 4 TByte groß werden.
Eine partitionierte Tabelle mit 16 Partitionen kann so bis zu 64 Terabyte groß werden.

Ein großer Space besteht aus maximal 1.073.741.822 Blöcken. Alle Verweise auf Blocknummern werden in einem Feld mit vier Byte Länge gespeichert. Dadurch vergrößert sich der Platzbedarf einer Datenbank geringfügig.

Beispiele

Platzbedarf einer Basistabelle mit 250 Mio. Sätzen und einer durchschnittlichen Bruttosatzlänge von 2.000 Bytes. Länge des Primärschlüssels: 20 Byte, Density: 80%.

Primärdatenblöcke	ca. 155 Mio. 4KB-Blöcke
Primärschlüsselindexblöcke	ca. 1,3 Mio. 4KB-Blöcke
Blöcke für Platten-Satznummern-Zuordnungstabelle	ca. 0,25 Mio. 4KB-Blöcke
Gesamt	ca.156,55 Mio. 4KB-Blöcke = 597 GByte

Platzbedarf für einen Sekundärindex mit 1 Mio. Werten, 250 Treffer pro Wert und 20 Byte Länge, 80% Density.

Sekundärindex-Grundstufe	166.667 4KB-Blöcke
Sekundärindex-Indexstufe	1.254 4KB-Blöcke
Statistikblock	1 4KB-Block
Gesamt	167.922 4KB-Blöcke = 656 MByte

Platzbedarf eines Sekundärindex für einen Unique-Index mit 250 Mio. Werten, 1 Treffer pro Wert und 20 Byte Länge, 80% Density.

Sekundärindex-Grundstufe	2,57 Mio. 4KB-Blöcke
Sekundärindex-Indexstufe	19.381 4KB-Blöcke
Statistikblock	1 4KB-Block
Gesamt	ca. 2,6 Mio. 4KB-Blöcke = 9,9 GByte

Die Sicherungs- und Recoveryzeiten hängen sowohl von der Größe des Spaces als auch von der verwendeten Peripherie ab. Bei sehr großen Datenvolumen ist zu überlegen, ob nicht eine partitionierte Tabelle mit mehreren Partitionen besser geeignet ist als eine einzige große Tabelle.

Grundlegende Informationen zu nicht-partitionierten und partitionierten Tabellen finden Sie im „Basishandbuch“. Performance-Informationen zu partitionierten Tabellen finden Sie im Abschnitt „Partitionierte Tabellen“ auf Seite 67.

4.2 Primärdaten

Der Plattenspeicherplatzbedarf für die Primärdaten kann wie folgt berechnet werden:

Blockgröße	4096
Blockkopf	52
Block-Checkinfo	4
Satzkopf	17
pro signifikanter Wert	3
Länge CHAR	Länge ohne Blanks am Ende
Länge VARCHAR	≤ 253 Byte --> Länge Wert + 3 > 253 Byte --> ([Länge Wert/4040] + 1) * 4KB
Länge NCHAR	(Länge in Code Units ohne National-Blanks am Ende) * 2
Länge NVARCHAR	Länge in Code Units ≤ 126 --> Länge in Code Units * 2 + 3 Länge in Code Units > 126 --> ([Länge in Code Units * 2 / 4040] + 1) * 4KB
Länge NUMERIC	Anzahl Stellen ohne führende Nullen
Länge DECIMAL	[Anzahl Stellen/2], wobei [...] bedeutet „nächst-größere ganze Zahl“ (bei Spaces werden DECIMAL-Werte ohne führende Nullen abgespeichert)
Länge INTEGER	4
Länge SMALLINT	2
Länge REAL	4
Länge DOUBLE PRECISION	8
Länge TIME	8
Länge DATE	6
Länge TIMESTAMP	14

Tabelle 2: Plattenspeicherbedarf für die Primärdaten

Eine Sonderrolle beim Speicherplatzbedarf nehmen die BLOBs ein. Ein BLOB-Wert besteht aus mehreren Sätzen, die in einer speziell strukturierten Tabelle, der sogenannten BLOB-Tabelle, abgespeichert werden (siehe Handbuch „SQL-Sprachbeschreibung Teil 1: SQL-Anweisungen“).

Eine BLOB-Tabelle besteht grundsätzlich aus den folgenden Spalten:

- OBJ_NR vom Datentyp INTEGER
- SLICE_NR vom Datentyp INTEGER
- SLICE_VAL vom Datentyp VARCHAR(31000)

- OBJ_REF vom Datentyp CHAR(237)

OBJ_NR und SLICE_NR bilden den Primärschlüssel der BLOB-Tabelle.

Der BLOB-Wert wird innerhalb der Tabelle durch OBJ_NR identifiziert. Zu jedem BLOB-Wert gibt es einen Satz mit SLICE_NR 0 (siehe Beispiel). Dieser Satz enthält:

- in der Spalte SLICE_VAL die Attributbeschreibung (kann zum Beispiel bei CREATE TABLE OF BLOB über MIME oder USAGE angegeben werden)
- in der Spalte OBJ_REF den REF-Wert dieses BLOBs. Die Länge von OBJ_REF hängt vom aktuellen Tabellennamen ab und liegt zwischen 46 und 237 Byte.

Ferner gibt es zu jedem BLOB-Wert einen Satz mit SLICE_NR 2147483647 (Highvalue), der lediglich den Primärschlüssel beinhaltet. Die Länge dieses Satzes beträgt genau 28 Byte.

Der BLOB-Wert selbst ist in Portionen von maximal 31000 Byte in der Spalte SLICE_VAL in den Sätzen ab SLICE_NR 1 enthalten. Die Spalte OBJ_REF ist in diesen Sätzen nicht signifikant.

Das BLOB-Objekt belegt, außer in der BLOB-Tabelle selbst, auch noch in der referenzierenden Tabelle den Speicherplatz für den REF-Wert. Dieser REF-Wert wird in einer mit „FOR REF“ definierten Spalte gespeichert. Die Länge dieses Wertes beträgt zwischen 46 und 237 Byte.

Beispiel

Ein BLOB-Wert hat eine Länge von 100.000 Byte. Der zugehörige REF-Wert ist 70 Byte lang, die Attributbeschreibung 150 Byte. In der BLOB-Tabelle wird dieser Wert in Form von 6 Sätzen dargestellt:

OBJ_NR	**SLICE_NR**	**SLICE_VAL**	**OBJ_REF**
nnn	0	Attributbeschreibung	REF-Wert
nnn	1	BLOB-Wert (31000 Byte)	NULL
nnn	2	BLOB-Wert (31000 Byte)	NULL
nnn	3	BLOB-Wert (31000 Byte)	NULL
nnn	4	BLOB-Wert (31000 Byte)	NULL
nnn	2147483647	NULL	NULL

Tabelle 3: Beispiel für eine BLOB-Tabelle

Gemäß der Regeln für VARCHAR belegt SLICE_VAL in den Slices 1 bis 3 jeweils 8 * 4 KB, im Slice 4 dagegen 2 * 4 KB. Rechnet man die übrigen Spalten und Sätze hinzu, dann belegt der BLOB in der BLOB-Tabelle selbst insgesamt 104 KB + 397 Byte, in der referenzierenden Tabelle 73 Byte.

4.3 Metadaten

In der aktuellen Version von SESAM/SQL gibt es insgesamt 28 Systemtabellen, die eine Datenbank und die Anwenderdaten beschreiben.

Selbstbeschreibung der Datenbank	ca. 1500	4KB-Seiten
pro Space	ca. 400	Byte
pro Tabelle	ca. 1300	Byte
pro Spalte	ca. 700	Byte
pro Index	ca. 300	Byte

Tabelle 4: Plattenspeicherbedarf für die Metadaten

Beispiel

10 Tabellendefinition mit 100 Spalten benötigen ca. 700 KByte.
200 Indexdefinitionen benötigen ca. 60 KByte.

4.4 Integritätsbedingungen

Durch Integritätsbedingungen können Einschränkungen für die zulässigen Werte von Spalten angegeben werden. Es werden folgende Typen von Integritätsbedingungen unterschieden:

- Referenzbedingung
- Check-Bedingung
- Nicht-NULL-Bedingung
- Eindeutigkeitsbedingung
- Primärschlüsselbedingung

Nach einer Änderung (z.B. mit den SQL-Anweisungen INSERT, UPDATE oder DELETE) müssen die Daten den durch die Integritätsbedingungen definierten Einschränkungen genügen. In SESAM/SQL werden nur diejenigen Integritätsbedingungen geprüft, die von der Datenänderung tatsächlich direkt betroffen sind. In Betracht kommen alle Integritätsbedingungen, die auf der geänderten Tabelle definiert sind, und zusätzlich alle Referenzbedingungen, an denen diese Tabelle (als referenzierende oder referenzierte Tabelle) beteiligt ist.

Im Folgenden wird erläutert, wie die Prüfung von Integritätsbedingungen erfolgt und was dabei aus Performance-Gesichtspunkten zu beachten ist. Generell kann man folgende Aussagen machen:

- Die Primärschlüsselbedingung sollte möglichst für jede Tabelle definiert werden. Hier existieren keine Performance-Nachteile, vielmehr können durch den Primärschlüsselindex viele Anfragen sehr effizient abgearbeitet werden.
- Sonst sollten Integritätsbedingungen bei der Schemadefinition gezielt und sparsam verwendet werden. Es sollten möglichst nur solche Integritätbedingungen definiert werden, die für die Konsistenz der Daten wesentlich sind und ansonsten im Rahmen der Verarbeitungslogik eines Anwendungsprogramms geprüft würden.

Referenzbedingung

Im Folgenden wird das Vorgehen bei der Prüfung von Referenzbedingungen beschrieben. Außerdem wird für bestimmte Fälle eine Möglichkeit gezeigt, bei der Prüfung einer Referenzbedingung die Performance durch Definition eines Index zu verbessern.

Für eine Referenzbedingung wird z.B. in der Tabelle AUFTRAG ein Fremdschlüssel definiert, dem Spalten einer zweiten Tabelle KUNDE (referenzierte Tabelle) zugeordnet sind (z.B. ... FOREIGN KEY kunden_nr REFERENCES KUNDE (nr) ...).

Bei der Prüfung einer Referenzbedingung wird unterschieden, ob die referenzierende oder die referenzierte Tabelle geändert wird (Update einer Spalte, Löschen bzw. Einfügen von Sätzen). Die folgende Übersicht zeigt, wie eine (nicht-selbstreferenzierende) Referenzbedingung bei den verschiedenen Anweisungen geprüft wird.

Zur Bezeichnung:

alt bezeichnet den Wert vor der Änderung

neu bezeichnet den neuen Wert

Beispiel

Prüfung einer Referenzbedingung bei einer Änderung der referenzierenden Tabelle AUFTRAG:

1. UPDATE von AUFTRAG.kunden_nr
   Für jeden geänderten Satz von AUFTRAG wird geprüft:
   Existiert ein Satz in der Tabelle KUNDE mit
   KUNDE.nr = AUFTRAG.kunden_nr(neu)
2. DELETE
   keine Prüfung
3. INSERT
   Für jeden in AUFTRAG eingefügten Satz wird geprüft:
   Existiert ein Satz in der Tabelle KUNDE mit
   KUNDE.nr = AUFTRAG.kunden_nr(neu)

Beispiel

Prüfung einer Referenzbedingung bei einer Änderung der referenzierten Tabelle KUNDE:

1. UPDATE von KUNDE.nr
   Für jeden Satz von AUFTRAG wird geprüft:
   Existiert ein Satz in der Tabelle KUNDE mit KUNDE.nr = AUFTRAG.kunden_nr

2. UPDATE ... WHERE CURRENT OF von KUNDE.nr
   Für den geänderten Satz von KUNDE wird geprüft:
   Existiert in der Tabelle AUFTRAG ein Satz mit AUFTRAG.kunden_nr = KUNDE.nr(alt)
3. DELETE und es existiert kein Sekundärindex für AUFTRAG.kunden_nr
   Für jeden Satz von AUFTRAG wird geprüft:
   Existiert ein Satz in der Tabelle KUNDE mit KUNDE.nr = AUFTRAG.kunden_nr
4. DELETE und es existiert ein Sekundärindex für AUFTRAG.kunden_nr
   Für jeden zu löschenden Satz von KUNDE wird geprüft: Existiert ein Satz in der Tabelle AUFTRAG mit AUFTRAG.kunden_nr = KUNDE.nr
5. INSERT
   keine Prüfung

Die Suche nach einem Satz in der Tabelle KUNDE, wobei der Wert der Spaltennummer vorgegeben ist, kann immer performant über den Primärschlüsselindex erfolgen.
Der Vergleich von (3) und (4) zeigt, dass die Prüfung einer Referenzbedingung bei einer DELETE-Anweisung auf die referenzierte Tabelle dadurch wesentlich unterstützt werden kann, dass zuvor für die referenzierenden Spalten ein Sekundärindex definiert wird.

Check- und Nicht-NULL-Bedingung

Eine Check- bzw. Nicht-NULL-Bedingung wird sofort (on-the-fly) bei Änderung jedes einzelnen Satzes geprüft. Daher ist die Prüfung dieser Integritätsbedingungen nicht performance-kritisch.

Eindeutigkeitsbedingung

Für eine Eindeutigkeitsbedingung richtet der DBH automatisch einen geeigneten Index ein. Dieser Index kann ggf. auch vom Anwender explizit definiert worden sein.
Die Prüfung einer Eindeutigkeitsbedingung erfolgt mit der Aktualisierung des Index bei einer SQL-Änderungsanweisung (INSERT, UPDATE, DELETE, MERGE). Es fallen daher gegenüber einem normalen Index keine zusätzlichen Kosten an.

Primärschlüsselbedingung

Die Prüfung einer Primärschlüsselbedingung erfolgt implizit mit der Änderung eines Satzes in der Basistabelle und ist daher nicht performance-kritisch.

4.5 Datenverteilung und -allokation

Überlegungen zur Verteilung der Daten auf die zur Verfügung stehenden Plattengeräte müssen für alle Dateien getroffen werden, auf die in einer SESAM/SQL-Session zugegriffen wird. Also Datenbankdateien (Catalog-Space und Anwender-Spaces), Sicherungskopien für die Datenbankdateien, DA-LOG-Dateien, CAT-LOG-Dateien, CATREC-Datei und die DBH-Dateien, WA-LOG-Datei, TA-LOG-Dateien, Cursor-Dateien und temporäre Arbeitsdateien.

Für die Verteilung der Daten sind folgende Gesichtspunkte maßgeblich.

- Performante I/O
- Sicherungskonzept
- Sperrgranulat bei DDL, SSL und Utilities
- Allokation von Indizes

Performante I/O

TA-LOG-Dateien sind Ein-/Ausgabe-intensiv, in Ausnahmefällen kann es zu Ein-/Ausgabe-Engpässen kommen. TA-LOG-Dateien sollten deshalb auf einem eigenen, möglichst schnellen Gerät liegen.
Die Größe der Schreibeinheit für die TA-LOG-Dateien ist abhängig vom Plattentyp. SESAM/SQL nutzt die maximal mögliche Ein-/Ausgabe-Länge (64 bis 160 KB).
Information über die maximale Ein-/Ausgabe-Länge in Half-Pages (2KB) erhalten Sie mit dem BS2000-Kommando

```
/SHOW-PUBSET-CONFIGURATION PUBSET=<catid>,INFORMATION=*PUBSET-FEATURES.
```

Wenn (mit dem Performance Monitor) nennenswerte Schreibzugriffe auf die WA-LOG-Datei zu beobachten sind, dann sollte auch die WA-LOG-Datei auf einem eigenen, möglichst schnellen Gerät liegen. Zuvor sollte aber versucht werden, diese Zugriffe auf die WA-LOG-Datei zu verringern, indem die Einstellungen für User- und System-Data-Buffer vergrößert werden (siehe Abschnitt „User-Data-Buffer dimensionieren“ auf Seite 74 und Abschnitt „System-Data-Buffer dimensionieren“ auf Seite 75)

Die Tabellen und Indizes müssen so auf Spaces und die Spaces auf die Geräte verteilt werden, dass die Geräte gleichmäßig ausgelastet sind. Die Verteilung hängt von der Anwendung ab, z.B.:

- Wenn eine Tabelle groß ist, häufig auf die Tabelle zugegriffen wird, und die Zugriffe einigermaßen gleichmäßig über die Tabelle verteilt sind, ist es sinnvoll, diese Tabelle in einen eigenen Space zu legen und den Space (die Datei) auf mehrere Geräte zu verteilen.

- Bei gleichmäßiger Verteilung der Zugriffe auf verschiedene Tabellen, sollten die Tabellen ebenfalls in verschiedene Spaces und die Spaces dann auf verschiedene Geräte gelegt werden.

Bei DA-LOG-, CAT-LOG- und CATREC-Datei ist die Allokation hauptsächlich durch das Sicherungskonzept bestimmt. Alle drei Dateien können auf demselben Gerät liegen. CAT-LOG- und CATREC-Datei sind hinsichtlich I/O unproblematisch.
I/O auf temporäre Arbeits- und Cursor-Dateien kann vermieden werden, wenn die Puffer über die DBH-Optionen groß genug eingestellt werden. Ist das nicht mehr möglich, sollten die temporären Arbeitsdateien auf ein schnelles Gerät gelegt werden.

Sicherungskonzept

Damit bei einem Plattenausfall die Daten wiederhergestellt werden können, müssen die Daten (Catalog-Space, Anwender-Spaces) auf anderen Geräten liegen als die Sicherungsdateien (CATREC-Datei, CAT-LOG-Dateien, DA-LOG-Dateien und Sicherungskopien der Spaces) (siehe auch Handbuch „SQL-Sprachbeschreibung Teil 2: Utilities“).

Sperrgranulat bei DDL, SSL und Utilities

Das Sperrgranulat bei DDL, SSL und Utilities ist im Allgemeinen der Space und damit eine Datei. Falls Tabellen voneinander unabhängig sind, z.B. keine Referenzbeziehungen zwischen ihnen bestehen, kann man die Parallelität dadurch erhöhen, dass man sie auf unterschiedliche Spaces legt. DDL, SSL oder Utility auf eine Tabelle behindert dann nicht den Zugriff auf eine Tabelle in einem anderen Space.

Allokation von Indizes

Unter dem Gesichtspunkt des Sperrgranulats kann es sinnvoll sein, Indizes zu einer Tabelle auf denselben Space zu legen, auf dem die Tabelle liegt. Unter dem Performance-Gesichtspunkt ist es wahrscheinlich günstiger, sie auf Spaces zu allozieren, die auf anderen Geräten liegen.
Hinsichtlich des Recovery sollten Tabelle und zugehörige Indizes auf einem Space-Set liegen, d.h. einer Menge gemeinsam gesicherter Spaces, die dann auch gemeinsam wiederhergestellt werden können.
Beim Recovery gibt es noch einen weiteren Aspekt. Bei der logischen Datensicherung werden auch Index-Änderungen protokolliert, die beim Recovery auf einem Sicherungsstand nachgefahren werden. Die Protokollierung kostet CPU- und I/O-Zeit. Die Indexprotokollierung kann man umgehen, wenn ein Index auf einem Space liegt, für den die logische Protokollierung ausgeschaltet ist. Bei RECOVER muss dann allerdings nach der Reparatur des Space, auf dem die Tabelle liegt, der Index neu aufgebaut werden.

4.6 Partitionierte Tabellen

Sie können Basistabellen auf mehr als einen Anwender-Space verteilen. Diese Tabellen werden als partitionierte Tabellen bezeichnet. Grundlegende Informationen zu partitionierten Tabellen finden Sie im „Basishandbuch“.

Die Partitionen einer partitionierten Tabelle können auf bis zu 16 Anwender-Spaces verteilt werden (eine nicht-partitionierte Tabelle wird auf einem Anwender-Space gespeichert). Die Größe aller Anwender-Spaces einer partitionierten Tabelle entspricht ungefähr der Größe des Anwender-Spaces der entsprechenden nicht-partitionierten Basistabelle, wenn auf den Anwender-Spaces keine weiteren Tabellen, Partitionen oder Indizes gespeichert sind.

Partitionierte Tabellen werden meist dazu verwendet, Anwenderdaten über den Primärschlüssel nach bestimmten Kriterien zu strukturieren, z.B. monatsweise. Eine Partition umfasst dann die Anwenderdaten eines Monats. In vielen Fällen wird nur auf die Daten des aktuellen Monats zugegriffen. Die Daten der vorangehenden Monate müssen nur aus Revisionsgründen aufgehoben werden. Auf diese Daten wird aber in der Regel nicht zugegriffen.
Mit Hilfe von partitionierten Tabellen kann der Datenbestand, auf den aktuell zugegriffen wird, verkleinert werden.

Die kleinste Sicherungs- und Reparatureinheit in SESAM/SQL ist ein Anwender-Space. Für eine partitionierte Tabelle reduziert sich die für Sicherung und Reparatur benötigte Zeit, da mehrere (kleinere) Anwender-Spaces mit einer Anweisung parallel gesichert oder repariert werden können.
Ist z.B. nur eine Partition von einer Reparatur betroffen, so kann gezielt der dazu gehörende Anwender-Space repariert werden.

Auch bei einer partitionierten Tabelle kann der Primärschlüssel oder ein Teil des Primärschlüssels als „automatisches Zählfeld“ (COUNTING_FIELD) genutzt werden,
siehe „Basishandbuch“.
Das alleinstehende Zählfeld bzw. der konstante Teil des Primärschlüssels zusammen mit dem Zählfeld wird als „Primärschlüsselbereich“ bezeichnet. Um für einen Primärschlüsselbereich den Wert des Zählfeldes bei der INSERT- und LOAD ONLINE-Anweisung zu bestimmen, wird ausgehend vom größten möglichen Primärschlüsselwert in diesem Primärschlüsselbereich rückwärts der bisher höchste verwendete Primärschlüsselwert gesucht. Auf den Wert im Zählfeld wird 1 addiert. Zusammen mit dem konstanten Teil ergibt sich so der neue Primärschlüsselwert.
Falls ein Primärschlüsselbereich über Partitionsgrenzen geht, kann also die Bestimmung des neuen Werts für ein automatisches Zählfeld einen höheren Aufwand verursachen, da mehrere Partitionen gelesen werden müssen.

5 Performance-relevante Aspekte des Datenbankbetriebs

Die folgenden Abschnitte beschreiben Aspekte des Datenbankbetriebs, die die Performance beeinflussen können, und auf die der Systemverwalter beim Starten oder im laufenden Betrieb der DBH-Session steuernd einwirken kann.

5.1 Diagnose und Maßnahmen bei Performance-Problemen

Tabelle 5 zeigt in einer Übersicht, welche Ursachen Performance-Problemen zu Grunde liegen können, welche Diagnosemöglichkeiten es gibt und wie der Systemverwalter das jeweilige Problem beheben kann.

Mögliche Ursache	Diagnosemöglichkeit	Maßnahme
Prefetch-Puffer zu klein	SESMON-Maske PREFETCH-BUFFERS: Hitraten (bei Speicheranforderungen) < 100%	Parameter PREFETCH-BUFFER in der Konfigurationsdatei erhöhen
SQL-Planpuffer zu klein	SESMON-Maske SQL INFORMATION (Plans)	DBH-Option SQL-SUPPORT/ Parameter PLANS erhöhen
Transfer-Container zu klein	SESMON-Maske SYSTEM INFORMATION	DBH-Option anpassen (im laufenden Betrieb mit MODIFY-STORAGE-SIZE)
Work-Container zu klein	SESMON Maske SYSTEM INFORMATION	DBH-Option anpassen (im laufenden Betrieb mit MODIFY-STORAGE-SIZE)
Schubmodus bei Prefetch-Cursor unterbleibt	Folgende Punkte prüfen: 1. DECLARE CURSOR muss ein (korrektes) Pragma --% PREFETCH *n* enthalten 2. DECLARE CURSOR darf keine FOR UPDATE-Klausel enthalten 3. Die Konfigurationsdatei der Anwendung muss den Parameter PREFETCH-BUFFER enthalten 4. Die maximale Nachrichtenlänge muss ausreichen, um mehr als einen Satz zu transportieren	1. und 2.: SQL-Anwendung ändern 3.: Parameter angeben 4.: Nachrichtenlänge erhöhen
Sperrkonflikte durch unnötige Indizes (z.B. Index: Geschlecht; mögliche Werte: männlich bzw. weiblich)	SESMON-Maske TRANSACTIONS (Locked Transactions), View SYS_INFO_SCHEMA. SYS_LOCK_CONFLICTS	DROP auf den Index, der keine Selektivität besitzt

Tabelle 5: Ursache, Diagnose und Maßnahme bei Performance-Problemen (Teil 1 von 2)

Mögliche Ursache	Diagnosemöglichkeit	Maßnahme
Sperrkonflikte durch zu große TA-Klammern	SESMON-Maske TRANSACTIONS, View SYS_INFO_SCHEMA. SYS_LOCK_CONFLICTS	Versuchen, TA-Klammern zu verkleinern; prüfen, ob das Konsistenzniveau angemessen ist
Sperrkonflikte durch parallele Utilities bzw. parallele DDL	SESMON-Maske TRANSACTIONS, View SYS_INFO_SCHEMA. SYS_LOCK_CONFLICTS	Vermeiden von Utilities oder DDL zu Hochlastzeiten
Frontmaskierung führt in der Regel zu Performance-Verlusten bei der Indexverarbeitung	SESCOS - SESCOSP Ein-/Ausgabe-Statistik: Viele Zugriffe auf den System-Data-Buffer; View SYS_DML_RESOURCES des SYS_INFO_SCHEMA	Frontmaskierung vermeiden oder weitere einschränkende Bedingungen vereinbaren
I/O Wartezeiten zu lang	BS2000 Software Monitor openSM2; View SYS_DML_RESOURCES des SYS_INFO_SCHEMA	Plattengerät entlasten
Physikalische I/Os häufig, obwohl ständig auf gleiche Daten zugegriffen wird	SESMON-Maske I/O; View SYS_DML_RESOURCES des SYS_INFO_SCHEMA	DBH-Optionen SYSTEM- und USER-DATA-BUFFER bzw. BUFFER STRATEGY prüfen
Sekundärindex defekt	INFORMATION_SCHEMA bzw. SYS_INFO_SCHEMA (View INDEXES bzw. SYS_INDEXES)	Utility-Anweisung RECOVER INDEX
SQL-Pläne werden nicht wiederverwendet, obwohl gleiche Anweisungen gesendet werden	SESMON-Maske SQL INFORMATION (Plans): Pläne werden generiert, obwohl noch Platz im Planpuffer.	evtl. gleiche Anweisungen aus unterschiedlichen Übersetzungseinheiten --> Module neu strukturieren

Tabelle 5: Ursache, Diagnose und Maßnahme bei Performance-Problemen (Teil 2 von 2)

5.2 Work-Container

Der Performance-Monitor SESMON gibt in der Maske „SYSTEM INFORMATION“ Hinweise zur Qualität der Work-Container-Einstellung. Eine ungünstige Einstellung kann im laufenden Betrieb mit der Administrationsanweisung MODIFY-STORAGE-SIZE oder beim nächsten DBH-Kaltstart über die DBH-Option WORK-CONTAINER korrigiert werden.

SESMON gibt Folgendes aus: Die aktuelle Containergröße und die aktuelle relative Auslastung sowie die maximale Größe laut Options und die bisherige maximale Auslastung relativ zur Option.

Bei einer geringen relativen Auslastung kann der Work-Container verkleinert werden.

Bei einer Auslastung sehr nahe an 100% ist es möglich, dass Anweisungen wegen Platzmangel im Work-Container nicht ausgeführt werden können und mit einem SQLSTATE bzw. CDML-Status abgebrochen werden. In diesem Fall sollte der WORK-CONTAINER vergrößert werden.

5.3 Transfer-Container

Der Performance-Monitor SESMON gibt in der Maske „SYSTEM INFORMATION“ Hinweise zur Qualität der Transfer-Container-Einstellung. Eine ungünstige Einstellung kann im laufenden Betrieb mit der Administrationsanweisung MODIFY-STORAGE-SIZE oder beim nächsten DBH-Kaltstart über die DBH-Option TRANSFER-CONTAINER korrigiert werden.

SESMON gibt Folgendes aus: Die aktuelle Containergröße und die aktuelle relative Auslastung sowie die maximale Größe laut Options und die bisherige maximale Auslastung relativ zur Option.

Bei einer geringen relativen Auslastung kann der Container verkleinert werden.

Bei einer Auslastung sehr nahe an 100% ist es möglich, dass Anweisungen wegen Platzmangel im TRANSFER-CONTAINER nicht ausgeführt werden können und mit einem SQLSTATE bzw. CDML-Status abgebrochen werden. In diesem Fall sollte der TRANSFER-CONTAINER vergrößert werden.

5.4 User- und System-Data-Buffer

Durch die Puffertechnik des DBH ist es möglich, Blöcke, die im Laufe einer Session öfters benötigt werden, nur einmal physikalisch von der Platte zu lesen und bei Bedarf verfügbar zu haben. Da in der Regel nicht alle Blöcke einer Datenbank resident gehalten werden können, sorgt ein Verdrängungsmechanismus dafür, dass seltener benutzte Blöcke bei Puffer-engpass überschrieben werden.

Die Güte der Puffereinstellung drückt sich in der Hitrate aus, also dem Prozentsatz der im Puffer vorgefundenen Blöcke. Performance-steigernd wirkt sich ein Puffer erst dann aus, wenn ein Block vor seiner Verdrängung öfters benötigt wird. Die Wahrscheinlichkeit, mit der ein bestimmter Block mehrmals angefordert wird, hängt einerseits von der Anwendung ab, andererseits von der Funktion des Blockes (Indexblock, Verwaltungsblock, Anwenderdatenblock). So werden Index-Blöcke wesentlich häufiger benötigt als Anwenderdatenblöcke.

Bei einer sequenziellen Suche über einen großen Anwenderdatenbestand könnten alle Indexblöcke verdrängt werden. Um dieses zu verhindern, unterscheidet SESAM/SQL zwischen dem User-Data-Buffer, in dem die Anwenderdaten der Tabellen hinterlegt werden und dem System-Data-Buffer, in dem alle Zugriffs- und Verwaltungsdaten abgelegt werden.

Über die DBH-Optionen SYSTEM-DATA-BUFFER und USER-DATA-BUFFER kann der Systemverwalter die jeweilige Puffergröße beim Starten des DBH bedarfsgerecht dimensionieren. Mit dem Administrationskommando RECONFIGURE-DBH-SESSION kann er die jeweilige Puffergröße im laufenden DBH-Betrieb anpassen.

5.4.1 User-Data-Buffer dimensionieren

Normalerweise lässt sich keine große Hitrate im User-Data-Buffer erreichen, da in der Regel der Zugriff auf die gleichen Anwenderdaten selten ist. Anzustreben ist, dass bei READ- und WRITE-Operationen die gelesenen Blöcke zum Zeitpunkt des WRITE noch im Buffer resident sind. Dies führt zu der vereinfachten Rechnung:

`Anzahl benötigter Blöcke =` $n * m$

wobei die Platzhalter folgende Bedeutung haben:

n: Anzahl der in einer Sekunde eingelesenen Anwenderdaten-Blöcke

m: Anzahl der Sekunden, die zwischen dem Lesen und dem Rückschreiben des Blockes vergehen.

Zusätzlich ist noch ein kleiner Sicherheitszuschlag notwendig, da n und m keine scharfen Größen sind.

Residenthalten von Tabellen

Sollen kleine, häufig benutzte Tabellen bzw. Tabellenausschnitte im User-Data-Buffer resident gehalten werden, so ist ihre Größe auf die oben genannte Formel zu addieren.
Der Platzbedarf eines Satzes kann mit Hilfe von Tabelle 2 auf Seite 59 berechnet werden, indem 17 Byte für den Satzkopf, 3 Byte pro signifikanter Wert und die Wertelängen der einzelnen Spalten abhängig vom Datentyp addiert werden.

Maximal stehen 4040 Byte pro Block zur Verfügung.

Vermeiden unnötiger I/Os auf die WA-LOG-Datei

Wird ein Block vorzeitig, also vor dem nächsten Konsistenzpunkt, aus dem Puffer verdrängt, so schreibt der DBH zunächst das physikalische Before-Image auf die WA-LOG-Datei. Um unnötige I/Os auf die WA-LOG-Datei zu vermeiden, sollte der User-Data-Buffer groß genug sein, die Anwenderdatenblöcke aller parallelen Transaktionen (physikalische Before-Images und After-Images) aufzunehmen.

5.4.2 System-Data-Buffer dimensionieren

Ziel der Dimensionierung des System-Data-Buffer sollte es sein, eine möglichst hohe Hitrate zu erreichen. Falls möglich, sollten ganze Indizes resident gehalten werden, ist das nicht möglich, so doch wenigstens deren Zugriffspfade.

Die Größe eines Sekundärindex (SI) lässt sich folgendermaßen grob abschätzen:

Anzahl Blöcke SI-Grundstufe =
`n * (Wertlänge + 7 + 2 * m) / (4040 * Density)`

Für Spacegrößen bis zu 64 GByte: Anzahl Blöcke SI-Index(i)=
`Anzahl Blöcke SI-Index(i-1) * (Wertlänge + 3)/(4040 * Density)`
Für Spacegrößen größer als 64 GByte: Anzahl Blöcke SI-Index(i)=
`Anzahl Blöcke SI-Index(i-1) * (Wertlänge + 4)/(4040 * Density)`

Dabei bezeichnen n, m und i folgende Platzhalter:

`n` = Anzahl Werte

`m` = mittlere Anzahl Verweise pro Wert

`i` = Indexstufe; `i=0` entspricht der Grundstufe

Für die Größe des Primärschlüsselindex gilt die folgende Faustformel:

Für Spacegrößen bis zu 64 GByte: Anzahl Blöcke der Grundstufe =
`Anzahl der ZD-Blöcke * (Länge Primärschlüssel + 6)/(4040 * Density)`
Für Spacegrößen größer als 64 GByte: Anzahl Blöcke der Grundstufe =
`Anzahl der ZD-Blöcke * (Länge Primärschlüssel + 7)/(4040 * Density)`

Für die weiteren PSI-Stufen gilt:

Für Spacegrößen bis zu 64 GByte: Anzahl Blöcke der PSI-Stufe n+1 =
`Pn * (Länge Primärschlüssel + 6)/(4040 * Density)`
Für Spacegrößen größer als 64 GByte: Anzahl Blöcke der PSI-Stufe n+1 =
`Pn * (Länge Primärschlüssel + 7)/(4040 * Density)`

Dabei bedeutet:

`Pn` = Anzahl Blöcke der PSI-Stufe n

Vermeiden unnötiger I/Os auf die WA-LOG-Datei

Analog zum User-Data-Buffer (siehe Seite 74) wird auch beim System-Data-Buffer ein Vorhalt benötigt, um unnötige I/Os auf die WA-LOG-Datei zu vermeiden.

5.4.3 Beispiel für die Berechnung der Puffergrößen

Die folgende Beispielrechnung zeigt, wie User-Data- und System-Data-Buffer dimensioniert werden müssen, wenn eine bestimmte Tabelle resident gehalten werden soll.

Beispiel

Eine Tabelle mit 5000 Sätzen von 20 Spalten und einer Nettodatenlänge von 500 Bytes soll resident gehalten werden. Der Blockfüllgrad beträgt 80 Prozent, die Länge des Primärschlüssels beträgt 20 Byte.

Die Anzahl benötigter Anwenderdatenblöcke lässt sich folgendermaßen errechnen:

```
5000 * (17 + 20 * 3 + 500)/(4040 * 0.8) = 893 Blöcke
```

Die DBH-Option USER-DATA-BUFFER muss um den Wert `893 * 4 = 3572` erhöht werden.

Die Anzahl benötigter Sekundärdatenblöcke errechnet sich folgendermaßen:

Für Spacegrößen bis zu 64 GByte:

```
Database-Translationtable: 5000 / 1346 + 1 = 5
```

Für Spacegrößen größer als 64 GByte:

```
Database-Translationtable: 5000 / 1010 + 1 = 6
```

Primärschlüsselindex:

Für Spacegrößen bis zu 64 GByte:

```
unterste Stufe: 893 * (20 + 6)/(4040 * 0.8) = 8
oberste Stufe:    8 * (20 + 6)/(4040 * 0.8) = 1
```

Für Spacegrößen größer als 64 GByte:

```
unterste Stufe: 893 * (20 + 7)/(4040 * 0.8) = 8
oberste Stufe:    8 * (20 + 7)/(4040 * 0.8) = 1
```

Die DBH-Option SYSTEM-DATA-BUFFER muss um folgenden Wert erhöht werden:

Für Spacegrößen bis zu 64 GByte:

```
(5+8+1) * 4 = 56
```

Für Spacegrößen größer als 64 GByte:

```
(6+8+1) * 4 = 60
```

5.5 DBH-Option RESTART-CONTROL

Die DBH-Option RESTART-CONTROL bietet mit den Parametern TALOG-LIMIT und BUFFER-LIMIT eine Einflussmöglichkeit auf folgende Performance-Faktoren:

- die Häufigkeit physikalischer Schreib-IOs auf die Spaces
- die Dauer eines potentiellen Restarts
- die Laufzeit einiger Administrationskommandos

Durch die Wahl eines möglichst kleinen Wertes für TALOG-LIMIT und BUFFER-LIMIT kann die Laufzeit einiger Administrationskommandos (siehe auch Kapitel „Performance-relevante Aspekte von Administrationsanweisungen“ auf Seite 235) bzw. eines potientiellen Restarts verkürzt werden.

Dadurch werden Blöcke, die logisch geschrieben sind, schneller physikalisch auf Platte geschrieben. In einem Block werden weniger logische Schreibvorgänge gesammelt, bevor er physikalisch geschrieben wird. Die regelmäßige Schreiblast für die Platten, auf denen die Spaces liegen, steigt.

Dieser Effekt kann in der SESMON-Maske I/O bei der Anzahl der „Phys. Write“ in den Spalten „System Data Buffer“ und „User Data Buffer“ beobachtet werden, siehe Handbuch „Datenbankbetrieb“. In der SYSLST-Ausgabe werden diese Werte sogar pro Space zur Verfügung gestellt.

5.6 Cursor-Puffer

Der Cursor-Puffer bildet mit den internen Cursor-Dateien ein temporäres Speichermedium für Zwischenergebnismengen. Folgende Informationen werden im Cursor-Puffer bzw. in den Cursor-Dateien zwischengespeichert:

- Databasekeys, bei großen Treffermengen in Bitlistenrepräsentation
- Ergebnismenge bei Sortierung
- Werte für den Indexupdate bei Mengenfunktionen
- Information über die Recovery-Units bei der Funktion RECOVER

Die Größe des gesamten Cursor-Puffers wird in der DBH-Option BUFFER-SIZE und die Größe eines Pufferrahmens wird in der DBH-Option FRAME-SIZE angegeben. Die Standardgröße des Pufferrahmens beträgt 4KB. Es werden BUFFER-SIZE/FRAME-SIZE Pufferrahmen in der Größe von FRAME-SIZE angelegt.

Eine Vergrößerung der Pufferrahmen (DBH-Option FRAME-SIZE) ohne die gleichzeitige Vergrößerung des gesamten Cursor-Puffers (DBH-Option BUFFER-SIZE) führt zu einer Verringerung der Anzahl Pufferrahmen.

Falls Pufferrahmen verdrängt werden, werden bei einem größeren Pufferrahmen mehr Daten pro SVC geschrieben.

Es sollte ein größerer Pufferrahmen eingestellt werden, falls bei Selektionen über invertierte Attribute große Treffermengen zu erwarten sind.

Unbedingtes Schreiben in die Cursor-Datei

Auch wenn der Cursor-Puffer groß genug ist, schreibt der DBH unter folgenden Bedingungen in eine interne Cursor-Datei:

- bei Sortierung, falls die zu sortierende Menge größer als ein Pufferrahmen ist.
- bei Übergabe der Werte an die Service-Task bei Recover.

Ausgabemaske „I/O“ des Performance-Monitors

Der Performance-Monitor SESMON gibt in der Maske „I/O“ Informationen über logische und physikalische Lese- und Schreibzugriffe (I/Os) aus.

Ist die angezeigte Hitrate gering, so kann der Systemverwalter beim nächsten DBH-Kaltstart oder mit der Administrationsanweisung RECONFIGURE-DBH-SESSION den Cursor-Puffer (DBH-Option BUFFER-SIZE) vergrößern und so versuchen, die Hitrate zu verbessern.

Falls der Systemverwalter die Pufferrahmen (DBH-Option FRAME-SIZE) vergrößert hat und daraufhin die Anzahl der physikalischen I/Os ansteigt, sollte für FRAME-SIZE wieder der ursprüngliche Wert eingestellt werden oder der Cursor-Puffer (DBH-Option BUFFER-SIZE) vergrößert werden, da mehr Pufferrahmen verdrängt wurden.

Ausgabemaske „Service Orders“ des Performance-Monitors

Der Performance-Monitor SESMON gibt in der Maske „Service Orders“ Informationen über die in der Service-Task bearbeiteten Sortieraufträge und DDL- bzw. Utility-Anweisungen aus.

Werden zu viele Sortieraufträge in der Service-Task bearbeitet, sollte der Systemverwalter den Pufferrahmen (DBH-Option FRAME-SIZE) vergrößern, damit mehr Sortierungen in der DBH-Task durchgeführt werden. Falls die zu sortierende Menge kleiner als ein Pufferrahmen ist, wird die Sortierung in der DBH-Task durchgeführt (siehe Abschnitt „Sortierung beeinflussen“ auf Seite 98).

5.7 Anzahl Threads

Die aktuelle Auslastung der Threads ist in der Maske „SYSTEM INFORMATION“ des Performance-Monitors SESMON ersichtlich. Werden dort regelmäßig freie Threads angezeigt, so kann die Ladegröße des DBH mit einer kleineren Anzahl Threads verringert werden (DBH-Option THREADS). Stellt dagegen die Ladegröße des DBH kein Problem dar, so verschlechtert sich die Performance durch ungenutzte Threads nicht.

Die Anzahl Threads bestimmt die Anzahl Aufträge, die der DBH gleichzeitig parallel bearbeiten kann. Es ist zu berücksichtigen, dass in SESAM/SQL auch unterbrochene Aufträge - z.B. wegen einer Transaktionssperre - einen Thread belegen. Eine zu kleine Anzahl Threads kann zu einem Auftragsstau führen. Wenn der letzte Thread auf eine Transaktionssperre läuft, wird die Transaktion zurückgesetzt.

Wenn CREATE INDEX-Anweisungen für partitionierte Tabellen ausgeführt werden, dann sollte die DBH-Option THREADS größer oder gleich der Anzahl Partitionen sein.
Siehe Abschnitt „Optimierter Indexaufbau für partitionierte Tabellen“ auf Seite 230.

Die Anzahl Threads kann beim nächsten DBH-Kaltstart oder mit der Administrationsanweisung RELOAD-DBH-SESSION korrigiert werden.

5.8 Anzahl DBH-Tasks und Multitasking

Lastbalancierung

Die Lastbalancierung, d.h. die Aufteilung der Last durch die Aufträge auf die DBH-Tasks, wird in SESAM/SQL automatisch vorgenommen und ist durch die Administration nicht beeinflussbar.

Dabei wird die Last nicht gleichmäßig über alle DBH-Tasks verteilt, sondern es wird versucht, eine vorhandene Last mit möglichst wenigen Tasks abzuwickeln. Bei wachsender Last werden ausgehend von der Starttask jeweils weitere DBH-Tasks in die Auftragsabwicklung einbezogen. Wenn die Last wieder sinkt, deaktivieren sich diese Tasks wieder.

Option DBH-TASKS

Der Performance-Monitor SESMON gibt in der DBH-Maske „TASKS“ Hinweise zur Qualität der Einstellung dieser Option. Eine ungünstige Einstellung kann beim nächsten DBH-Kaltstart oder mit der Administrationsanweisung RELOAD-DBH-SESSION über die DBH-Option „DBH-TASKS“ korrigiert werden.

SESMON gibt hier für jede DBH-Task Folgendes aus:

- Die TSN der Task.
- Die Anzahl der gerade an diese Task gebunden Threads.
- Die Anzahl der vor dieser Task wartenden Aufträge.
- Die Anzahl Wartezustände wegen I/O der Task im letzten Messintervall.
- Die Anzahl der angestossenen I/Os der Task im letzten Messintervall.
- Den Quotienten der beiden Werte.
- Die verbrauchte CPU-Zeit der Task (der Wert kann aus technischen Gründen nur ausgegeben werden, wenn SESMON in der gleichen Kennung läuft wie der DBH).

Wenn in der Maske DBH-Tasks am Ende der Tabelle DBH-Tasks ausgewiesen werden, die kaum CPU aufgenommen haben, so ist dies ein Hinweis, dass diese Tasks entbehrlich sind.

Wenn in der Maske DBH-Tasks für alle ausgewiesenen Tasks vor der Task wartende Aufträge ausgewiesen werden, so ist dies als Indiz zu betrachten, dass zurzeit die Anzahl DBH-Tasks zu klein ist.

Für eine optimale Nutzung von Multiprozessoren ist Folgendes zu beachten:

- Die Anzahl der DBH-Tasks sollte in der Regel (ab 3 CPUs) kleiner der Anzahl der CPUs sein.
- Das Verhältnis zwischen der Anzahl CPUs und der Anzahl DBH-Tasks ist von der Aufteilung des CPU-Bedarfs des DBH und der Anwendung abhängig.
- Wenn die Anzahl DBH-Tasks kleiner als die Anzahl CPUs ist, sollte für die DBH-Tasks eine feste Priorität vergeben werden.
- Den Anwendertasks sollten ggf. eine feste Priorität zugewiesen werden, die aber um 3-5 Prozentpunkte höher sein sollte als die Priorität der DBH-Tasks, damit die DBH-Tasks immer Arbeit vorfinden.

5.9 Suborders

Die Anzahl der genutzten und der maximal verfügbaren Suborders ist in der Maske „SYSTEM INFORMATION“ des Performance-Monitors SESMON ersichtlich. Liegt die Anzahl der zur Verfügung stehenden Suborders deutlich über der maximal genutzten, so bedeutet das zwar keine Laufzeitverluste für den DBH, aber der Speicherverbrauch wird unnötig groß.

5.10 Planpuffer

SQL-Anweisungen werden vor der ersten Ausführung in eine einheitliche, interne Darstellung, den SQL-Zugriffsplan, transformiert (siehe Abschnitt „Aufbau und Steuerung des Optimizers“ auf Seite 24). Diese interne Darstellung der SQL-Anweisung, im Folgenden kurz Plan genannt, wird in einem DBH-spezifischen Speicherbereich, dem Planpuffer, abgelegt. Bei einer erneuten Ausführung der SQL-Anweisung kann der bereits erstellte Plan wiederverwendet werden. Da die Plangenerierung im Vergleich zur Ausführung relativ teuer ist, sollte eine Plangenerierung für eine SQL-Anweisung so selten wie möglich erfolgen.

Die Größe des Planpuffers ist abhängig von folgenden DBH-Optionen:

- DBH-Option COLUMNS
- Parameter PLANS der DBH-Option SQL-SUPPORT
- DBH-Option USERS (in geringem Maße)

Für die Dimensionierung des Planpuffers ist die Kenntnis einiger Zusammenhänge hilfreich. Diese werden in den folgenden Abschnitten erläutert.

5.10.1 Zuordnung Plan zu SQL-Anweisung

Jeder Plan gehört zu einer oder mehreren SQL-Anweisungen. Andererseits kann es zu einer SQL-Anweisung einen, mehrere oder keinen Plan geben. Wieviele Pläne pro SQL-Anweisung generiert werden, ist abhängig von der Art der SQL-Anweisung:

- Für transaktionssteuernde Anweisungen (COMMIT WORK, ROLLBACK WORK, etc.) und session-steuernde Anweisungen (SET CATALOG, SET SCHEMA etc.) werden keine Pläne erzeugt.
- Für statische DML-, DDL-, SSL-, USER- und UTILITY-Anweisungen, die sich nicht auf Cursor beziehen, gibt es genau einen Plan.
- Für einen statischen Cursor gibt es genau einen Plan. Darauf kann mit den zugehörigen Cursor-Anweisungen (OPEN, FETCH, CLOSE, etc.) zugegriffen werden, d.h. ein Plan gehört zu mehreren Anweisungen.
- Für ändernde Anweisungen auf einen Cursor (DELETE ... WHERE CURRENT OF ..., UPDATE ... WHERE CURRENT OF ...) gibt es zwei Pläne: einen Plan für die Anweisung plus einen weiteren Plan für den Cursor (s.o.).
- Bei den dynamischen Anweisungen PREPARE und EXECUTE IMMEDIATE werden für die zu präparierenden SQL-Anweisungen Pläne generiert. Allerdings existiert kein Plan für die dynamischen SQL-Anweisungen selbst (PREPARE, EXECUTE IMMEDIATE, EXECUTE).

Auf den via PREPARE generierten Plan kann mit EXECUTE zugegriffen werden bzw., falls mittels PREPARE ein Cursor-Plan erstellt wurde, mit den Anweisungen für dynamische Cursor (OPEN, FETCH etc.).

Für zwei statische SQL-Anweisungen innerhalb eines SQL-Moduls werden zwei Pläne erzeugt, auch wenn die Anweisungen textuell vollkommen identisch sind. Dagegen können zwei verschiedene dynamische Anweisungen eines Moduls dem gleichen Plan zugeordnet werden, wenn sie sich auf identische Anweisungstexte beziehen. Zwei Anweisungstexte gelten als identisch, wenn die Texte inklusive Pragmas zeichenweise identisch sind und die gleichen Voreinstellungen bzgl. Datenbankname und Schemaname gelten. Es findet keine Normalisierung (also z.B. Komprimierung von Leerzeichen) statt.

Mehrbenutzerfähige Pläne

Ein SQL-Modul kann von verschiedenen Auftraggebern ausgeführt werden.

Statische SQL-Anweisungen sind nicht auftraggeberspezifisch. Es genügt daher, für sie jeweils nur einen Plan zu generieren, der von allen Auftraggebern benutzt wird.

Die dynamischen Anweisungen PREPARE und EXECUTE IMMEDIATE sind vom aktuellen Kontext des Auftraggebers abhängig (nämlich vom Inhalt der Anweisungsvariablen). Die SQL-Anweisung, die in einer solchen Anweisungsvariablen enthalten ist, kann wiederum vom aktuell voreingestellten Datenbank- oder Schemanamen abhängen oder auf temporäre Views Bezug nehmen.

Zwei dynamische Anweisungen, deren Anweisungvariablen den identischen Text beinhalten, können den gleichen Plan benutzen, falls die voreingestellten Datenbank- und Schemanamen gleich sind.

Somit sind alle Pläne mehrbenutzerfähig.

5.10.2 Lebensdauer von Plänen

Ein Plan wird im Planpuffer solange wie möglich aufbewahrt, damit er bei erneuter Ausführung der zugehörigen SQL-Anweisung wiederverwendet werden kann. Allerdings muss ein Plan gelöscht werden, wenn durch SQL-DDL-, SQL-SSL- oder Utility-Anweisungen die dem Plan zu Grunde liegenden Catalog-Informationen verändert werden. Beim physikalischen Schliessen einer Datenbank werden alle Pläne zu dieser Datenbank gelöscht.

Ein Plan wird je nach zugehöriger SQL-Anweisung unterschiedlich lange benötigt:

- Bei statischen Cursorn von OPEN CURSOR bis CLOSE CURSOR und darüber hinaus, solange noch mindestens ein Auftraggeber (per STORE) eine Cursor-Position gespeichert hat.
- Bei sonstigen statischen SQL-Anweisungen und EXECUTE IMMEDIATE nur während der Bearbeitung dieser Anweisung.
- Bei dynamischen Cursorn bis Transaktionsende und darüber hinaus, solange noch mindestens ein Auftraggeber (per STORE) eine Cursor-Position gespeichert hat.
- Bei sonstigen dynamischen SQL-Anweisungen bis Transaktionsende.

Ein Plan, der im Augenblick nicht mehr benötigt wird, kann verdrängt werden, um einem neuen Plan Platz zu machen. SESAM/SQL wählt dabei nach Möglichkeit denjenigen Plan aus, der am längsten nicht mehr verwendet wurde.

5.10.3 Anzahl von Plänen

Im Parameter PLANS der DBH-Option SQL-SUPPORT legt der Systemverwalter die Mindestanzahl paralleler SQL-Zugriffspläne für eine DBH-Session fest. Bezüglich der Anzahl von Plänen sind zwei Fragestellungen vorstellbar:

1. Wieviele unterschiedliche Pläne gibt es insgesamt über alle Auftraggeber der DBH-Session betrachtet?
2. Wieviele Pläne werden maximal gleichzeitig benötigt?

Die Frage nach der Gesamtzahl von Plänen (1) liefert einen Maximalwert für die Vergabe des Parameters PLANS. Hier geht man davon aus, dass jeder in der DBH-Session neu angelegte Plan einen bisher freien Speicher belegt und bis zum Ende der DBH-Session im Planpuffer verbleibt, gleichgültig, wie lange er schon existiert und ob er jemals wiederverwendet wurde. Je nach Anwendungs-Szenario kann der Planpuffer dann viel zu groß dimensioniert sein und unnötig Betriebsmittel belegen.

Die Antwort darauf, wieviele Pläne maximal benötigt werden (2), liefert einen unteren Grenzwert für die Einstellung des Parameters PLANS. Ist der Planpuffer noch kleiner, kann er vollständig mit Plänen belegt sein, die noch nicht verdrängt werden dürfen, z.B., weil sie zu einem gerade offenen Cursor gehören. Der Versuch, den Plan für eine weitere SQL-Anwei-

sung zu erzeugen, würde dann wegen Betriebsmittelengpasses (Planpuffer-Überlauf) abgewiesen. Es kann aber auch der unerwünschte Effekt eintreten, dass ein Plan verdrängt werden muss, der kurz darauf wieder benötigt wird, neu generiert werden muss, dabei seinerseits einen dritten Plan verdrängt etc.

Um zu verhindern, dass häufig benutzte Pläne ständig neu generiert werden müssen, ist daher ein Mittelweg zu beschreiten. Dabei sollte man Folgendes berücksichtigen:

- Anzahl Pläne, die in regelmäßigen Abständen wiederverwendet werden. Hier sind sowohl solche Pläne zu berücksichtigen, die derselbe Auftraggeber wiederverwendet, als auch Pläne, die verschiedene Auftraggeber verwenden (z.B. durch eine UTM-Anwendung mit zahlreichen Terminals).
- Anzahl Pläne, die zwar nicht wiederverwendet werden, die aber einen anderen noch benötigten Plan verdrängen könnten, wenn der Abstand der Zugriffe auf den benötigten Plan zu groß wird.
- Anzahl Pläne, die eine längere Lebensdauer haben (siehe Abschnitt „Lebensdauer von Plänen“ auf Seite 84).

Statische SQL-Anweisungen

Um bei statischen SQL-Anweisungen die Anzahl Pläne zu ermitteln, sind Kenntnisse über alle Anwendungsprogramme einer DBH-Session erforderlich. Insbesondere muss bekannt sein, welche SQL-Anwendungen parallel in der betreffenden Konfiguration laufen.

Folgende Werte sind zu ermitteln:

- Anzahl SQL-Anweisungen in einem Modul
- Anzahl der zu diesem SQL-Modul gehörigen Pläne (siehe auch Abschnitt „Zuordnung Plan zu SQL-Anweisung“ auf Seite 82)
- Anzahl Auftraggeber, die gemeinsam das SQL-Modul in einer Konfiguration benutzen
- Die hieraus errechnete Anzahl der für ein Modul benötigten Pläne
- Anzahl Pläne (von statischen Anweisungen) für die gesamte Konfiguration

Dynamische SQL-Anweisungen

Bei dynamischen Anweisungen kann die Gesamtzahl Pläne nur geschätzt werden. Da der Anweisungstext im Allgemeinen nicht von vornherein bekannt ist, kann ein Anwender z.B. mit einer einzigen EXECUTE IMMEDIATE-Anweisung nacheinander beliebig viele unterschiedliche Pläne erzeugen. Es sind aber auch Anwendungen denkbar, bei denen die SQL-Anweisungstexte dynamischer Anweisungen nicht etwa spontan am Terminal eingegeben werden, sondern einem begrenzten Pool von Anweisungen entnommen werden.

Parameter PLANS

Bevor man endgültig einen geeigneten Wert für den Parameter PLANS der DBH-Option SQL-SUPPORT festlegt, sollte man die daraus resultierende Planpuffergröße bestimmen (siehe Abschnitt „Größe des Planpuffers“). Mit Hilfe des Performance-Monitors lässt sich dann die Größe des Planpuffers optimieren (siehe Abschnitt „Optimieren der Planpuffergröße“ auf Seite 87).

Sind nicht alle Informationen für eine theoretische Abschätzung des Parameters PLANS bekannt, so kann man natürlich zunächst die Standard-Optionen benutzen, und die optimale Größe mit Hilfe des Performance-Monitors ermitteln.

Der Parameter PLANS der DBH-Option SQL-SUPPORT stellt allerdings nur einen Anhaltspunkt dafür dar, wieviele Pläne im Planpuffer abgespeichert werden können. Der durchschnittlich für einen Plan zur Verfügung gestellte Speicherplatz orientiert sich an der DBH-Option COLUMNS und reicht im Allgemeinen auch für komplexe Pläne aus. Liegen relativ einfache SQL-Anweisungen vor oder ist die Option COLUMNS zu hoch eingestellt, finden weitaus mehr Pläne im Planpuffer Platz, als im Parameter PLANS angegeben.

5.10.4 Größe des Planpuffers

Die Größe eines Planes ist abhängig von der zugehörigen SQL-Anweisung sowie der zugehörigen Schemainformation. Je mehr Tabellen und Spalten in einer SQL-Anweisung angesprochen werden, um so größer wird im Allgemeinen der zugehörige Plan. Ein Plan wird auch um so größer, je mehr Integritätsbedingungen bei Durchführung der zugehörigen SQL-Anweisung geprüft werden müssen.

SESAM/SQL berechnet die ungefähre Planpuffergröße aus der durchschnittlich erwarteten Plangröße, wobei die DBH-Optionen COLUMNS, USERS und SQL-SUPPORT/Parameter PLANS in die Berechnung einfließen. Je nach der speziellen Anwendungssituation können die tatsächlichen Plangrößen jedoch stark variieren.

Der Planpuffer ist aufgeteilt in zwei Speicherbereiche, den Primärpuffer, der die Pläne enthält und den Sekundärpuffer mit der zugehörigen Verwaltungsinformation.

Die Tabelle 6 auf Seite 87 zeigt einige Eckwerte für die Größe des Planpuffers in Abhängigkeit von den oben genannten DBH-Optionen (Annahme: USERS=24). Aus den dargestellten Eckwerten lässt sich die Planpuffergröße für die aktuelle DBH-Session überschlägig durch Interpolieren ermitteln. Die Werte lassen sich nachträglich über den Performance-Monitor optimieren (siehe auch Abschnitt „Optimieren der Planpuffergröße“).

PLANS	COLUMNS			
	256		1024	
	Primärpuffer	Sekundärpuffer	Primärpuffer	Sekundärpuffer
1	150 KB	27 KB	591 KB	92 KB
10	468 KB	74 KB	1,78 MB	268 KB
70	1,73 MB	260 KB	6,28 MB	930 MB
100	2,36 MB	353 KB	8,53 MB	1,26 MB
1000	21,3 MB	3,14 MB	76,0 MB	11,2 MB
10000	210 MB	31,0 MB	751 MB	111 MB

Tabelle 6: Größe des Planpuffers in Abhängigkeit von DBH-Optionen

Bei den Standardwerten der DBH-Optionen (SQL-SUPPORT/PLANS=70, COLUMNS=256, USERS=24) beträgt die Gesamtgröße des Planpuffers ca. 2,0 Mbyte. Alle Werte gelten für eine Einstellung der Option USERS von 24. Für jeden User mehr oder weniger ändert sich die Größe des Sekundärpuffers entsprechend um ca. 200 Byte (der Primärpuffer ist unabhängig vom Wert der DBH-Option USERS).

In Folgeversionen von SESAM/SQL können sich die Plangrößen durch interne Anpassungen verändern. Dies kann entsprechende Auswirkungen auf die Größe des Planpuffers haben. Daher gelten die Werte in Tabelle 6 vorbehaltlich möglicher Änderungen in künftigen Versionen.

5.10.5 Optimieren der Planpuffergröße

Über die Maske „SQL-INFORMATION“ des Performance-Monitors SESMON kann man unter anderen folgende Kennzahlen ermitteln:

- Größe des Planpuffers (Primärpuffer)
- Anzahl der Planzugriffe (entspricht der Anzahl Ausführungen der zugehörigen SQL-Anweisung)
- Anzahl der Plangenerierungen
- Anzahl der Pläne im Planpuffer (Primärpuffer)
- belegter Platz im Planpuffer (Primärpuffer)
- Größe des Sekundärpuffers

Bei der ersten Ausführung einer SQL-Anweisung wird der zugehörige Plan generiert. Bei weiteren Ausführungen der SQL-Anweisung kann der zuvor generierte Plan wiederverwendet werden, vorausgesetzt, der von diesem Plan belegte Platz wurde nicht zwischenzeitlich für einen anderen Plan benötigt.

Der Planpuffer ist optimal dimensioniert, wenn nach einer Einschwung-Phase die Anzahl Plangenerierungen deutlich geringer als die Anzahl Planzugriffe ist. Zahlreiche Pläne werden dann nur einmal erzeugt und wiederholt ausgeführt, ohne dass sie zwischenzeitlich gelöscht werden. Hierbei handelt es sich entweder um Pläne, die von mehreren Auftraggebern benutzt werden, oder um Pläne, die vom selben Benutzer mehrfach abgearbeitet werden, z.B. in Schleifen.

Eine andere Situation liegt z.B. vor, wenn überwiegend Anwendungen laufen, bei denen spontan im Dialog SQL-Anweisungen eingegeben werden können. In diesem Falle werden dynamische Anweisungen mit wahrscheinlich unterschiedlichen Texten verarbeitet, so dass ständig neue Pläne generiert werden müssen und keine Optimierung bzgl. der Wiederverwendung von Plänen möglich ist.

Bei der Dimensionierung des Planpuffers sollte dann darauf geachtet werden, dass ausreichend Platz für alle gleichzeitig benötigten Pläne vorhanden ist, so dass keine SQL-Anweisung wegen Betriebsmittelengpasses abgewiesen werden muss. Insbesondere sei hier daran erinnert, dass ein mittels PREPARE erzeugter Plan mindestens bis Transaktionsende im Planpuffer verbleibt (falls nicht der gleiche Auftraggeber einen erneuten PREPARE für den gleichen Anweisungsbezeichner durchführt). Unnötig lange offen gehaltene Transaktionen sind hier also genauso von Nachteil wie das parallele Präparieren mehrerer Anweisungen, wenn womöglich auch eine sukzessive Abarbeitung (mit dem gleichen Anweisungsbezeichner) erfolgen kann.

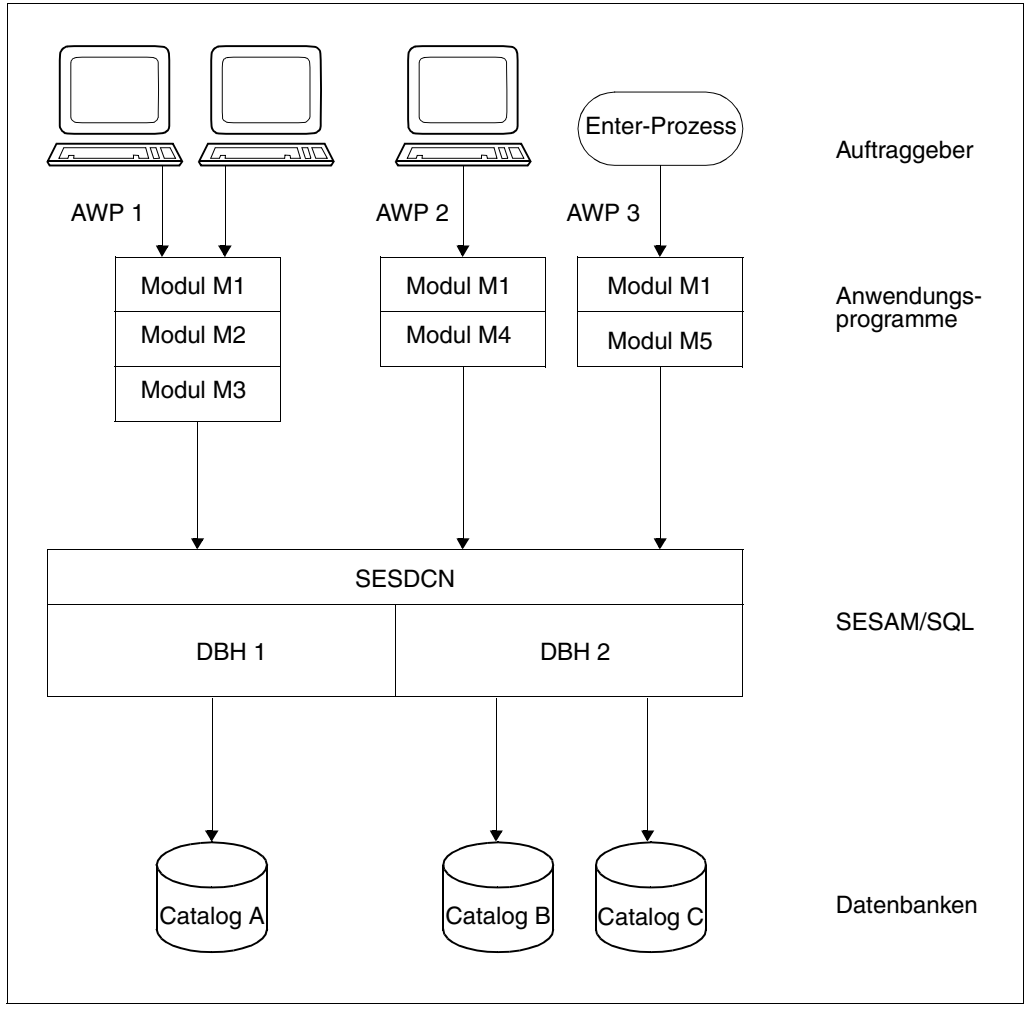
Pläne, die längere Zeit nicht benutzt wurden, werden nur gelöscht, wenn der belegte Platz für einen anderen Plan benötigt wird. Ein voll ausgelasteter Planpuffer ist also noch kein Anzeichen für eine zu geringe Dimensionierung. Daher kann der belegte Platz durchaus gleich der Gesamtgröße des Planpuffers (Primärpuffers) sein oder sollte nur unwesentlich unter der Gesamtgröße liegen.

Wenn sich die SQL-Anwendungen oder die Schema-Informationen der Datenbanken ändern, muss der Parameter PLANS der DBH-Option SQL-SUPPORT ggf. angepasst werden. Für eine spezielle Anwendungssituation kann man aus den obigen Kennzahlen die Anzahl Pläne und die durchschnittliche Plangröße ermitteln.

5.10.6 Abschätzen der Mindestanzahl Pläne in einer Beispielanwendung

Beispiel

Die folgende Darstellung zeigt eine Beispielanwendung, an der zwei DBHs beteiligt sind. Für den Parameter PLANS der DBH-Option SQL-SUPPORT soll jeweils ein optimaler Wert geschätzt werden.



Das Anwendungsprogramm AWP1 ist eine OLTP-Anwendung, die von maximal 300 Terminals parallel benutzt wird.
Das Anwendungsprogramm AWP2 ist eine Decision-Support-Anwendung, die von 20 Terminals mit individuell verschiedenen SQL-Abfragen benutzt wird.

Das Anwendungsprogramm AWP3 ist eine Batchanwendung, die von lediglich einem BS2000-Enter-Prozess gestartet wird.
Die OLTP Anwendung und die Decision-Support-Anwendung werden hauptsächlich tagsüber genutzt. Die Batch-Anwendung wird jede Nacht einmal gestartet.

Die Datenbank Catalog A ist DBH 1 zugeordnet, Catalog B und Catalog C sind DBH 2 zugeordnet.

Die Module M1, M2 und M3 bearbeiten den Catalog A mit statischen SQL-Anweisungen. Das Modul M4 enthält nur dynamische SQL-Anweisungen, die, abhängig von den präparierten Anweisungen, jeden der drei Cataloge ansprechen können. Das Modul M5 enthält zu jeweils 50% Anweisungen, die sich auf die Cataloge A und B beziehen.

Die Anzahl der Pläne ist bei statischen Anweisungen einfach durch Zählen zu ermitteln. Bei dynamischen Anweisungen kann man die Anzahl der benötigten Pläne nur schätzen (mit einem einzigen EXECUTE IMMEDIATE, das in einer Schleife abgearbeitet wird, können beliebig viele unterschiedliche Anweisungstexte eingegeben und somit unterschiedliche Pläne erzeugt werden).

In obigem Beispiel ist für Modul M4 angenommen, dass jeder Auftraggeber im Schnitt 60 (pro Catalog 20) spontan formulierte und somit individuell verschiedene Anweisungen im Dialog eingibt. Der Rest der SQL-Anweisungen stammt aus einem vom Programm vorgegebenen Pool von 30 Anweisungstexten (jeweils 10 pro Catalog).

Um die Mindestanzahl Pläne abzuschätzen, werden zunächst die beschriebenen Kennzahlen ermittelt:

Anzahl SQL-Anweisungen		**Pläne**		**Auftraggeber**	**generierte Pläne**	
		insgesamt	**individuell**		**DBH 1**	**DBH 2**
M1	30	20	-	321	20	-
M2	200	150	-	300	150	-
M3	12	10	-	300	10	-
M4	100	90	60	20	410	820
M5	1200	1000	-	1	500	500
				Summe	1090	1320

Tabelle 7: Kennzahlen zum Abschätzen der Mindestanzahl Pläne

In Tabelle 7 ergibt sich die Anzahl der Auftraggeber durch Aufsummieren über alle Anwendungsprogramme, in denen die Module verwendet werden. Die Anzahl generierter Pläne wiederum folgt aus der Anzahl individueller Pläne multipliziert mit der Anzahl Auftraggeber plus der Anzahl gemeinsam benutzter Pläne.

Anzahl Pläne in der Konfiguration zu „DBH 1“

Die Gesamtzahl der zur Konfiguration „DBH 1“ gehörenden Pläne beträgt theoretisch 1090. Allerdings werden nicht alle Pläne zur gleichen Zeit benötigt (s.o.). Der Planpuffer braucht nur den Plänen Platz zu bieten, die gleichzeitig im Puffer benötigt werden. Da der Planpuffer bei Belegung der DBH-Option PLANS mit dem Wert 1090 relativ groß wird (ca. 18 MByte für Primärpuffer und Sekundärpuffer zusammen bei COLUMNS=256 und USERS=321), sollte ein kleinerer Wert gesucht werden.

Ein sinnvoller Ansatz für eine erste Belegung ist die Anzahl der relativ häufig genutzten mehrbenutzerfähigen Pläne. In diesem Beispiel sind das die mehrbenutzerfähigen Pläne in den Modulen M1, M2 und M3 (Gesamtzahl 180) sowie die gemeinsam benutzter Pläne in Modul M4, die sich auf Catalog A beziehen (Gesamtzahl 10), insgesamt also 190. Außerdem muss man davon ausgehen, dass pro Terminal der Decison Support Anwendung noch ein Plan benutzt wird, der durch eine spontan formulierte SQL-Anweisung entsteht und auf den vermutlich kein zweiter Anwender zugreift. Berechnet man noch eine gewisse Reserve für gleichzeitig benutzte Pläne der Batch-Anwendung ein, dann dürfte 220 ein vernünftiger Wert für die Wahl des Parameters PLANS sein.

Allerdings orientiert sich der für einen Plan im Durchschnitt zur Verfügung gestellte Speicherplatz an der DBH-Option COLUMNS und reicht im Allgemeinen auch für komplexe Pläne aus. Liegen im Durchschnitt relativ einfache SQL-Anweisungen vor oder ist die Option COLUMNS zu hoch eingestellt, dann finden sehr viel mehr Pläne im Puffer Platz als der Parameter PLANS besagt.

Anzahl Pläne in der Konfiguration zu „DBH 2“

Die Anzahl der in der Konfiguration „DBH 2“ erstellten Pläne beträgt insgesamt 1320. Ein sinnvoller Ansatz für eine erste Belegung der Option PLANS ist sehr vom Aufbau des Batchprogramms (bzw. von Modul M5) abhängig. Da das Programm jedoch nur von einem Auftraggeber benutzt wird, ist die Gesamtanzahl Pläne hier nicht kritisch für die Optimierung. Lediglich die Mindestanzahl der gleichzeitig in der Batch-Anwendung verwendeten Pläne (siehe Abschnitt „Lebensdauer von Plänen“ auf Seite 84) sollte ermittelt werden. Addiert man dazu die 20 fest vorgegebenen (vermutlich häufiger benutzten) Pläne des Moduls M4, die sich auf die Cataloge B und C beziehen, dann hat man eine vernünftige untere Grenze für den Parameter PLANS in der Konfiguration „DBH 2“. Ob der Planpuffer tatsächlich sinnvoll dimensioniert ist, kann man allerdings nur mit Hilfe des Performance-Monitors prüfen.

5.11 Prefetch-Puffer für den Schubmodus

Im Prefetch-Puffer werden alle Teilergebnismengen (Schübe) von Cursorn verwaltet, die im Schubmodus geöffnet sind. Ein Schub enthält die Werte der Ergebnissätze und Verwaltungsinformation zu diesen Werten. Die Größe des Prefetch-Puffers legt der Anwender in der Konfigurationsdatei der Anwendung über den Parameter PREFETCH-BUFFER fest.

5.11.1 Größe des Prefetch-Puffers

Der Platz im Prefetch-Puffer wird durch folgende Daten belegt:

- Nutzdaten (Schübe)
- Daten der Speicherverwaltung (Verwaltung der Speicherbereiche für die Nutzdaten)

Der Speicherplatzbedarf eines Schubes ist abhängig von folgenden Faktoren:

- Anzahl der Spalten im Cursor
- Datentypen der einzelnen Spalten
- aktuelle Werte der zwischengespeicherten Ergebnissätze
- Anzahl von Ergebnissätzen im Schub.

Der für einen Schub benötigte Speicherbereich lässt sich anhand folgender Formel grob abschätzen:

```
376 Byte + sw + f * (40 Byte + 10 Byte * s)
```

mit f = Anzahl Spalten im Cursor
s = Anzahl von Ergebnissätzen im Schub
sw = Speicherbedarf für alle im Schub enthaltenen Werte der Ergebnissätze

Der Wert für sw kann dabei folgendermaßen bestimmt werden:

1. Für Cursor, deren Spalten feste Längen haben:

   $$SW = S * \left(\sum_{i=1}^{f} l(i) \right)$$

   l(i) = Länge des Spaltenwerts der Spalte i

2. Für Cursor, die Spalten variabler Länge enthalten:

   $$SW = \sum_{i=1}^{f} \sum_{j=1}^{s} l(i,j)$$

   l(i,j) = Länge des Spaltenwerts der Spalte i im Ergebnissatz j

Ein Teil des Prefetch-Puffers wird für die Speicherverwaltung benötigt. Daher weist SESAM/SQL ggf. einen Teil des Puffers als belegt aus (z.B. durch SESMON), obgleich noch keine Nutzdaten darin enthalten sind. Dies ist insbesondere dann zu erwarten, wenn nur ein sehr kleiner Puffer (wenige Kbyte) bereitgestellt wird.

Der für die Speicherverwaltung benötigte Platz ist abhängig von der Zahl der gleichzeitig zu verwaltenden Schübe. Eine grobe Abschätzung für die Größe des für die Verwaltung benötigten Speichers liefert fder Wert `a * 32 Byte`, wobei a die Anzahl der parallel zu verwaltenden Schübe ist.

Genauer gilt folgendes:

Der für die Speicherverwaltung benötigte Speicherbereich besteht aus zwei Bereichen:

- einer zentralen Datenstruktur mit einer festen Länge von 320 Byte
- einem Bereich variabler Länge zur Verwaltung der einzelnen Schübe.
  Pro Schub wird eine Struktur von 32 Byte Länge benötigt.

Bei der Initialisierung des Prefetch-Puffers wird für den Bereich variabler Länge ein Prozent des Speichers verwendet, jedoch mindestens 96 Byte und höchstens 4 Kbyte.

Bei Bedarf wird der variable Teil der Speicherverwaltung dynamisch erweitert. Die Größe der Erweiterung hängt ab vom noch verfügbaren freien Speicher und dem mittleren Speicherbedarf für einen Schub zum Zeitpunkt der Erweiterung. Die Größe der Erweiterung errechnet sich folgendermaßen:

```
                    freier Speicher
32 Byte * ----------------------------------------
          mittlerer Speicherbedarf für einen Schub
```

Aber auch hier gilt, dass um mindestens 96 Byte und um höchstens 4 Kbyte erweitert wird. Einmal vorgenommene Erweiterungen der Speicherverwaltung werden nicht mehr zurückgenommen.

In TIAM-Umgebung hat jeder SQL-Anwender seinen eigenen Prefetch-Puffer. In dieser Umgebung kann der Speicherbedarf für den Prefetch-Puffer anhand der im Schubmodus verwendeten Cursor recht exakt berechnet werden.

In UTM-Umgebung haben alle SQL-Anwender einer Applikation einen gemeinsamen Prefetch-Puffer. Damit ist der Speicherplatzbedarf nicht nur von der Struktur der Teilprogramme, sondern auch von der dynamischen Last auf der Applikation abhängig. Insbesondere in dieser Umgebung empfiehlt es sich, wie im folgenden Abschnitt beschrieben, den Prefetch-Puffer mit Hilfe des Performance-Monitors SESMON optimal zu dimensionieren.

Über den Wert des Parameters PREFETCH-BUFFER in der Konfigurationsdatei kann der Anwender den Prefetch-Puffer bei Bedarf neu dimensionieren. Der neue Wert wird beim nächsten Neustart der Anwendung wirksam. Eine Neudimensionierung des Prefetch-Puffers zur Laufzeit der Anwendung ist nicht möglich.

5.11.2 Optimieren der Prefetch-Puffergröße

Der Prefetch-Puffer einer Anwendung ist dann optimal dimensioniert, wenn alle Speicheranforderungen immer sofort befriedigt werden können und die maximale Speicherauslastung möglichst nahe bei 100% liegt. In der SESMON-Maske „PREFETCH-BUFFERS“ ist dies daran erkennbar, dass für eine Anwendung die Hitraten in allen Speicherklassen, in denen Speicheranforderungen angefallen sind, 100% betragen und dass der Wert für die maximale Speicherauslastung (Spalte „Max. Occ.“) nahe bei 100% liegt.

Mit Hilfe des Performance-Monitors SESMON kann die optimale Speichergröße des Prefetch-Puffers für eine TIAM-Anwendung oder UTM-Applikation folgendermaßen bestimmt werden:

1. Der Prefetch-Puffer wird zunächst so groß gewählt (z.B. 4 MByte), dass möglichst alle Speicheranforderungen sofort befriedigt werden können.
2. Nach dem Start der Task bzw. der Applikation lässt man diese solange mit Prefetch arbeiten, bis sich der Wert „Max. Occ.“ (= max. Speicherauslastung) in der SESMON-Maske „PREFETCH-BUFFERS“ nicht mehr ändert.
3. Ist die Hitrate in einer oder mehreren Speicherklassen kleiner als 100%, so war der Prefetch-Puffer nicht genügend groß, um alle Speicheranforderungen zu befriedigen und sollte deshalb, wenn möglich, vergrößert (z.B. verdoppelt) werden.
   – Ist eine Vergrößerung des Prefetch-Puffers möglich, so setzt man nach der Vergrößerung wieder bei Punkt 2 auf.
   – Ist eine Vergrößerung des Prefetch-Puffers nicht möglich, wird das Verfahren beendet.

4. Sind die Hitraten in allen Speicherklassen, in denen Speicheranforderungen anfallen, 100%, so kann man die optimale Größe für den Prefetch-Puffer anhand des Wertes für die maximale Speicherauslastung („Max. Occ.“) ermitteln.
   – Ist der Wert für „Max. Occ.“ bereits optimal (z.B. > 90%), so wird das Verfahren beendet.
   – Liegt der Wert für „Max. Occ.“ zwischen 10% und 90%, so ist die optimale Größe für den Prefetch-Puffer gleich dem in der Spalte „Max. Occ.“ angegebenen Anteil der momentan eingestellten Prefetch-Puffergröße, wobei man eine kleine Sicherheitsreserve einkalkulieren sollte. Bei UTM-Applikationen sollte diese Sicherheitsreserve großzügiger bemessen sein, da bei UTM-Applikationen der Prefetch-Puffer nicht reorganisierbar ist und der freie Speicher demzufolge sehr stark fragmentiert sein kann, so dass Pufferanforderungen trotz insgesamt genügend freien Speichers nicht mehr befriedigt werden können.
     Nach einer Anpassung der Prefetch-Puffergröße ist mit Punkt 2 fortzufahren.
   – Ist der Wert für „Max. Occ.“ zwischen 1% und 9%, so ist der Wert für die optimale Größe des Prefetch-Puffers nicht sehr genau zu ermitteln, da der Wert für „Max. Occ.“ nur eine signifikante Stelle hat. In diesem Fall wird empfohlen, den Prefetch-Puffer auf ein Zehntel der momentan eingestellten Prefetch-Puffergröße zu verringern und mit Punkt 2 fortzufahren.
   – Ist der Wert für „Max. Occ.“ 0%, so sollte der Prefetch-Puffer auf ein hundertstel der momentan eingestellten Prefetch-Puffergröße verringert werden und mit Punkt 2 fortgefahren werden.

Beispiel

Für die UTM-Applikation „MYAPPL“ wird zu Beginn eine Prefetch-Puffergröße von 1 MByte eingestellt. Nachdem die Applikation eine gewisse Zeit gelaufen ist, zeigt die SESMON-Maske „PREFETCH-BUFFERS“ folgendes Bild:

```
================================================================================
>>> PREFETCH-BUFFERS <<<    <ver>              CNF: X     Time: <time>

Appl./                      H i t r a t e                       Perc. Max.
TSN      T <=1KB <=2KB <=4KB <=8KB <=16KB <=32KB <=64KB >64KB  Occ. Occ.
 .
 .
 .
MYAPPL   U   --%  100%  100%   --%   100%    --%    --%   --%    3%   7%
 .
 .
 .

>>> INTERRUPT WITH KEY K2 <<<
================================================================================
```

Nachdem der Wert von „Max. Occ.“ kleiner als 10% ist, wird als neue Größe für den Prefetch-Puffer 100 Kbyte eingestellt. Nachdem die Anwendung mit diesem Wert neu gestartet wurde und einige Zeit gelaufen ist, liefert SESMON folgende Werte:

```
================================================================================
>>> PREFETCH-BUFFERS <<<    <ver>              CNF: X     Time: <time>

Appl./                      H i t r a t e                       Perc. Max.
TSN      T <=1KB <=2KB <=4KB <=8KB <=16KB <=32KB <=64KB >64KB  Occ. Occ.
 .
 .
 .
MYAPPL   U   --%  100%  100%   --%   100%    --%    --%   --%   47%  76%
 .
 .
 .

>>> INTERRUPT WITH KEY K2 <<<
================================================================================
```

76% der aktuell eingestellten Prefetch-Puffergröße von 100 Kbyte, also 76 Kbyte, müsste nun die optimale Größe sein. Berücksichtigt man noch eine Sicherheitsreserve von ca. 10%, so ergibt sich eine Größe des Prefetch-Puffers von 84 Kbyte. Nach einem Neustart der Anwendung mit der neuen Prefetch-Puffergröße liefert SESMON nach einiger Zeit folgende Werte:

```
================================================================================
>>> PREFETCH-BUFFERS <<<    <ver>              CNF: X     Time: <time>

Appl./                      H i t r a t e                       Perc. Max.
TSN      T <=1KB <=2KB <=4KB <=8KB <=16KB <=32KB <=64KB >64KB  Occ. Occ.
 .
 .
 .
MYAPPL   U   --%  100%  100%   --%   100%    --%    --%   --%   83%  91%
 .
 .
 .

>>> INTERRUPT WITH KEY K2 <<<
================================================================================
```

Damit hat man eine sehr gute Speicherauslastung erreicht und kann das Verfahren zur Ermittlung der optimalen Speicherauslastung beenden.

5.12 Service-Tasks

CPU-intensive Aktionen, etwa das Sortieren von Zwischenergebnismengen oder einige Utility-Funktionen, lagert der DBH in Service-Tasks aus. Der Systemverwalter kann die Strategie des DBH über die DBH-Option SERVICE-TASKS oder im laufenden Betrieb mit der Administrationsanweisung MODIFY-SERVICE-TASKS beeinflussen.

5.12.1 Service-Task-Strategie beeinflussen

Im Parameter INITIAL kann der Systemverwalter einstellen, wieviele Service-Tasks beim Hochfahren des DBHs oder im laufenden Betrieb sofort gestartet werden sollen. Wird dieser Wert zu hoch gewählt, verzögert dies den Ladevorgang, da sich die Tasks beim Hochfahren durch Zugriffe auf Memory-Pool und Modulbibliotheken gegenseitig behindern. Mit der Standardeinstellung INITIAL=1 ist gewährleistet, dass sofort eine Service-Task arbeitsbereit ist.

Im Parameter MAXIMUM kann der Systemverwalter einstellen, wieviele Service-Tasks maximal parallel in der Session für den DBH arbeiten können. Mit dem Performance-Monitor, Ausgabemaske „SERVICE TASKS“, kann der Systemverwalter die Auslastung der Service-Tasks beobachten. Hat „Orders-Not-Processed“ einen Wert größer 0, gibt es einen Auftragsstau. Der Wert für MAXIMUM sollte im laufenden Betrieb mit der Administrationsanweisung MODIFY-SERVICE-TASKS oder in der Folgesession angehoben werden.

Im Parameter JOBCLASS kann der Systemverwalter festlegen, mit welcher Jobklasse der DBH die Service-Tasks starten soll. Die Jobklasse kann auch im laufenden Betrieb mit der Administrationsanweisung MODIFY-SERVICE-TASKS eingestellt werden.
Es sollte eine Jobklasse mit mindestens 1000 CPU-Sekunden Zeitlimit sein, ansonsten stürzen die Service-Tasks zu oft wegen CPU-Zeitlimitablauf ab. Betroffene SQL-Anweisungen werden dann mit dem SQLSTATE 81SS0 abgebrochen.

Erlaubt der JOIN-Eintrag der DBH-Kennung NO-CPU-LIMIT=YES, werden die Service-Tasks mit CPU-LIMIT=*NO gestartet und haben somit keine Beschränkung der CPU-Zeit.

Der JOIN-Eintrag der DBH-Kennung sollte START-IMMED=YES erlauben, damit die Service-Tasks mit START=*IMMED gestartet werden können und nicht zu lange im Auftragstyp T1 verweilen.

Wenn CREATE INDEX-Anweisungen für partitionierte Tabellen ausgeführt werden sollen, dann müssen genügend Service-Tasks für die parallele Verarbeitung zur Verfügung stehen. Siehe Abschnitt „Optimierter Indexaufbau für partitionierte Tabellen“ auf Seite 230.

5.12.2 Sortierung beeinflussen

Bei der Abarbeitung einer Anweisung im DBH müssen Eingabedaten, Zwischenergebnismengen, Ergebnis-Cursor-Dateien usw. sortiert werden. Um den DBH zu entlasten, werden diese Sortierungen in Service-Tasks durchgeführt.

Art und Geschwindigkeit der Sortierung sind von folgenden Faktoren abhängig:

- Anzahl der Sortierkriterien
- Sortierreihenfolge im Modul SESFS90 (siehe Handbuch „Datenbankbetrieb“)
- Parameter WORK-FILES
- Parameter RECORDS-PER-CYCLE
- Parameter CORE der SORT-Voreinstellungsprozedur
- JOIN-Eintrag der DBH-Kennung

Anzahl der Sortierkriterien

Der DBH entnimmt die Sortierkriterien der ORDER BY-Angabe einer SQL-Anweisung bzw. der S-Teilfabfrage einer CALL-DML-Suchfrage. Sortierungen mit bis zu 2 Sortierkriterien werden mit dem SESAM/SQL-eigenen SORT-Baustein durchgeführt - allerdings nur unter folgenden Voraussetzungen: Zuerst wird im DBH geprüft,

1. ob die Summe der Längen der Sortierkriterien kleiner oder gleich 512 Byte ist
2. ob die Sortierreihenfolge im Modul SESFS90 nicht verändert wurde und
3. ob die zu sortierende Menge (Anzahl Sätze mal Satzlänge) kleiner als ein Pufferrahmen des Cursor-Puffers (DBH-Option FRAME-SIZE) ist.

Verläuft diese Prüfung positiv, erfolgt die Sortierung im DBH mit dem SESAM/SQL-eigenen SORT-Baustein. Verläuft sie negativ, so wird die Sortierung an eine Service-Task übergeben. Dort erfolgt auch eine Prüfung. Dabei wird zusätzlich zu den zwei erstgenannten DBH-Prüfungen noch kontrolliert, ob die zu sortierende Menge (Anzahl Sätze mal Satzlänge) kleiner oder gleich 96 Kilobyte ist. Verläuft diese Prüfung positiv, erfolgt die Sortierung (jetzt in einer Service-Task) mit dem SESAM/SQL-eigenen SORT-Baustein. Verläuft sie negativ, dann wird mit dem Softwareprodukt SORT sortiert. In jedem Fall erfolgt aber in der Service-Task eine Sortierung über das Softwareprodukt SORT, wenn mehr als 2 Sortierkriterien vorhanden sind.

Parameter WORK-FILES

Arbeits- und Hilfsdateien für die BS2000-Sortierung werden standardmäßig auf dem Defaultpubset der Kennung angelegt.
Im Parameter WORK-FILES der DBH-Option SERVICE-TASKS kann der Systemverwalter einen Datenträger seiner Wahl angeben (PUBLIC-DISK oder PRIVATE-DISK), auf dem die Arbeits- und Hilfsdateien gespeichert werden sollen. Der Datenträger sollte über ca. 100.000 freie PAM-Seiten verfügen und reorganisiert sein, damit beim Anlegen der Dateien möglichst wenige Extents entstehen. Durch Vergabe einer ausreichend großen Anzahl PAM-Seiten beim Parameter PRIMARY-ALLOCATION, kann der Systemverwalter das Erweitern der Dateien während der Sortierung unterbinden.
Wie die Größe der Arbeits- und Hilfsdateien für die BS2000-Sortierung berechnet wird, ist im Sort-Handbuch beschrieben (siehe Handbuch „SORT (BS2000)“).

Wurden in der DBH-Option PRIVATE-DISK bei VOLUME mehrere Privatplatten angegeben, so werden die Arbeits- und Hilfsdateien so angelegt, dass sie in jeder Service-Task (DBH-Option MAXIMUM) auf einen anderen VOLUME beginnen.

Beispiel

DBH-Option VOLUME:

```
VOLUME=(DISK01,DISK02,DISK03)
```

Arbeits- und Hilfsdateien werden in den Service-Tasks folgendermaßen angelegt:

in der 1. Service-Task: `VOLUME=(DISK01,DISK02,DISK03)`
in der 2. Service-Task: `VOLUME=(DISK02,DISK03,DISK01)`
in der 3. Service-Task: `VOLUME=(DISK03,DISK01,DISK02)`
in der 4. Service-Task: `VOLUME=(DISK01,DISK02,DISK03)`
usw.

Parameter RECORDS-PER-CYCLE

Das Softwareprodukt SORT bietet die Möglichkeit, die zu sortierende Menge gleichmäßig auf bis zu 9 Sort-Subtasks zu verteilen. Wieviele Sätze in einer Sort-Subtask sortiert werden sollen, kann der Systemverwalter im Parameter RECORDS-PER-CYCLE der DBH-Option SERVICE-TASKS oder im laufenden Betrieb in der Administrationsanweisung MODIFY-SERVICE-TASKS angeben (siehe auch RECORDS-PER-CYCLE-Operand im Handbuch „SORT (BS2000)“).

Eine Multitask-Sortierung des Softwareprodukts SORT benötigt maximal doppelt so viel Plattenspeicher wie eine Singletask-Sortierung, da die Sort-Subtasks ihre Daten über Hilfsdateien austauschen. Der DBH-Parameter RECORDS-PER-CYCLE sollte also nur dann angegeben werden, wenn auf dem Pubset genügend freier Speicher vorhanden ist und eine echte Parallelität der Sort-Subtasks (Multiprozessor) gegeben ist.

Der Wert des Parameters RECORDS-PER-CYCLE sollte nicht kleiner als 1.000.000 sein. Ist die Anzahl Sätze, die eine Sort-Subtask in einem Zyklus sortieren soll, zu niedrig gewählt, wächst der Kommunikationsaufwand derart, dass die Multitask-Sortierung keinen Performance-Vorteil mehr bietet.

Bei Multitask-Sortierung ist weiterhin Folgendes zu beachten:

Werden die zu sortierenden Daten der Service-Task in einer Cursordatei übergeben, z.B. bei SQL-Anweisungen mit „ORDER-BY“, steht die zu sortierende Menge bei der Initialisierung des Softwareprodukts SORT genau fest.

Die Anzahl der Sortzyklen wird berechnet aus:

- Anzahl Sortzyklen = Anzahl Sätze / RECORDS-PER-CYCLE
- Anzahl Sort-Subtasks = Anzahl Sortzyklen - 1

Für die berechnete Anzahl Sortzyklen werden Arbeitsdateien und für die berechnete Anzahl Sort-Subtasks Hilfsdateien angelegt. Die maximale Anzahl Sortzyklen ist auf 9 begrenzt. Es werden also maximal 8 Sort-Subtasks gestartet.

Werden die zu sortierenden Daten in der Service-Task während der Verarbeitung ermittelt, z.B. bei DDL-Anweisungen „MIGRATE“, „CREATE INDEX“ usw., steht die zu sortierende Menge bei der Initialisierung des Softwareprodukts SORT nicht fest. Die Anzahl der Sortzyklen wird immer auf das Maximum von 9 eingestellt. Entsprechend werden die Arbeits- und Hilfsdateien angelegt.

Parameter CORE der SORT-Voreinstellungsprozedur

Die Voreinstellungen können Server- und kennungsspezifisch mit dem Kommando MODIFY-SORT-DEFAULTS eingestellt werden. Hier kann auch die Jobklasse für die Sort-Subtasks eingestellt werden. Die Voreinstellungen werden in der Datei SYSPAR.SORT.vvv (vvv = Version des SORT) gespeichert. Genaue Beschreibung des Kommandos siehe im Handbuch „SORT (BS2000)“.

JOIN-Eintrag der DBH-Kennung

Der JOIN-Eintrag der DBH-Kennung sollte START-IMMED=YES und NO-CPU-LIMIT=YES erlauben (siehe Abschnitt „Service-Task-Strategie beeinflussen“ auf Seite 97).

Damit das Softwareprodukt SORT den in CORE angegebenen Wert voll ausnutzen kann, ist es ratsam, ADDRESS-SPACE-LIMIT=32767 zu wählen.

5.12.3 Speicherbedarf für Sort-Arbeitsdateien beim Indexaufbau

Der Plattenspeicherbedarf für einen Index beträgt (in Halfpages):

```
(anzahl * satzlänge / 2048)
```

`anzahl` ist die Anzahl der zu sortierenden Sätze.
Aus jedem Satz der Tabelle, in dem wenigstens ein Wert einer Spalte, aus denen der Index aufgebaut ist, signifikant ist, wird ein Sort-Satz aufgebaut. Maximalwert ist also die Anzahl Sätze der Tabelle.

`satzlänge` ist die Länge eines Sort-Satzes.
Sie ergibt sich aus einem festen Bestandteil der Länge 8 (Längenfeld, Index-ID, Satznummer) plus der Summe der Invertierungslängen der Spalten, aus denen der Index zusammengesetzt ist. Invertierungslänge ist die bei der Indexdefinition angegebene Länge, bzw., wenn nicht angegeben, die Länge der Spalte. Ist die einzige oder letzte Spalte des Index vom Typ CHARACTER, tragen Blanks am Ende des Sort-Satzes nicht zur Länge bei.

Wenn mehrere Indizes auf einmal definiert werden, so ist der Bedarf für die einzelnen Indizes zu addieren.

5.12.4 Zusammenfassung performance-steigernder Maßnahmen

Im Folgenden sind Kriterien und Maßnahmen zusammengefasst, die sich bei der Bearbeitung von Service-Aufträgen positiv auf die Performance auswirken:

- Parameter der DBH-Option SERVICE-TASKS:

INITIAL:	1
MAXIMUM:	Mit Hilfe des Performance-Monitors ermittelter Wert
PRIMARY-ALLOCATION:	Ausreichend große Anzahl PAM-Seiten
RECORDS-PER-CYCLE:	Wert größer 1.000.000 (auf Multiprozessor-Servern)
WORK-FILES:	Datenträger (schnell) für Arbeits- und Hilfsdateien mit ca. 100.000 freien PAM-Seiten (PUBLIC-DISK oder PRIVATE-DISK)

- JOIN-Eintrag der DBH-Kennung:

START-IMMED =	YES
NO-CPU-LIMIT =	YES
ADDRESS-SPACE-LIMIT =	32767

- SORT-Parameterdatei:

CORE =	32767

- Anzahl Sortierkriterien: ≤ 2
- Länge der Sortierkriterien: ≤ 512 byte
- Anzahl Sätze * Satzlänge: ≤ 96 Kbyte
- Sortierreihenfolge im Modul SESFS90 nicht verändern. Das Modul enthält eine Tabelle, die die Sortierreihenfolge festlegt (siehe Handbuch „Datenbankbetrieb“).

5.12.5 RECOVER/REFRESH-Optionen

Mit der DBH-Option RECOVER-OPTIONS stellen Sie folgende Optionen ein, die bei einem RECOVER- oder REFRESH-Lauf für den DBH in der Service-Task verwendet werden:

- die Größe des Puffers für Systemzugriffsdaten
- die Größe des Puffers für Anwenderdaten
- die Speicherinformationen für die Transaktionssicherungsdateien (TA-LOG-Dateien)
- die Speicherinformationen für die Wiederanlauf-Sicherungsdatei (WA-LOG-Datei)

Sie können die Werte der Parameter mit der Administrationsanweisung MODIFY-RECOVER-OPTIONS während der DBH-Session anpassen.

Im Vergleich zu Änderungen anderer DBH-Optionen im laufenden Betrieb ist zu beachten, dass sich die Änderung der RECOVER-OPTIONS mit MODIFY-RECOVER-OPTIONS erst auf kommende RECOVER- bzw. REFRESH-Läufe auswirkt.
Der Erfolg einer Änderung der RECOVER-OPTIONS kann am besten über eine Folge vergleichbarer RECOVER- bzw. REFRESH-Läufe kontrolliert werden. Dies ist z.B. bei der regelmäßigen Pflege eines Replikats der Fall.
Weiterhin können Sie die im SYSLST-Protokoll des Recover-Laufs enthaltene IO-Statistik bzgl. der Hitraten von Pufferzugriffen auswerten. Damit prüfen Sie die Güte der verwendeten Puffergrößen für den Recover (siehe Abschnitt „User-Data-Buffer dimensionieren“ auf Seite 74 und Abschnitt „System-Data-Buffer dimensionieren“ auf Seite 75).

Bei den Angaben zur TA-LOG-Datei ist zu beachten, dass nicht nur die Speichergrößen ausreichend sein müssen, sondern dass bei Privatplattenbenutzung auch ein ausreichend schnelles Gerät (DEVICE-TYPE) benutzt wird. Andernfalls würde, da auf die TA-LOG-Datei regelmäßig geschrieben wird, ein RECOVER- oder REFRESH-Lauf durch Ein-/Ausgabeengpässe gebremst werden.
Bei genügend großer Anfangszuweisung (PRIMARY-ALLOCATION) ergibt sich zudem der Vorteil, dass keine oder nur wenige Dateierweiterungen (SECONDARY-ALLOCATION) nötig sind. Eine Fragmentierung der TA-LOG-Datei wird dadurch vermieden.

Die Einstellungen für die WA-LOG-Datei sind weniger kritisch, weil normalerweise nur wenige Ein-/Ausgaben auf die WA-LOG-Datei anfallen.
Falls dennoch regelmäßige Ein-/Ausgaben auf die WA-LOG-Datei beobachtet werden, so deutet dies auf eine vorzeitige Verdrängung geänderter Blöcke wegen zu kleiner Puffer für Systemzugriffsdaten oder für Anwenderdaten hin. Dies kann dann durch eine Vergrößerung von SYSTEM-DATA-BUFFER und USER-DATA-BUFFER verbessert werden.

5.13 Reorganisationskriterien

Im Folgenden werden die Möglichkeiten beschrieben, zu erkennen, wann eine Reorganisation eines Spaces sinnvoll ist und was bei einer Volume-Reorganisation mit SPACEOPT zu beachten ist.

5.13.1 Physikalische DB-Struktur und ihre Veränderung im laufenden Betrieb

Nach der Migration oder dem Zuladen einer Tabelle mittels der Utility-Anweisungen MIGRATE bzw. LOAD OFFLINE sind die Datenbankblöcke entsprechend dem eingestellten Blockfüllgrad gefüllt. Die Sätze der Tabelle sind nach dem Primärschlüssel aufsteigend sortiert in den Datenblöcken gespeichert. Die logische Verkettung der Blöcke entspricht in der Regel ihrer physikalischen Lage.

Werden nur Sätze an das Ende der Tabelle über DML angefügt, verändert sich an diesem Zustand nichts. Die Tabelle bleibt reorganisiert. Werden hingegen Sätze zwischen die existierenden Sätze eingefügt oder bestehende Sätze verlängert, so dass der freigehaltene Platz auf dem Block nicht ausreicht, so wird ein Auslagerungsblock logisch eingefügt. D.h. die logische Verkettung zeigt nicht mehr auf den Block mit der nächsthöheren Blocknummer, sondern auf den Auslagerungsblock.

Gleiches kann durch Löschen von vielen Sätzen geschehen. Werden Sätze gelöscht und werden durch das Löschen Blöcke frei, so werden die Blöcke ausgekettet und an die Freiplatzverwaltung gegeben. Wird beim Einfügen oder bei der Änderung eines Satzes ein neuer Block benötigt, so werden die Blöcke der Freiplatzverwaltung verwendet.

Ungünstiges Löschen und Neuaufnehmen von Sätzen kann zu schlechter Blocknutzung führen, insbesondere wenn der Blockfüllungsgrad zu hoch gewählt wurde, so dass immer Blöcke freigegeben werden und Auslagerungsblöcke angelegt werden.

5.13.2 Indizien für die Notwendigkeit einer Reorganisation

Folgende Anhaltspunkte gibt es, um die Notwendigkeit einer Reorganisation zu erkennen:

- Die Buffer-Hitrate (SESMON DBH-Maske I/O) nimmt ab, obwohl der Datenbestand (Anzahl Sätze) nicht wächst und das Anwendungsprofil sich nicht geändert hat.
- Die Größe der Spaces nimmt zu, obwohl sich der Datenbestand nicht vergrößert hat.
- Die Spacestatistik von SESDIAG (siehe den folgenden Unterabschnitt) weist viele Blöcke mit geringem Blockfüllungsgrad auf.

5.13.3 Spacestatistik von SESDIAG

Mit der Funktion Spacestatistik von SESDIAG kann der Inhalt des Spacestatistikblocks auf SYSLST ausgegeben werden. Im Spacestatistikblock wird festgehalten, wie der Blockfüllungsgrad der in den DBH-Tasks eingelesenen Blöcke für diesen Space war. Hierbei werden nur die Primärdatenblöcke und die Blöcke der untersten Hierarchiestufe des Sekundärindexes berücksichtigt. Die Ausgabe erfolgt in der Form:

- Erfassungszeitraum durch zwei Zeitstempel.
- Ausgabe der absoluten und relativen Anzahl Blöcke und deren Blockfüllungsgrad in Intervallen von 10 Prozent.

Es werden die letzten 54 Sessions, in denen Datenänderungen stattgefunden haben, ausgegeben. (Nur bei Änderungen wird die Information in den Space zurückgeschrieben.) Eine Session ist der Zeitraum zwischen physikalischem Open und physikalischem Close eines Spaces. Findet eine Reorganisation statt, so wird diese mit Zeitstempel eingetragen.

Beim Übergang eines Spaces von der DBH-Task zur Service-Task bei Utilities wird der Space physikalisch geschlossen.

Eine Kurzbeschreibung zu SESDIAG finden Sie in der SESAM/SQL-Bibliothek SIPANY.SESAM-SQL.090.TOOLS

5.13.4 Reduzierung der Anzahl Extents einer BS2000-Datei

Durch das ständige Anlegen, Löschen, Vergrößern und Verkleinern von Dateien im laufenden Betrieb kommt es auf den Volumes eines Pubsets zu einer immer stärkeren Fragmentierung des freien Speicherbereichs und der anzulegenden Dateien. Die Fragmentierung beeinträchtigt in zunehmendem Maße die Performance der Dateizugriffe und die gleichmäßige Verteilung der I/O-Last über alle Volumes des Pubsets. Die Extent-Liste in den Katalogeinträgen wird durch das zwangsläufige Anlegen mehrerer kleiner Datei-Extents beim Vergrößern der Dateien verlängert.

Das kostenpflichtige Softwareprodukt SPACEOPT bereinigt eine Fragmentierung durch die optimale Verlagerung (Reorganisation) der Datei-Extents auf den Volumes eines Pubsets, siehe Handbuch „SPACEOPT (BS2000)“. Ziel von SPACEOPT ist es, auf einem Volume möglichst große, zusammenhängende, freie Speicherbereiche zu schaffen, damit das Einrichten großer Dateien mit einer geringeren Anzahl an Extents erfolgen kann.

Mit SPACEOP können auch geöffnete Dateien reorganisiert werden.
Folgendes ist aber zu beachten:

- Das Reorganisieren von Dateien ist mit einem physikalischen Verlagern von Datei-Extents verbunden. Deshalb sollte diese Funktion nur zu Zeiten durchgeführt werden, in denen die I/O-Last der betroffenen Volumes gering ist.
- Für das Reorganisieren der Dateien sperrt SPACEOPT temporär die entsprechenden BS2000-Katalogeinträge.
  Soll eine betroffene Datei geöffnet oder erweitert werden, so unterbricht SPACEOPT seine Verarbeitung, gibt seine Sperre auf, lässt den Auftrag passieren. SPACEOPT setzt dann nach erneutem Erwerb der Katalogsperre seine Verarbeitung fort. Dieser Vorgang wird pro Datei bis zu dreimal wiederholt, bevor die Datei von der Reorganisation ausgenommen wird.

5.13.5 Reorganisation eines Spaces mit Neuvergabe der Satznummern

Die Satznummern werden von SESAM/SQL beim Einfügen von Sätzen in die Tabelle (Funktionen INSERT, LOAD) vergeben. Beim Löschen von Sätzen werden sie intern in den sogenannten Satznummernverwaltungsblöcken verwaltet. Mit Hilfe der Satznummern und der Database-Translation-Table (DBTT) wird die Verbindung zwischen den Werten im Sekundärindex und den Primärdatensätzen hergestellt. Die Satznummer wird hierzu in den Sekundärindizes hinter den entsprechenden Werten gespeichert. Sie bildet außerdem den Zugriffindex innerhalb der DBTT zum Auffinden des Verweises auf den zugehörigen Primärdatensatz. Die Größe der DBTT wird bestimmt durch die bisher höchste vergebene Satznummer. Im Laufe der Zeit kann durch Löschen und Einfügen von Sätzen die Streuung der Satznummern in einem Primärschlüssel-Spektrum groß werden. Hat die Tabelle den Charakter einer „History-Table“, in der für gewisse Zeit Daten aufgehoben und dann zyklisch gelöscht werden (z.B. die Buchungsvorgänge für ein Bankkonto), so empfiehlt es sich, diesen Reorganisationszyklus zeitlich in kleineren Abständen mit dann entsprechend geringeren Datenmengen durchzuführen. Größere Zeitabstände mit entsprechend größeren Datenmengen führen zu unnötigem Aufblähen der DBTT und der Satznummernverwaltung. Die Folgen sind schlechtere Buffer-Hitraten im SYSTEM-DATA-Buffer.

Eine Satznummernverwaltung mit vielen freien Satznummern kann auch zu einer Verschlechterung der Nachfahrzeiten von INSERT-Anweisungen beim RECOVER führen, da bei der Recovery die gleichen Satznummern wie im Original verwendet werden müssen. Damit muss die Satznummer eventuell sequenziell in der Satznummernverwaltung gesucht werden.

Ist die Situation eingetreten, dass zu viele freie Satznummern existieren oder dass die Streuung der vergebenen Satznummern recht hoch ist, so kann die Funktion REORG SPACE mit dem Parameter NEW ROW_IDS helfen. Hierbei ist jedoch zu beachten, dass mit der Neuvergabe der Satznummern alle Indizes auf defekt gesetzt werden und die logische Datensicherung unterbrochen wird.

Einen Hinweis, wie viele freie und verwaltete Satznummern für eine Tabelle existieren, erhält man über das Tool SESDIAG Funktion 3 (Ausgabe der Tabelleninformation).

Eine Kurzbeschreibung zu SESDIAG finden Sie in der SESAM/SQL-Bibliothek SIPANY.SESAM-SQL.090.TOOLS

5.14 Verteilte Verarbeitung

Bei der verteilten Verarbeitung mit SESAM/SQL-DCN hat der Systemverwalter folgende Möglichkeiten, auf die Performance einzuwirken:

- Dimensionieren des SESDCN-Memory-Pools
- Starten mehrerer SESDCNs

5.14.1 Dimensionieren des SESDCN-Memory-Pools

Die Größe des SESDCN-Memory-Pools kann der Systemverwalter über den Parameter USERS der Steueranweisung SET-DCN-OPTIONS steuern. Abhängig vom USERS-Wert, wird bei der Initialisierung des Pools durch den Master-DCN eine bestimmte Anzahl Container zu je 256 Byte angelegt (ca. 4 * USERS-Wert, also ca. 1KB je User). Dies ist jedoch nur ein Orientierungswert. Je nach Anforderungsprofil eines Users kann der tatsächliche Platzbedarf hiervon abweichen.

Eine zu große Dimensionierung des Pools führt zu unnötigem Adressraumverbrauch in den an diesem Pool hängenden Tasks (SESDCNs, DBHs, Anwenderprogramme). Eine zu niedrige Dimensionierung führt zu Ressourcen-Engpässen, die sich in häufigem Auftreten von SQLSTATE 91SC5 bzw. Status 2B/UG bemerkbar machen.

Die Maske CAPACITY des Performance-Monitors weist die tatsächliche Anzahl belegter und freier Container aus (Used Pool Elements und Free Pool Elements). Diese Information kann der Systemverwalter für eine bedarfsgerechte Pool-Dimensionierung nutzen.

5.14.2 Starten mehrerer SESDCNs

Nachrichten aus fremden Konfigurationen, die an einen DBH einer bestimmten Konfiguration (K) gerichtet sind, werden durch einen SESDCN dieser Konfiguration K empfangen und dann an den angesprochenen DBH weitergeleitet. Dieses Empfangen und Weiterleiten von Nachrichten erfolgt in asynchronen Contingency-Prozessen der SESDCN-Task. Eine hohe Auftragslast kann dazu führen, dass eine SESDCN-Task durch diese Aktivitäten vollständig ausgelastet ist, oder dass es sogar zu einem Nachrichtenstau vor der SESDCN-Task kommt. Der SESDCN-Task bleibt dann u. U. keine Zeit mehr für andere Aufgaben wie z.B. Administration und Lock-Überwachung.

Zur Vermeidung eines solchen Engpasses kann der Systemverwalter in einer Konfiguration mehrere SESDCN-Tasks starten. Umso eine Lastverteilung zu erreichen, ist es allerdings erforderlich, den zu dieser Konfiguration gehörenden Datenraum auf mehrere DBHs aufzuteilen, damit jeder SESDCN-Task mindestens ein DBH zugeordnet werden kann.

Die Maske CAPACITY des Performance-Monitors weist aus, wie stark die SESDCN-Task(s) einer Konfiguration durch asynchrone Aktivitäten ausgelastet ist (sind) (Elapsed time in Contingency). Diese Information kann als Entscheidungshilfe dafür dienen, ob es sinnvoll bzw. notwendig ist, zur Lastverteilung weitere SESDCN-Tasks in dieser Konfiguration zu starten.

5.15 Analyse von Sperrkonflikten

Für die Diagnose von Sperrkonflikten können der Performance-Monitor SESMON (siehe Handbuch „Datenbankbetrieb“) und der View SYS_LOCK_CONFLICTS benutzt werden, siehe Abschnitt „Diagnose und Maßnahmen bei Performance-Problemen“ auf Seite 70.

In der SYSLST-Ausgabe der SESMON-Maske TRANSACTIONS sind diverse Informationen zu gesperrten Transaktionen zu finden.

Erweiterte Informationen enthält der View SYS_LOCK_CONFLICTS, der mit SQL-Mitteln gelesen werden kann. Dieser View enthält die 1000 zuletzt aufgetretenen Sperrkonflikte mit detaillierten Informationen über Sperrtypen, gesperrte Objekte und Verursacher. Beim Zugriff auf den View werden aber nur die interen IDs der betroffenen Objekte ausgegeben, so dass die zugehörigen Objektnamen ermittelt werden müssen.

Durch die Kombination der Views SYS_LOCK_CONFLICTS und SYS_DBC_ENTRIES (Informationen über die aktiven Datenbanken) kann die Sperrinformation mit den Objektnamen ausgegeben werden. Ein SELECT auf den im Folgenden definierten View liefert dann Sperrinformationen über die letzten 1000 Sperren, die im DBH aufgetreten sind. Dabei werden alle Objekte mit den entsprechenden Namen ausgegeben.

Es wird z.B. folgender View definiert:

```
CREATE VIEW LOCK_INFO AS
   SELECT ALL TIME_OF_CONFLICT, OBJECT_TYPE, DBC_NUMBER,
              GENERAL_LOCKS.CATALOG_NAME AS CATALOG_NAME, SPACE_ID,
              CASE WHEN DBC_NUMBER IS NOT NULL AND SPACE_ID < 32767
                   THEN COALESCE (
                        (SELECT SPACE_NAME FROM SYS_INFO_SCHEMA.SYS_SPACES
                         WHERE SPACE_ID = GENERAL_LOCKS.SPACE_ID), '???')
                   ELSE '---'
              END AS SPACE_NAME,
              TABLE_ID,
              CASE WHEN DBC_NUMBER IS NOT NULL AND TABLE_ID < 30720
                        AND SPACE_ID < 32767
                   THEN COALESCE (
                        (SELECT TABLE_NAME FROM SYS_INFO_SCHEMA.SYS_TABLES
                         WHERE TABLE_ID = GENERAL_LOCKS.TABLE_ID
                         AND SPACE_ID = GENERAL_LOCKS.SPACE_ID), '???')
                   ELSE '---'
              END AS TABLE_NAME,
              INDEX_ID,
```

```
        CASE WHEN DBC_NUMBER IS NOT NULL AND INDEX_ID IS NOT NULL
             THEN COALESCE (
                  (SELECT INDEX_NAME FROM SYS_INFO_SCHEMA.SYS_INDEXES
                   WHERE INDEX_ID = GENERAL_LOCKS.INDEX_ID
                   AND SPACE_ID = GENERAL_LOCKS.SPACE_ID
                   AND ORDINAL_POSITION = 1), '???')
             ELSE '---'
        END AS INDEX_NAME,
        CASE WHEN DBC_NUMBER IS NOT NULL AND INDEX_ID IS NOT NULL
             THEN COALESCE (
                  (SELECT TABLE_NAME FROM SYS_INFO_SCHEMA.SYS_INDEXES
                   WHERE INDEX_ID = GENERAL_LOCKS.INDEX_ID
                   AND SPACE_ID = GENERAL_LOCKS.SPACE_ID
                   AND ORDINAL_POSITION = 1), '???')
             ELSE '---'
        END AS INDEX_TABLE_NAME,
        ROW_ID, SI_VALUE, PLAN_ID, META_SCHEMA, META_TABLE, HOST_NAME,
        APPLICATION_NAME, CUSTOMER_NAME, CONVERSATION_ID, TAC_NAME,
        MODULE_NAME, STATEMENT_NAME, STATEMENT_TYPE, LOCK_MODE,
        LOCK_TYPE, REQUEST_ANNOUNCED, LOCKING_OBJECT_TYPE,
        LOCKING_HOST_NAME, LOCKING_APPLICATION_NAME,
        LOCKING_CUSTOMER_NAME, LOCKING_CONVERSATION_ID,
        LOCKING_LOCK_MODE

FROM (SELECT SYS_INFO_SCHEMA.SYS_LOCK_CONFLICTS.*, CATALOG_NAME
      FROM SYS_INFO_SCHEMA.SYS_LOCK_CONFLICTS
      LEFT JOIN SYS_INFO_SCHEMA.SYS_DBC_ENTRIES
      ON SYS_INFO_SCHEMA.SYS_LOCK_CONFLICTS.DBC_NUMBER
         = SYS_INFO_SCHEMA.SYS_DBC_ENTRIES.DBC_NUMBER)
      AS GENERAL_LOCKS

WHERE DBC_NUMBER IS NULL
      OR DBC_NUMBER = (SELECT DBC_NUMBER
         FROM SYS_INFO_SCHEMA.SYS_DBC_ENTRIES
         WHERE CATALOG_NAME = CURRENT_REFERENCED_CATALOG)
```

6 Gezielte Hinweise zum Tuning von Anwendungen

Das folgende Kapitel gibt Hinweise zum Tuning von Anwendungen, getrennt nach SQL- und CALL-DML-Anwendungen.

6.1 Optimierungsmöglichkeiten bei SQL-Anwendungen

Dieser Abschnitt gibt zunächst allgemeine Empfehlungen zum Programmieren von SQL-Anwendungen und befasst sich dann mit dem Tuning von SQL-Anweisungen.

6.1.1 Allgemeine Programmierempfehlungen

Für das Programmieren von SQL-Anwendungen gibt es einige Empfehlungen, deren Beachtung sich positiv auf die Performance auswirkt. Diese Programmierempfehlungen beziehen sich auf folgende Themenkreise:

- Deskriptoren und präparierte SQL-Anweisungen
- Anzahl Spalten in SELECT-Listen und Update-Anweisungen
- SQL-Cursor im Schubmodus
- Maximallänge von VARCHAR

Deskriptoren und präparierte SQL-Anweisungen

Deskriptoren und präparierte Anweisungen belegen Ressourcen des SQL- RTS (Vorgangsmemory von openUTM) und des SQL-DBH (Planpuffer).

Folgende Maßnahmen reduzieren den Ressourcenbedarf:

- So weit möglich Folgendes bevorzugen:
  - statische vor dynamischer SQL
  - bei dynamischer SQL „USING *benutzervariable* ...“ vor „USING SQL DESCRIPTOR ...“
- Möglichst wenige SQL-Deskriptorbereiche allozieren; möglichst nur einen einzigen Input- und Output-Deskriptor definieren und diesen für alle präparierten Anweisungen verwenden.
- Große Deskriptorbereiche vor Ende des Dialogschritts (auch PEND KP) freigeben. Das Vorgangsmemory wird von openUTM bei Dialogschrittwechsel gesichert.
- Anweisungsnamen für präparierte SQL-Anweisungen wiederverwenden (implizite Freigabe der vorher präparierten SQL-Anweisung).

Anzahl Spalten in SELECT-Listen und UPDATE-Anweisungen

Die Anzahl der Spalten in einer SELECT-Liste hat erheblichen Einfluss auf Ressourcen-Verbrauch (s.o.) und Performance.

Nach Möglichkeit sollte der Anwender daher Folgendes beachten:

- eine SELECT-Liste der Form * vermeiden
- grundsätzlich nur die benötigten Spalten angeben.

Entsprechendes gilt auch für UPDATE-Anweisungen: In der SET-Klausel einer UPDATE-Anweisung sollten lediglich die tatsächlich zu ändernden Spalten angegeben werden.

SQL-Cursor im Schubmodus

Sollen in einem ESQL-COBOL-Anwendungsprogramm mehrere Ergebnissätze eines Cursors in einer Folge von FETCH-Aufträgen ermittelt werden, kann man durch Einsatz des Schubmodus (Pragma PREFETCH) eine deutliche Performance-Verbesserung erzielen (siehe Handbuch „SQL-Sprachbeschreibung Teil 1: SQL-Anweisungen“). Im Schubmodus werden Mengen von Ergebnissätzen eines Prefetch-Cursors mit einem einzigen Auftrag vom DBH zur Anwendertask transportiert. Bei nachfolgenden FETCH-Aufträgen entfallen dadurch Kommunikationsschritte zwischen Anwendertask und DBH.

Der Performance-Gewinn durch den Schubmodus hängt vor allem davon ab, wie teuer die Kommunikation zwischen Anwendung und DBH ist. Bereits in lokaler Umgebung (Anwendung und DBH auf einem Rechner) ist eine Beschleunigung festzustellen. Von besonderer Bedeutung ist der Schubmodus jedoch in verteilter Umgebung, wenn Anwendungsprogramm und DBH auf verschiedenen Rechnern ablaufen.

Einschränkungen

- Der Prefetch-Cursor darf nicht in den Anweisungen UPDATE ... WHERE CURRENT OF *cursor* und DELETE ... WHERE CURRENT OF *cursor* verwendet werden.
- Der Prefetch-Cursor darf nicht mit SCROLL vereinbart werden.
- Der Prefetch-Cursor darf nicht mit der Anweisung STORE *cursor* bearbeitet werden.

Randbedingungen

- Der FETCH-Auftrag für den ersten Satz eines Schubes ist etwas langsamer als ein einzelner FETCH-Auftrag. Nachfolgende FETCH-Aufträge sind dafür schneller.
- Die Ergebnisschübe werden in der Anwendertask zwischengespeichert. Dafür fällt zusätzlicher Verwaltungsaufwand an und der Hauptspeicherbedarf der Anwendertask wird größer.

Einstellungen

- Die Übertragung der Schübe vom DBH zum Anwendungsprogramm erfordert eine entsprechende Größe der Nachrichtenlänge. Der Parameter PUF in der Konfigurationsdatei der UTM-Anwendung sollte daher möglichst groß eingestellt werden.
- Die Schübe von Cursorn werden im Prefetch-Puffer verwaltet. Die Größe des Prefetch-Puffers legt der Anwender in der Konfigurationsdatei des Anwendungsprogramms fest (siehe Abschnitt „Größe des Prefetch-Puffers“ auf Seite 92). Die Auslastung des Prefetch-Puffers kann er mit dem Performance-Monitor überprüfen.
- Die Anzahl der Sätze in einem Schub legt der Anwender als Blockungsfaktor (n) über das Pragma PREFETCH fest. Der Blockungsfaktor sollte den Belangen des SQL-Anwendungsprogramms entsprechen. Falls die Ergebnisse von Cursorn in Teilmengen gleicher Größe weiterverarbeitet werden, bietet sich z.B. die Kardinalität der Teilmenge als Blockungsfaktor an.
  Ergibt sich aus der Logik des Anwendungsprogramms kein geeigneter Blockungsfaktor, sollten folgende Aspekte berücksichtigt werden:
  - In einer SQL-Anwendung konkurrieren alle im Schubmodus geöffneten Cursor um Speicher im Prefetch-Puffer. Einmal belegter Speicher bleibt einem Cursor solange zugeordnet, bis der Cursor geschlossen wird. Die Größe des benötigten Speicherbereichs ergibt sich aus der Ergebnissatzlänge und dem als Blockungsfaktor vereinbarten Wert.
    Kann für einen Prefetch-Cursor wegen eines Engpasses kein Speicherbereich im Prefetch-Puffer belegt werden, wird das Pragma PREFETCH ignoriert, der Schubmodus wird nicht eingeschaltet. Die Ergebnissätze des entsprechenden Cursors werden mit einzelnen FETCH-Aufträgen bearbeitet.
    Stellt der Prefetch-Puffer keinen Engpass dar, kann der Blockungsfaktor großzügig gewählt werden.
  - Unabhängig vom angegebenen Blockungsfaktor können in einem Schub nur so viele Ergebnissätze transportiert werden, wie in einen Nachrichtenpuffer passen. Hat der Blockungsfaktor diese Größe bereits erreicht, kann die Performance mit einer Erhöhung des Blockungsfaktors nicht verbessert werden.
  - Über den im Pragma PREFETCH vereinbarten Blockungsfaktor sollten nicht mehr Ergebnissätze angefordert werden, als im Anwendungsprogramm verarbeitet werden.

Vergleich von Zeilen („lexikografischer Vergleich“)

In bestimmten Fällen ist es erforderlich, zeilenwertige Operanden zu vergleichen, z.B.
`(NAME, VORNAME) >= ('Meier', 'Otto')`

Es werden im Beispiel also Treffersätze gesucht, für die das Tupel `(NAME, VORMAME)` lexikografisch größer oder gleich dem Tupel `('Meier', 'Otto')` ist. Das heißt:

1. die Spalte `NAME` hat einen Wert `>'Meier'` oder
2. die Spalte `NAME` hat den Wert `='Meier'` hat und gleichzeitig hat die Spalte `VORNAME` einen Wert `>='Otto'`

Dafür kann folgendes Prädikat benutzt werden:
```
NAME > 'Meier' OR (NAME = 'Meier' AND VORNAME >= 'Otto')
```

Folgende Formulierung ist aber einfacher und übersichtlicher:
```
(NAME, VORNAME) >= ('Meier', 'Otto')
```
In bestimmten Fällen hat diese Schreibweise auch Performance-Vorteile:
– es entfallen explizite Verknüpfungen mit `OR`
– die Nutzung geeigneter Indizes (auch Primärschlüssel) ist einfacher

Eine SQL-INSERT-Anweisung für mehrere Sätze

Auch beim Einfügen von Sätzen kann eine Performance-Verbesserung erzielt werden. In einer SQL-INSERT-Anweisung können mehrere Sätze angegeben werden (siehe Handbuch „SQL-Sprachbeschreibung Teil 1: SQL-Anweisungen“). Wie beim Lesen im Schubmodus können damit Kommunikationsschritte zwischen Anwendungsprogramm und DBH eingespart werden. Die mögliche Performance-Verbesserung ist auch hier im Wesentlichen davon abhängig, wie teuer die eingesparten Kommunikationsschritte sind.

Einsatz von Routinen

Der Einsatz von Routinen kann zu Performance-Verbesserungen führen, siehe Abschnitt „Einsatz von Routinen“ auf Seite 55.

6.1.2 Tuning von SQL-Anweisungen

Dieser Abschnitt gibt gezielte Hinweise, wie die Performance und damit die Antwortzeit einer einzelnen SQL-DML-Anweisung verbessert werden kann. Eine einzige ungünstig formulierte SQL-Anweisung kann möglicherweise die Performance der gesamten DBH-Session stark beeinträchtigen. Daher sollte man sein Augenmerk zunächst auf die Optimierung von SQL-Anweisungen richten, bevor man versucht, die Performance des DBH (z.B. durch Änderung von DBH-Optionen) zu verbessern.

6.1.2.1 Allgemeine Vorgehensweise

Bevor mit dem Tuning begonnen werden kann, muss man herausfinden, welche SQL-Anweisung die Performance-Probleme verursacht. Hierzu sind folgende Werkzeuge zur Performance-Analyse nützlich:

- View SYS_DML_RESOURCES:
  Der View SYS_DML_RESOURCES des SYS_INFO_SCHEMA liefert Informationen über besonders „teure“ SQL-DML-Anweisungen, siehe
  Abschnitt „View SYS_DML_RESOURCES“ auf Seite 21 und das Handbuch „SQL-Sprachbeschreibung Teil 1: SQL-Anweisungen“.
- Maske „SQL INFORMATION“ von SESMON:
  Liefert eine globale Übersicht über die Häufigkeit von SQL-DML-Anweisungen, Plan-Generierungen und Aufrufe der SQL-Komponenten.
- EXPLAIN:
  Mit der Erklärungskomponente erhält man eine aufbereitete Darstellung des Zugriffsplans für eine SQL-DML-Anweisung. Man erhält Informationen, in welchen Einzelschritten die SQL-Anweisung abgearbeitet wird und welche Zugriffspfade dabei genutzt werden können. Die Erklärungskomponente kann wichtige Hinweise geben, wo bei der Ausführung der SQL-Anweisung Performance-Probleme auftreten können. Daher empfiehlt es sich, die Zugriffspläne einer neuen SQL- Anwendung damit zu kontrollieren.
- SESCOS:
  Über einen gewissen Zeitraum können Transaktionen, SQL Anweisungen und einzelne Verarbeitungsschritte mitprotokolliert und anschließend mit SESCOSP ausgewertet und analysiert werden.

Diese Werkzeuge zur Performance-Analyse sind im Kapitel „Werkzeuge zur Performance-Analyse“ auf Seite 13 ausführlich beschrieben. Erläuterungen zur Optimierung von SQL-Anweisungen finden Sie im Abschnitt „Aufbau und Steuerung des Optimizers“ auf Seite 24. Die aufbereitete Darstellung eines Zugriffsplans, wie sie von der Erklärungskomponente erzeugt wird, ist im Kapitel „Erklärungskomponente des Optimizers“ auf Seite 131 beschrieben.

Sobald man eine performance-kritische SQL-Anweisung eingekreist hat, liefert die Erklärungskomponente die notwendigen Informationen über den vom Optimizer gewählten Zugriffsplan. Dieser Zugriffsplan ist die Ausgangsbasis für die weiteren Überlegungen, wie der Plan gezielt verbessert werden kann. Dazu muss man selbst eine Vorstellung entwickeln, wie der optimale Plan für diese SQL-Anweisung aussehen könnte. Folgende Maßnahmen kommen für die Verbesserung des Zugriffplans infrage:

- Einrichten von Indizes
- Steuern des Optimizers mit den Pragmas OPTIMIZATION LEVEL, SIMPLIFICATION, JOIN, KEEP JOIN ORDER und IGNORE INDEX
- Umformulieren der SQL-Anweisung bzw. Umstrukturieren der SQL- Anwendung.

Den optimalen bzw. einen nahezu optimalen Zugriffplan kann man durch gezielten Einsatz dieser Maßnahmen erreichen.

6.1.2.2 Typische Performance-Probleme

Im Folgenden werden einige typische Fälle betrachtet, die bei der Abarbeitung von SQL-Anweisungen potenziell Performance-Probleme verursachen können. Es wird jeweils beschrieben, wie man anhand der Explain-Darstellung erkennen kann, dass das genannte Problem vorliegt, und welche Abhilfemaßnahmen möglich sind.

Problem

Die Sätze, die die Suchbedingung erfüllen, werden durch sequenzielles Lesen der gesamten Basistabelle ermittelt.

Explain:

Beim Knoten „table_scan“ fehlt das Attribut „used index“ ganz, oder es ist zwar das Attribut „used index: <Primary Key>“ vorhanden, aber die Suchbedingung auf den Spalten des Primärschlüssels selektiert zu schwach. Ein Indiz dafür ist das Attribut „cost_ratio: n / m“. Prüfen, falls der Wert n sehr hoch ist (ungefähr gleich der Anzahl der Blöcke der Basistabelle).

Abhilfe:

Einrichten eines oder mehrerer Indizes auf den Spalten der Suchbedingung. Insbesondere bei Suchbedingungen der Form `col1 = wert1 AND col2 = wert2 AND ...` sollte ein zusammengesetzter Index auf den Spalten (col1, col2, ...) in Betracht gezogen werden.

Problem

Für die Auswertung einer Suchbedingung auf einer Basistabelle wird ein Index mit geringer Selektivität verwendet.

Explain:

Der betreffende Index ist unter „used index:“ aufgeführt. Der Wert m im Attribut „cost_ratio: n / m“ ist von derselben Größenordnung wie die Anzahl der Sätze in der Basistabelle.

Abhilfe:

Mit dem Pragma IGNORE INDEX kann der betreffende Index von der Auswertung ausgeschlossen werden. Unter Umständen kann der Index mit DROP INDEX ganz gelöscht werden.

Eventuell ist auch nur die Statistik-Information für diesen Index veraltet. Abhilfe schafft dann ein REORG STATISTICS auf diesem Index.

Problem

Für die Auswertung der SQL-Anweisung müssen viele Sätze sortiert werden.

Explain:

Der Zugriffsplan kann verschiedene Inhalte haben:

a) Einen „sort“-Knoten

b) Der Knoten „table_scan“ enthält das Attribut „sortorder:“, ohne dass ein „sort index:“ verwendet werden kann.

Abhilfe:

Im Fall (a) muss man unterscheiden, ob der „sort“- Knoten wegen einer ORDER BY-Klausel oder einem Sort-Merge-Join notwendig ist. Eventuell ist es günstiger, den Sort-Merge-Join durch einen Nested-Loop-Join zu ersetzen (siehe unten), so dass die Sortierung entfallen kann. Wenn der „sort“- Knoten auf Grund einer ORDER BY-Klausel in der SQL- Anweisung entstanden ist, sollte man diese Klausel - falls von der Anwendungslogik nicht unbedingt erforderlich - weglassen.
Im Fall (b) sollte ein Index eingerichtet werden, der mindestens die unter „sortorder:“ aufgeführten Spalten umfasst.

Problem

Der Optimizer wählt den Algorithmus für die Berechnung einer Join-Ergebnismenge so, dass die Bestimmung der gesamten (!) Ergebnismenge möglichst performant wird. Falls man aber gar nicht an der gesamten Treffermenge der Joins interessiert ist, sondern nur an den ersten n Sätzen oder falls man sogar nur den ersten Satz (den dafür aber möglichst schnell) haben will, so ist der vom Optimizer erstellte Plan für dieses Ziel evtl. nicht optimal. Wenn z.B. die gesamte Join-Ergebnismenge aus sehr vielen Sätzen besteht und beim Nested-Loop-Join der Zugriff auf die innere Relation nicht durch geeignete Indizes unterstützt wird, so ist es aus Optimizersicht besser, die beiden Joinrelationen zu sortieren und das Ergebnis durch einen Sort-Merge-Join zu ermitteln. Für eine Anwendung, die aus der großen Treffermenge aber nur an einem Satz interessiert ist, bräuchte man bei der Auswertung zum ersten passenden Satz der äußeren Tebelle nur einen einzigen Partner in der inneren Tabelle zu suchen. Diese Suche wäre durch den Nested-Loop-Join wesentlich performanter zu erledigen.

Explain:

Sort-Merge-Join im Explain und für das gesamte Joinergebnis wird eine große Treffermenge erwartet.

Abhilfe:

Erzwingen des Nested-Loop-Join mit dem Pragma JOIN NESTED LOOP.

Problem

Ein Nested-Loop-Join ist sehr performant, wenn auf dem linken (inneren) Operanden ein passender Index für die Join-Bedingung existiert und die Join-Bedingung sehr selektiv ist; das trifft insbesondere zu, wenn sich die Join-Bedingung auf den Primärschlüssel der inneren Basistabelle bezieht. Ist die Join-Bedingung allerdings weniger selektiv, so ist ein Nested-Loop-Join deutlich schlechter als ein Sort-Merge-Join.

Explain:

In der Explain-Darstellung muss der Knoten „table_scan“ betrachtet werden, der unter „left:“ beim Knoten „nljoin“ angegeben ist. Es gibt zwei Möglichkeiten:

a) Unter „used index:“ existiert kein geeigneter Index für die Join-Bedingung

b) Bei dem Attribut „cost_ratio: n / m“ ist der Wert m um Größenordnungen kleiner als die tatsächliche Zahl von Sätzen, die die Join-Bedingung erfüllen.

Abhilfe:

Im Fall (a) kann ein passender Index definiert werden.
Im Fall (b) sollte zunächst mit REORG STATISTICS die Statistik-Information aktualisiert werden.

In beiden Fällen kann mit dem Pragma JOIN SORT MERGE ein Sort-Merge-Join erzwungen werden. Oft reicht es auch schon aus, den für die Auswertung der Join-Bedingung verwendeten Index (außer Primärschlüssel-Index) mit dem Pragma IGNORE INDEX abzuschalten.

Problem

Sort-Merge-Joins sind im Allgemeinen performanter als Nested-Loop-Joins. Nur wenn der eine Join-Operand aus sehr wenigen Sätzen und der andere Join-Operand aus sehr vielen Sätzen besteht, ist der Nested-Loop-Join unter gewissen Voraussetzungen günstiger (siehe oben).

Explain:

Für diese Situation liefert die Explain-Darstellung keine weiteren Hilfen. Man muss anhand der Größe der jeweiligen Join-Operanden beurteilen, ob ein Nested-Loop-Join günstiger ist.

Abhilfe:

Es sollte ein geeigneter Index auf der größeren Tabelle für die Auswertung der Join-Bedingung definiert sein. Außerdem muss die Statistik-Information aktuell sein. Sollte trotzdem nicht der gewünschte Nested-Loop-Join entstehen, so kann man ihn mit dem Pragma JOIN NESTED LOOP erzwingen.

Beispiel

```
SELECT ....              SELECT ....
FROM R,S           ⇒     FROM R
WHERE R.a = S.a          WHERE R.a IN (SELECT S.a FROM S)
```

Hinweis: Nach dieser Umformung enthält die Explain-Darstellung keinen „nljoin“-Knoten mehr. Die geänderte Abarbeitungsstrategie entspricht jedoch, auch bezüglich Antwortzeit, einem Nested-Loop-Join.

Problem

Statt eines Joins wird das kartesische Produkt gebildet.

Explain:

Die Explain-Darstellung enthält den Knoten „cross“. Es gibt allerdings eine Situation, in der das kartesische Produkt performanter als ein Join ist:

```
SELECT ...
FROM R, S
WHERE R.a=wert AND R.a=S.a
```

Hier wird vom Optimizer eine zusätzliche Bedingung S.a=wert generiert, und dafür die Join-Bedingung R.a=S.a eliminiert. Im Plan enthält der Knoten „table_scan“ der Basistabelle S die zusätzliche Bedingung und ist über einen „cross“-Knoten mit dem „table_scan“ der Basistabelle R verbunden.

Abhilfe:

Man sollte kontrollieren, ob alle Tabellen in der FROM-Klausel mit mindestens einer Join-Bedingung der Form Tabelle1.Spalte1 = Tabelle2. Spalte2 miteinander verknüpft sind. Andere Join-Bedingungen sollten vermieden werden, z.B. die Folgenden (führt häufig zum Knoten „nljoin“):

```
R.a > S.a
R.a = S.a * 1.15
R.a + S.a = 10
```

Problem

Wahl einer ungünstige Join-Reihenfolge bei einem Join von mehreren Tabellen.

Explain:

Anhand der Explain-Darstellung sieht man deutlich, in welcher Reihenfolge und mit welchen Join-Methoden die Tabellen verknüpft werden.

Abhilfe:

1. Man überlegt sich, welche Join-Reihenfolge - auf Grund der Größe der Tabellen und der nach jedem paarweisen Join anfallenden Zwischenergebnisse - die günstigste ist. Diese Join-Reihenfolge kann dem Optimizer vorgegeben werden, wenn mit dem Pragma SIMPLIFICATION OFF die Normalisierung von SQL-Anweisungen abgeschaltet wird und in der SQL-Anweisung explizite Join-Ausdrücke verwendet werden oder wenn mit dem Pragma KEEP KOIN ORDER die Join-Reihenfolge direkt angegeben wird.

   Beispiel

   ```
   SELECT ...
   FROM R, S, T , U
   WHERE R.a=S.a AND S.b=T.b AND T.c=U.c AND ...
   ```

   ⇒

   ```
   --%PRAGMA SIMPLIFICATION OFF
   SELECT ...
   FROM ( (R JOIN S ON R.a=S.a) JOIN
   (T JOIN U ON T.c=U.c) ON S.b=T.b )
   WHERE ...
   ```

Bei der zweiten, umformulierten SQL-Anweisung wird zunächst R mit S und T mit U verknüpft, anschließend werden die jeweiligen Zwischenergebnisse verknüpft.

2. Bei Verwendung des Pragmas OPTIMIZATION LEVEL n (n < 9) wählt der Optimizer eine Join-Reihenfolge aus, die durch Umstellen der Join-Bedingungen in der WHERE-Klausel beeinflusst werden kann.

Problem

Die Prüfung von Referenzbedingungen beeinträchtigt die Antwortzeit.

Explain:

Bei den Knoten „insert_stmt“, „delete_stmt“ bzw. „update_stmt“ enthält der Teilbaum unter „check_after_all“ oder „check_on_the_fly“ einen Quantor („some“-, „all“-Knoten) und dieser wieder einen „table_scan“ auf die referenzierte bzw. referenzierende Basistabelle.
Die Antwortzeit kann insbesondere dann ungünstig sein, wenn die Referenzbedingung unter „check_after_all“ abgeprüft werden muss. Das ist dann der Fall, wenn es sich um eine selbst-referenzierende Integritätsbedingung (d.h. die referenzierende Tabelle ist gleichzeitig die referenzierte Tabelle) handelt, oder wenn eine UPDATE- bzw. DELETE-Anweisung sich auf die referenzierte Tabelle bezieht.

Abhilfe:

Bei einer DELETE-Anweisung kann die Prüfung beschleunigt werden, wenn auf den referenzierenden Spalten ein Index definiert wird
(siehe Abschnitt „Integritätsbedingungen“ auf Seite 62).

Problem

Beim Auftreten von nicht korrelierten Unterabfragen in Prädikaten kann es zu Performance-Problemen kommen.

Explain:

Die Tabelle, für die das Prädikat ausgewertet wird, wird sequenziell gelesen und das Prädikat wird als „some“-Prädikat eines „rest“-Knotens dargestellt.

Abhilfe:

Evtl. Anfrage als Join umformulieren.

Beispiel

```
SELECT *
FROM   R
WHERE  R.key = ANY  ( SELECT S.b
                      FROM   S
                      WHERE  S.c = :var )

oder auch

SELECT *
FROM   R
WHERE  EXISTS ( SELECT S.b
                FROM   S
                WHERE  S.c = :var AND S.b = R.key )

⇒

SELECT R.*
FROM   R, (SELECT DISTINCT S.b
           FROM   S
           WHERE  S.c = :var ) AS T(x)
WHERE  R.key = T.x
```

Während in den ersten beiden Formulierungen die Tabelle R sequenziell gelesen werden muss, wird die Bedingung R.key = T.x über einem auf R.key definiertem Index ausgewertet.

Problem

Bei EXISTS- / NOT EXISTS-Prädikaten können bei häufiger Auswertung der Unterabfrage Performance-Probleme auftreten.

Explain:

Dieses Prädikat wird i.a. über einen „some“- oder „all“-Knoten im Explain dargestellt. Allerdings geht aus dem Explain nicht hervor, wie oft eine Unterabfrage ausgewertet werden muss. Daher kann i.a. nicht erkannt werden, ob Performance-Probleme auftreten.

Abhilfe:

Wenn die Bedingungen in der Unterabfrage für Join-Auswertung geeignet sind (Equi-Join-Prädikate; siehe Abschnitt „Phase 3: Zugriffspfadauswahl“ auf Seite 30), kann bei einem NOT EXISTS-Prädikat u. U. eine Umformulierung in einen Outer Join günstiger sein.

Beispiel

```
SELECT ...
FROM   R
WHERE  NOT EXISTS ( SELECT ...
                    FROM   S
                    WHERE  S.b = R.key )
```

⇒

```
SELECT ...
FROM   (R LEFT OUTER JOIN S on S.b = R.key)
WHERE  S.key IS NULL
```

Hier muss die Unterabfrage jedes Mal neu ausgewertet werden, da die Spalte R.key als Schlüssel eindeutig ist; in diesem Fall ist die Outer Join-Variante günstiger, wenn eine Sortierung von S bezüglich Spalte b (z.B. über Index) gegeben ist. Dann kann nämlich ein performanter „left_outer_mjoin“ geplant werden.

Problem

Mengenfunktionen (AVG,MAX,MIN,SUM,COUNT(*),COUNT) für alle Sätze einer Basistabelle werden durch sequenzielles Lesen aller Sätze einer Basistabelle ermittelt, z.B.

- `SELECT MIN(column) FROM T`
- `SELECT MAX(column) FROM T`
- `SELECT COUNT(*) FROM T`

Explain:

Es tritt ein Knoten „sorted_group“ bei der Bestimmung der Mengenfunktionen auf.

Abhilfe:

Die Spalte darf nicht vom Datentyp VARCHAR sein.
Für das Mengenfunktionsargument ist ein Index einzurichten.
Für die Bestimmung von MIN(column) bzw. MAX(column) sollte column die erste (und evtl. einzige) Komponente des Index sein. Für die Bestimmung von COUNT(*) ist ein (beliebiger) Index für eine Spalte mit der NOT NULL-Eigenschaft einzurichten.
Die Spalte muss jeweils in ihrer ganzen Länge in den Index einbezogen sein.

Problem

Mengenfunktionen für eine selektierte Menge von Sätzen werden durch sequenzielles Lesen aller Sätze einer Basistabelle ermittelt, z.B.

```
SELECT MIN(col1),MAX(col2),AVG(col3),SUM(col4),COUNT(*),...
   FROM T
   WHERE   search-condition
```

Für jeden Satz wird geprüft, ob die Suchbedingung erfüllt ist.

Explain:

Es tritt ein Knoten „sorted_group“ bei der Bestimmung der Mengenfunktionen auf.

Abhilfe:

- Für die Mengenfunktionsargumente müssen Indizes eingerichtet sein.
  Die Mengenfunktionsargumente dürfen nicht vom Datentyp VARCHAR sein und müssen jeweils in ihrer ganzen Länge in den Index einbezogen sein.
- Die Suchbedingung muss über Indizes ausgewertet werden.

  Zu diesem Zweck ist für jede Spalte der Suchbedingung ein Index einzurichten. Die Spalten müssen jeweils in ihrer ganzen Länge in den Index einbezogen sein.

  Als Prädikate der Suchbedingung dürfen nur folgende Prädikate auftreten, die mit AND bzw. OR verknüpft sind:
  – Vergleich von zwei Werten der Form „column vergleichs_op value“
  – Bereichsabfragen „column BETWEEN value_1 AND value_2“
  – Mustervergleich „column LIKE value“
  – Elementabfrage „column IN (value_1,...,value_n)“

  Dabei ist column keine multiple Spalte, vergleichs_op ist ein Vergleichsoperator. value ist ein Literal, eine Benutzervariable oder ein zuvor berechenbarer Ausdruck.

6.2 Optimierungsmöglichkeiten bei CALL-DML

In diesem Abschnitt werden Optimierungsmöglichkeiten beschrieben, die für CALL-DML-Anwendungen empfehlenswert sind:

- Einsparen von Kommunikationspfadlänge
- Einsparen von Pfadlänge bei der Syntaxanalyse
- Beachten allgemeiner Programmier-Regeln
- Beschleunigen der Join-Verarbeitung
- Beeinflussen der Abarbeitungsreihenfolge bei der Suchfrage

6.2.1 Einsparen von Kommunikationspfadlänge

In der CALL-DML gibt es folgende Möglichkeiten, Kommunikationspfadlängen einzusparen:

- Kettung von CALL-DML-Aufträgen. Folgende Kettungen sind erlaubt:
  - Begin-of-Transaction mit einer UPDATE- oder Wiedergewinnungsanweisung
  - n Open-Anweisungen
  - n Open-Anweisungen mit Begin-of-Transaction vorgekettet
  - End-of-Transaction oder Rollback-of-Transaction gekettet mit Begin-of-Transaction (nicht in UTM-Umgebung zulässig)
  - End-of-Transaction oder Rollback-of-Transaction gekettet mit einer close-Anweisung
- Ausnutzung des Schubmodus

6.2.2 Einsparen von Pfadlänge bei der Syntaxanalyse

In der CALL-DML gibt es folgende Möglichkeiten, Pfadlängen bei der Syntaxanalyse einzusparen:

- Ausnutzen des Schubmodus
- Ausnutzen von Folgeanweisungen
- Angabe von Bereichen bei multiplen Attributen statt Ausgabe von allen einzelnen Ausprägungen

- Ausnutzen von implizitem Schließen der logischen Dateien bei End-of-Transaction für alle innerhalb dieser Transaktion eröffneten logischen Dateien
- Sollen alle logischen Dateien eines Benutzers geschlossen werden, so ist der User-Close billiger als mehrere Datei-Closes.

6.2.3 Allgemeine Programmierempfehlungen

Bei der Programmierung von CALL-DML-Anwendungen sollten Sie die folgenden Empfehlungen beachten:

- Anzahl Attribute in der Projektion auf die notwendigen beschränken. Jedes unnötig ausgegebene Attribut kostet Pfadlänge bei der Konvertierung vom internen Format ins externe, bei der Übertragung und bei der Syntaxanalyse.
- Anzahl Attribute bei der UPDATE-Anweisung auf die wirklich geänderten beschränken. Jedes Attribut, das nur einen Pseudo-Update darstellt (z.B. Farbe von rot nach rot), erzeugt unnötige Pfadlänge bei der Syntaxanalyse, bei der Abarbeitung (das Attribut wird wirklich ausgetauscht) und bei der Protokollierung für Transaktionssicherung und Media Recovery.
- Zusammenfassen von mehreren gleichen Suchbedingungen zu einem SAN mit einer Vergleichsbedingung in einer Oder-Gruppe zu einer Suchbedingung mit mehreren Vergleichsbedingungen.

  Beispiel

  ```
  col1=wert1 v col1=wert2 v col1=wert3 v... v col1=wertn (san5010san501...)
  ⇒
  col1=wert1 v =wert2 v =wert3 v... v =wertn (san50101...01)
  ```

  Da SESAM/SQL bei der CALL-DML keine Optimierung durchführt, die die Suchbedingung übergreift, kann durch die äquivalente Umformulierung der Suchfrage erhebliche Pfadlänge eingespart werden. Dies ist insbesondere dann der Fall, wenn ein Sekundärindex für dieses Attribut vorhanden ist. (Der Sekundärindex wird pro Suchbedingung ausgewertet).
- Vermeiden von Frontmaskierung bzw. von Nichtsignifikanzbedingungen. Teilfragen, die Werte mit Frontmaskierung enthalten, sind in der Regel nicht performant über Sekundärindex auswertbar. Die Nichtsignifikanzbedingung kann nicht über den Index ausgewertet werden, da bei SESAM/SQL ein nichtsignifikanter Wert durch das Nicht-Vorhandensein des Wertes dargestellt wird.

6.2.4 Beschleunigen der Join-Verarbeitung für CALL-DML

Die Join-Verarbeitung für CALL-DML kann dadurch beschleunigt werden, dass man zu einem geeigneten Zeitpunkt und mit einer dafür geeigneten Treffermenge von Nested-Loop-Join auf Merge-Join umschaltet.
Der wesentliche Faktor, der diese Umschaltung bewirkt, ist die Anzahl der Treffer, die die zwei Join-Komponenten liefern. Dabei werden folgende Fälle unterschieden:

- eine oder beide Komponenten liefern durch die angegebene Primärschlüssel-Funktion (4/8) höchstens einen Treffer
- eine der Komponenten liefert durch die vorausgehende Index-Auswertung eine Anzahl Treffer, die den Optimierungswert nicht überschreitet. Der Optimierungswert kann über einen optionalen Rep gesteuert werden (siehe die Freigabemitteilung von SESAM/SQL). Der Standardwert ist 10.

6.2.5 Abarbeitungsreihenfolge der Suchfrage für CALL-DML

Bei einer sequenziellen Suche (keine Index-Auswertung gewünscht oder keine Index-Auswertung möglich) wird üblicherweise vorausgesetzt, dass die Teilfragen in der von der Anweisung bestimmten Reihenfolge abgearbeitet werden.

Der folgende Abschnitt erklärt, warum dies nicht der Fall ist, und wie die Abarbeitungsreihenfolge durch die Anweisung bzw. durch die Reihenfolge der einzelnen Teilfragen in der Anweisung beeinflusst werden kann.

Die einschränkenden Teilfragen (Bedingungen) einer Suchfrage werden intern in einer Postfix-Notation (Polish-List) dargestellt. Diese Postfix-Darstellung wird während der Analyse der Anweisung von links nach rechts erstellt. Da es sich aber um eine Postfix-Notation handelt, wird sie von rechts nach links abgearbeitet. Die Anweisung wird also in umgekehrter Reihenfolge abgearbeitet.

Um die sequenzielle Suche zu beschleunigen, wird die Selektivität der einzelnen Teilfragen genutzt, umso eine optimale Abarbeitung zu erzwingen. Für die einzelnen Teilfragen wird ein so genanntes statisches Gewicht vergeben, welches das Sortierkriterium innerhalb der gleichen Schachtelungstiefe darstellt.

Teilfragen mit gleicher Selektivität werden also nicht umsortiert, und diese werden nicht in der angegebenen Reihenfolge abgearbeitet.

Fazit: Innerhalb der gleichen Schachtelungstiefe sollten die selektivsten Teilfragen ganz rechts innerhalb dieser Schachtelungstiefe angegeben werden.

7 Erklärungskomponente des Optimizers

Die Erklärungskomponente des Optimizers soll dem Anwender Aufschluss geben über die vom Optimizer gewählte Abarbeitung seiner Anfrage. Sie bezieht sich immer auf die Realisierung einer Datenbankanfrage, konkret also einer SQL-DML-Anweisung. Nur für SQL-DML-Anweisungen ist das Pragma EXPLAIN (siehe Abschnitt „SQL-Optimizer“ auf Seite 17) definiert.

Im folgenden Abschnitt „Einführung“ wird zunächst die Erklärungskomponente an einem Beispiel vorgestellt.

In den weiteren Abschnitten werden Arbeitsweise und Darstellung der Erklärungskomponente für die einzelnen SQL-Elemente vorgestellt:

- Tabellenwertige Operationen (RELATION) auf Seite 140
- DML-Anweisungen auf Seite 172
- Quantoren auf Seite 192
- die Wertabfrage auf Seite 194
- Ausdrücke, Funktionsaufrufe und Prädikate auf Seite 195
- Zwischendateien auf Seite 203

Änderungen bleiben ausdrücklich vorbehalten:
Wie in den übrigen Teilen des Handbuchs bezieht sich der Inhalt der folgenden Abschnitte auf den aktuellen Stand der Implementierung.
Auf die Reihenfolge der Attribute innerhalb eines Knotens besteht kein Einfluss. Die hier gezeigten Beispiele können in diesem Punkt von den Ausgaben des Produktes abweichen.
Bei einer Weiterentwicklung des Optimizers können Knoten oder Attribute hinzukommen, wegfallen oder ihre Bedeutung ändern.
Schließlich kann sich auch das äußere Bild der Erklärung verändern oder die Erklärungskomponente durch ein verbessertes Werkzeug ersetzt werden.

7.1 Einführung

Die Ausgabe der Erklärungskomponente sieht auf den ersten Blick einer SQL-Anweisung überhaupt nicht ähnlich, denn sie leitet sich direkt vom Operatorbaum ab, der dem Interpreter zur Abarbeitung übergeben wird. Das macht die Erklärung zunächst etwas unübersichtlich, hat aber den Vorteil, dass der tatsächliche Abarbeitungsaufwand sehr genau eingeschätzt werden kann.
Wie Sie sehen werden, gibt es für einige der Operatoren auch keine entsprechende Darstellung in SQL, weil es bei ihnen beispielsweise eine Rolle spielt, in welcher Reihenfolge die Operanden ausgewertet werden.
Diese Einführung soll zeigen, wie die Erklärung zu lesen ist und wie man die Wirkung von performance-beeinflussenden Maßnahmen ablesen kann.

7.1.1 Verwendete Begriffe

Bei den Ausführungen zur Erklärungskomponente schimmert gelegentlich das mathematische Modell durch - der attributierte Graph. Dann werden die Operationen (relationale oder auch arithmetische) als Knoten bezeichnet. Ihre Operanden und Optionen werden unter der Bezeichnung Attribute zusammengefasst. Einige Attribute stellen eine veränderliche Zahl von Operanden dar, sie werden als Sequenzen bezeichnet.

Ein Knoten (Node) wird folgendermaßen dargestellt:

```
nodename [
   attributename1: attibutvalue1,
   attributename2: <
      attributvalue2.1,
      ...
   >
   ...
]
```

Wenn Sie gelegentlich Attribute vermissen, die eigentlich zu einem Knoten gehören, so hat die Erklärungskomponente sie weggelassen, weil sie nicht besetzt waren.
Einer Erklärung wird der zugehörige Quelltext vorangestellt. (Er enthält am Anfang zusätzliche Pragmas, die während der Aufbereitung der Anweisung generiert wurden.)

7.1.2 Ausgangsbeispiel

Gegeben seien zwei Tabellen:

- Die Tabelle DEPT enthält die Spalten DEPT_NO (Primärschlüssel), DEPT_NAME und CITY. Die Eindeutigkeitsbedingung DEPT bezieht sich auf die Spalten DEPT_NAME und CITY.
- Die Tabelle EMP enthält die Spalten EMP_NO (Primärschlüssel), DEPT, EMP_NAME und SAL. Für die Spalte DEPT ist die Referenzbedingung F1_DEPT definiert, die sich auf den Primärschlüssel DEPT.DEPT_NO bezieht.

Die Tabellen können durch folgende DDL-Anweisungen erzeugt werden:

```
CREATE TABLE dept
                (dept_no    CHAR(5),
                 dept_name  CHAR(30) NOT NULL,
                 city       CHAR(30) NOT NULL,
                 CONSTRAINT dept_no PRIMARY KEY (dept_no),
                 CONSTRAINT dept UNIQUE (dept_name, city) )
CREATE TABLE emp
                (emp_no     CHAR(5),
                 dept       CHAR(5),
                 emp_name   CHAR(30),
                 sal        DEC(9,0),
                 CONSTRAINT emp_no PRIMARY KEY (emp_no),
                 CONSTRAINT f1_dept FOREIGN K (dept)
                    REFERENCES dept (dept_no) )
```

Wir betrachten die folgende DML-Anweisung, die den Durchschnittslohn jeder Abteilung berechnet.

```
--%PRAGMA EXPLAIN INTO 'tm.expl'
--%PRAGMA SIMPLIFICATION ON

SELECT dept_name, AVG(sal)
FROM  emp, dept
WHERE emp.dept = dept.dept_no
GROUP BY dept_name
```

Die Erklärung wird ausgegeben, wenn die Anweisung das Pragma EXPLAIN enthält.
Da die Anweisung ziemlich einfach ist, wurde auch das Pragma SIMPLIFICATION ON angegeben. Generell ist das nicht nötig, denn bei einfachen Anweisungen bringt die Simplifizierung relativ wenig, bei komplizierteren Abfragen wird sie automatisch angestoßen.
(Sie wird hier im Beispiel jedoch angefordert, weil einige überflüssige eproj-Operationen verschwinden und die Erklärung damit kürzer wird.)

Man könnte das Ergebnis so bestimmen, dass

- zunächst das Kreuzprodukt aus EMP und DEPT gebildet wird,
- dann alle Paare verworfen werden, in denen EMP.DEPT und DEPT.DEPT_NO verschieden sind,
- die verbleibenden nach DEPT_NAME sortiert werden,
- und über allen Zeilen mit gleichem DEPT_NAME der Durchschnitt über SAL gebildet wird.

Das ist bestimmt nicht die beste Methode, und Sie dürfen vom SESAM/SQL-Optimizer mehr erwarten. Welches Vorgehen er bestimmt, sehen Sie mit Hilfe der Erklärungskomponente.

Die folgenden Beispiele wurden mit einer schwach gefüllten Testdatenbank erzeugt. Da die Optimierung auch von der geschätzten Trefferzahl abhängt, kann dieselbe Anweisung auch anders realisiert werden.

Der Abarbeitungsplan des Beispiels

Für die genannte Anweisung ergibt sich die folgende Ausgabe der Erklärungskomponente. Zur besseren Orientierung wurden hier rechtsbündig stichwortartige Erläuterungen in kursiver Schrift eingefügt.

```
dynamic_select_stmt  [              Anweisung als dyn. SELECT-Anweisung optimiert
  output: <
     DEPT_NAME,                                          Spalte der Ergebnistabelle
     (column #2)                                         Spalte der Ergebnistabelle
  >                                  Anfang der Beschreibung der Ergebnistabelle
  query:  sorted_group  [                                Gruppierungsoperation
     from:  nljoin  [                                    basiert auf Nested-Loop-Join
        left:  table_scan  [                             linker Join-Operand: Table-Scan
           outer_reference_s: <                          nur Zeilen von EMP
              (expr #1) ::= DEPT_NO                      für dept=dept.dept_no und
           >                                             aktuellen Wert von dept.dept_no
           base: ROCAT.DDLOPT.EMP                        zu Grunde liegende Basistabelle
           column_s: <                                   emp  mit den Spalten
              SAL,                                       sal und
              DEPT                                       dept
           >
           isolation level: set at runtime               Isolationslevel
           lock mode: shared
           cost_ratio: 6.000000E+00/4.000000E+00
           pred:                                 Spaltenwerte müssen (emp.dept=dept.dept_no)
                 ((DEPT = (expr #1)))                    erfüllen
        ]
        right:  table_scan  [                            rechter Join-Operand: Table-Scan
           base: ROCAT.DDLOPT.DEPT                       zu Grunde liegende
           column_s: <                                   Basistabelle mit den Spalten
              DEPT_NAME #3,                              dept_name und
              DEPT_NO                                    dept_no
           >
           sortorder: <                                  nach DEPT_NAME
              DEPT_NAME #3 ascending                     aufsteigend geordnet lesen
           >                                             und den Index für die
           sort_index: UI9941102110408000                Eindeutigkeitsbedingung  verwenden
           isolation level: set at runtime
           lock mode: shared
           cost_ratio: 0.000000E+00/0.000000E+00
        ]
     ]
     group_col_s: <
        DEPT_NAME ::= DEPT_NAME #3                       Gruppierungsspalte
     >
     aggregate_s: <                                      Für jede Gruppe
        (column #2) ::= all_avg(SAL)                     Durchschnitt AVG() berechnen
     >
  ]                                  Ende der Beschreibung der Ergebnistabelle
  is_for_update: False
  read_only: True
]
```

Die Erklärung ist ziemlich umfangreich, da sie alle Informationen über die geplante Abarbeitung der Anweisung enthält. Sie ist folgendermaßen zu lesen:

Die Anweisung wurde als dynamische SELECT-Anweisung (dynamic_select_stmt) optimiert, also mit dem Default-Wert 9 für das Pragma OPTIMIZATION LEVEL.

Das Ergebnis der Anweisung finden wir ab der Zeile

```
query: sorted_group [
```

Von hier bis zur korrespondierenden schließenden Klammer ist die Ergebnistabelle beschrieben. Es ist eine Operation, bei der alle Zeilen mit gleichen Werten in folgenden Gruppierungsspalten zusammengefasst werden:

```
group_col_s: <
   DEPT_NAME ::= DEPT_NAME #3
>
```

Zusätzlich werden für jede Gruppe die entsprechenden Mengenfunktionen berechnet:

```
aggregate_s: <
   (column #2) ::= all_avg(SAL)
>
```

Das Argument der Gruppierungsoperation ist eine Tabelle, die von einer so genannten Nested-Loop-Join-Operation (nljoin) erzeugt wird. Dabei werden zu jeder Zeile des rechten Operanden alle Zeilen des linken Operanden gesucht, die die Join-Bedingung erfüllen. Der rechte Operand ist hier ein Table-Scan. Das SESAM-Kernsystem soll die Tabelle ROCAT.DDLOPT.DEPT lesen und bereits beim Lesen dafür sorgen, dass die Tabelle nach DEPT_NAME aufsteigend geordnet wird. Dazu verwendet das System einen Hilfsindex, der zur Realisierung der Eindeutigkeitsbedingung angelegt wurde:

```
sortorder: <
   DEPT_NAME #3 ascending
>
sort_index: UI9941102110408000
```

Der linke Operand des Nested-Loop-Join ist ebenfalls ein Table-Scan. Diesem wird jedoch folgendes Prädikat mitgegeben:

```
pred:
      ((DEPT = (expr #1)))
```

Es sollen nur die Zeilen von ROCAT.DDLOPT.EMP geliefert werden, die das Prädikat erfüllen. Der Ausdruck (expr #1) bezieht sich auf die Außenreferenz:

```
   (expr #1) ::= DEPT_NO
```

Diese bezieht sich auf den gerade aktuellen Wert der Spalte DEPT_NO im rechten Operanden.

Zur Berechnung des Nested-Loop-Join wird für jede Zeile des rechten Operanden der linke Operand berechnet. Da dieser nur jeweils die Zeilen enthält, die das Prädikat erfüllen, besteht das Ergebnis also aus allen Paaren von Zeilen aus rechtem und linkem Operanden. Diese Tabelle ist nach DEPT_NAME sortiert, erfüllt also die Voraussetzung von sorted_group.

Zusammenfassung

Die Tabelle DEPT wird zeilenweise gelesen. Dabei wird ein Index verwendet, der die Sortierung nach DEPT_NAME bewirkt. Für jede Zeile von DEPT wird die Tabelle EMP nach allen Zeilen durchsucht, deren Spalte DEPT denselben Wert enthält wie die Spalte DEPT_NO in der aktuellen Zeile von DEPT. Jede gefundene Kombination von Zeilen aus DEPT und EMP bildet nun eine Zeile. Alle Zeilen mit demselben Wert von DEPT_NAME (sie sind unmittelbar aufeinanderfolgend) werden zu jeweils einer Gruppe zusammengefasst. Daraus wird eine neue Zeile gebildet, die als Spalten den Wert von DEPT_NAME und den Durchschnitt von SAL in dieser Gruppe enthält.

Ein Verbesserungsversuch

Wie die obige Analyse gezeigt hat, wird die Tabelle EMP für jede Abteilung von vorn bis hinten durchsucht. Hier könnte ein Index auf EMP.DEPT die Zahl der zu lesenden Sätze stark einschränken.

```
CREATE INDEX ddlopt.empdept(dept) ON TABLE ddlopt.emp
```

Beim erneuten Optimieren unserer Anweisung erhalten wir folgende Erklärung (Ausschnitt):

```
...
left: table_scan [
   outer_reference_s: <
      (expr #1) ::= DEPT_NO
   >
   base: ROCAT.DDLOPT.EMP
   column_s: <
      SAL,
      DEPT
   >
   used index: <EMPDEPT>
   isolation level: set at runtime
   lock mode: shared
   cost_ratio: 3.000000E+00/2.000000E+00
   pred:
         ((DEPT = (expr #1)))
]
...
```

Der Index EMPDEPT wird jetzt verwendet, es werden also nur noch die Sätze von EMP gelesen, die zur Treffermenge gehören. Das Prädikat ist sehr selektiv, so dass der Overhead zum Lesen des Index mehr als kompensiert wird.

Ein ganz anderes Ergebnis

Es kann nicht oft genug betont werden, dass das Ergebnis der Optimierung auch von der Statistik-Information der Datenbank abhängt. Deshalb muss bei der Analyse der Abarbeitungspläne mit der Originaldatenbank oder mit realistischen Testdaten gearbeitet werden. Schon relativ kleine Änderungen können ein ganz anderes Bild liefern.
Wie das aussehen kann, soll hier mit einem kleinen Trick demonstriert werden (die obige Anweisung wurde geringfügig geändert):

```
SELECT dept_name, AVG(sal)
FROM emp, (SELECT * FROM dept WHERE dept_name <= 'zzz') AS d
WHERE emp.dept = d.dept_no
GROUP BY dept_name
```

Es ist ein Prädikat eingefügt worden, von dem der Optimizer annimmt, dass es die Treffermenge der Tabelle DEPT reduziert. Die Kosten- und Trefferabschätzung führt jetzt auf einen völlig anderen Abarbeitungsplan (Ausschnitt):

```
dynamic_select_stmt  [
  query:  sorted_group  [
     from:  sort  [
        from:  mjoin  [
           left:  table_scan  [
              base: ROCAT.DDLOPT.EMP
              sortorder: <
                 DEPT ascending
              >
              sort_index: EMPDEPT
              pred:
                    ((DEPT IS NOT NULL))
           ]
           right:  table_scan  [
              base: ROCAT.DDLOPT.DEPT
              sortorder: <
                 DEPT_NO ascending
              >
              sort_index: Primary Key
              pred:
                    ((DEPT_NAME #2 <= 'zzz'))
           ]
           joincond: (row(DEPT) = row(DEPT_NO))
        ]
        sort_spec_s: <
           DEPT_NAME #2 ascending
        >
     ]
     group_col_s: <
        DEPT_NAME ::= DEPT_NAME #2
     >
     aggregate_s: <
        (column #1) ::= all_avg(SAL)
     >
  ]
]
```

Hier werden der Index EMPDEPT von EMP sowie der Primärschlüssel von DEPT dazu verwendet, die beiden Join-Operanden nach den Spalten in der Join-Bedingung zu sortieren. Die hier verwendete Join-Operation ist der so genannte Merge-Join (mjoin). Dabei werden beide Operanden-Tabellen parallel vorwärts gelesen, und nur wenn einer der Operanden Duplikate in den Join-Bedingungs-Spalten hat, wird der andere entsprechend zurückgesetzt.

Das Ergebnis des Merge-Join ist jedoch nach DEPT_NO und nicht nach DEPT_NAME sortiert, so dass eine explizite Sortierung bezüglich DEPT_NAME für die Sorted-Group-Operation erforderlich ist.

7.2 Tabellenwertige Operationen (RELATION)

Die in diesem Abschnitt behandelten Knoten haben als Ergebnis eine Tabelle. Sie werden zur Klasse RELATION zusammengefasst.
Der Interpreter bearbeitet die Tabellen zeilenweise. Eine Basistabelle wird zeilenweise gelesen, wobei Prädikate auf Indizes genutzt werden, um die Treffermenge möglichst klein zu halten. Für die übrigen Tabellen wird im Interpreter im Wesentlichen eine Eröffnungsphase und eine Lesephase unterschieden. In der Lesephase wird satzorientiert gearbeitet.
D.h., der Interpreter gibt beim Lesen die Kontrolle an den Aufrufer zurück, wenn der nächste Satz feststeht, der seinerseits durch ein- oder mehrmaliges Lesen von Sätzen der darunter liegenden Tabellen ermittelt wurde.

7.2.1 cast_rows_to_rel

Semantik: Tabellenkonstruktor, d.h. Erzeugen einer (Zwischen-)Tabelle aus einer Menge von Zeilen. Tritt normalerweise auf bei der Nutzung von Zeilenvergleichen oder bei der Nutzung von Views, die über eine VALUES-Klausel definiert sind.

Attribute:

column_s	Die aus der erzeugten Tabelle benötigten Spalten
rows	Die Zeilen, aus denen die Tabelle erzeugt wird

SQL-Fragment:

```
SELECT    PK1,PK2,PK3,PK4,PK5
FROM      T
WHERE     (C11, C12, C13) in (1,11,111),(1,11,122),(1,11,133) )
```

Ausschnitt aus der Erklärung:

```
query: rest [
      from: table_scan [
         base: MYCAT.MYSCHEMA.T
         column_s: <
          ...
         >
         isolation level: set at runtime
         lock mode: shared
         cost_ratio: 8.000000E+00/1.000000E+00
      ]
      pred:
         some [
            outer_reference_s: <
               (expr #1) ::= C11,
               (expr #2) ::= C12,
               (expr #3) ::= C13
            >
            arg: rest [
               from: cast_rows_to_rel [
                  column_s: <
                     (column #4),
                     (column #5),
                     (column #6)
                  >
                  rows: <
                     row(1, 11, 111),
                     row(1, 11, 122),
                     row(1, 11, 133)
                  >
               ]
               pred: (row((expr #1), (expr #2), (expr #3)) =
                           row((column #4), (column #5), (column #6)))
               cost_ratio: 1.000000E+00/2.999999E-01
            ]
            pred: true
         ]
```

7.2.2 cross

Semantik: Kreuzprodukt.
Eine Zeile entsteht durch Aneinanderreihung von Zeilen aus allen Operanden. Dabei wird der erste Operand für jeden Satz des zweiten Operanden durchlaufen, dieser für jeden Satz des dritten usw.

Attribute:

from_s	Liste der Operanden, die jeweils eine RELATION darstellen.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.

Das Schema von cross ergibt sich aus der Vereinigung der Schemata der Teiltabellen.

SQL-Fragment:

```
SELECT t1.tellerid, t2.tellerid
FROM teller t1, teller t2
```

Ausschnitt aus der Erklärung:

```
query: eproj [
   from:  cross  [
       from_s : <
         table_scan [
            column_s: <
               TELLERID #3
            >
         ],
         table_scan [
            column_s: <
               TELLERID #4
            >
         ]
      >
   ]
   column_s: <
      TELLERID ::= TELLERID #4,
      TELLERID #2 ::= TELLERID #3
   >
]
```

7.2.3 cursor_scan

Semantik: Der aktuelle Satz eines Cursors.
Der Satz, auf den der Cursor positioniert ist, wird erneut gelesen. cursor_scan tritt nur in Zusammenhang mit UPDATE... CURRENT- und DELETE... CURRENT-Anweisungen auf.

Attribute:

base, column_s, isolation level, lock mode haben dieselbe Bedeutung wie in table_scan.

SQL-Fragment:

```
UPDATE t
SET c1 = x
WHERE CURRENT OF c
```

Ausschnitt aus der Erklärung:

```
update_stmt [
   object_rows:  cursor_scan  [
       base : cat.schema.t
       column_s : <
         C1
      >
       isolation level : at least repeatable read
       lock mode : exclusive
   ]
   cursor_id_ref: MY_MODULE.C
   ...
]
```

7.2.4 Empty

Semantik: Die leere Tabelle.
(In der Algebra der Relationen ist es das Null-Element.)

SQL-Fragment:

```
--%PRAGMA SIMPLIFICATION ON
DECLARE any_cursor CURSOR FOR
SELECT * FROM account WHERE 4 = 5
```

Ausschnitt aus der Erklärung:

```
cursor_select_stmt [
   query:  Empty
   is_for_update: False
   read_only: False
]
```

7.2.5 eproj

Semantik: Tabelle, die für jede Zeile des Operanden eine Zeile enthält, deren Spalten entsprechend den Ausdrücken in column_s gebildet werden.
In den meisten Fällen verschmilzt eproj bei der Vereinfachungs-Phase (Simplification) des Optimizers mit einer anderen Relation mit explizitem Schema. Überflüssige eproj-Knoten deuten darauf hin, dass keine Vereinfachung stattgefunden hat. Sie kann erzwungen werden mit:

```
--%PRAGMA SIMPLIFICATION ON
```

Attribute:

column_s	Liste von Ausdrücken, die die Berechnung der einzelnen Spalten angeben.
from	Operand der einstelligen Operation
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.

SQL-Fragment:

```
SELECT balance*balance*balance FROM account
```

Ausschnitt aus der Erklärung:

```
eproj  [
   from : table_scan [
     ...
  ]
   column_s : <
     (column #1) ::= ((BALANCE * BALANCE) * BALANCE),
  >
]
```

7.2.6 full_outer_join

Semantik: Kreuzprodukt mit Bedingung, zusätzlich alle Zeilen beider Tabellen (jeweils aufgefüllt mit NULL), die darin nicht vorkommen.

Der linke Operand wird sequenziell gelesen. Für jede Zeile des linken Operanden wird der rechte Operand auf den Anfang gesetzt und sequenziell gelesen. Es werden alle rechten Zeilen gesucht, die die Join-Bedingung mit der aktuellen linken Zeile erfüllen.
Für jedes gefundene Paar aus linker und rechter Zeile wird eine Ergebniszeile gebildet.

Findet sich für eine linke Zeile keine rechte Zeile, so wird eine Ergebniszeile aus der linken gebildet, rechts durch NULL ergänzt.
Anschließend wird der rechte Operand sequenziell gelesen. Für jede Zeile des rechten Operanden wird der linke Operand auf den Anfang gesetzt und sequenziell gelesen. Findet sich dabei keine linke Zeile, die mit der rechten zusammen das Prädikat erfüllt, so wird eine Ergebniszeile aus der rechten gebildet, links durch NULL ergänzt.

Attribute:

column_s	Liste von Ausdrücken, die die Berechnung der einzelnen Spalten angeben.
left, right	Operanden, die jeweils eine RELATION darstellen
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.
pred	Join-Bedingung

SQL-Fragment:

```
SELECT p1.ps_partkey, p1.ps_availqty + p2.ps_availqty
FROM   partsupp AS p1 FULL JOIN partsupp AS p2
ON p1.ps_partkey = p2.ps_partkey AND
   p1.ps_suppkey <> p2.ps_suppkey AND
   p1.ps_availqty + p2.ps_availqty <= :maximum
```

Ausschnitt aus der Erklärung:

```
full_outer_join  [
   left : table_scan [
     column_s: <
        PS_PARTKEY #7,
        PS_SUPPKEY,
        PS_AVAILQTY
     >
   ]
    right : table_scan [
     column_s: <
        PS_PARTKEY #8,
        PS_SUPPKEY #9,
        PS_AVAILQTY #10
     >
   ]
    column_s : <
     PS_PARTKEY #11 ::= PS_PARTKEY #7,
     PS_AVAILQTY #12 ::= PS_AVAILQTY,
     PS_AVAILQTY #13 ::= PS_AVAILQTY #10
   >
    pred : (
     (PS_PARTKEY #8 = PS_PARTKEY #7)
     AND (PS_SUPPKEY <> PS_SUPPKEY #9)
     AND ((PS_AVAILQTY + PS_AVAILQTY #10) <= :MAXIMUM\1)
     )
]
```

7.2.7 full_outer_mjoin

Semantik: Kreuzprodukt mit Bedingung, zusätzlich alle Zeilen beider Tabellen (jeweils aufgefüllt mit NULL), die darin nicht vorkommen.

Der Interpreter geht von einer Sortierung der beiden Operanden gemäß dem Join-Prädikat aus. Dieses entspricht einer Gleich-Bedingung.
Beide Operanden werden sequenziell gelesen, bis ein Paar gefunden wird, das die Join-Bedingung erfüllt. Dann wird eine Ergebniszeile aus der linken und rechten Zeile gebildet. Für Zeilen des linken Operanden, die in keinem Paar vorkommen, wird eine Ergebniszeile aus der linken Zeile, aufgefüllt mit NULL, gebildet. Für Zeilen des rechten Operanden, die in keinem Paar vorkommen, wird eine Ergebniszeile aus der rechten Zeile, aufgefüllt mit NULL, gebildet.
Hat einmal eine Zeile des rechten Operanden (in den Spalten gemäß der Join-Bedingung) dieselben Werte wie die vorangehende, so wird der linke Operand um soviele Zeilen zurückgesetzt, wie zur vorangehenden Zeile des rechten Operanden gelesen wurden.

Attribute:

column_s	Liste von Ausdrücken, die die Berechnung der einzelnen Spalten angeben.
joincond	Die Equi-Join-Bedingung zwischen den beiden Operanden. Die Row-Komponenten von „joincond“ sind entsprechend der garantierten Sortierreihenfolge der Join-Columns angeordnet. Man beachte, dass hier ein Vergleich von Row-Expressions stattfindet, die mit dem Operator **value** gebildet werden.
left, right	Operanden, die jeweils eine RELATION darstellen, sortiert nach den Spalten, die in joincond vorkommen.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.
right_is_unique	Ist true, wenn der linke Operand niemals zurückgesetzt werden muss.

SQL-Fragment:

```
SELECT professors.pname ,  papers.title
FROM   professors FULL JOIN papers
       ON professors.pnr = papers.pnr
```

Ausschnitt aus der Erklärung:

```
full_outer_mjoin  [
   left : table_scan [
     column_s: <
        PNR,
        PNAME #1
     >
   ]
    right : table_scan [
     column_s: <
        PNR #2,
        TITLE #3
     >
   ]
    column_s : <
     PNAME ::= PNAME #1,
     TITLE ::= TITLE #3
   >
    joincond : (row(PNR #2) = row(PNR))
    right_is_unique : False
]
```

7.2.8 left_outer_join

Semantik: Kreuzprodukt mit Bedingung, zusätzlich alle Zeilen der linken Tabelle (aufgefüllt mit NULL), die darin nicht vorkommen.

Die Bedingung befindet sich unter dem rechten Operanden.
Der linke Operand wird sequenziell gelesen. Für jede Zeile des linken Operanden wird der rechte Operand auf den Anfang gesetzt und sequenziell gelesen. Der rechte Operand liefert jeweils nur die Zeilen, die die Join-Bedingung mit der aktuellen linken Zeile erfüllen. Für jedes gefundene Paar aus linker und rechter Zeile wird eine Ergebniszeile gebildet. Findet sich für eine linke Zeile keine rechte Zeile, so wird eine Ergebniszeile aus der linken gebildet, rechts durch NULL ergänzt.

Attribute:

column_s	Liste von Ausdrücken, die die Berechnung der einzelnen Spalten angeben.
left, right	Operanden, die jeweils eine RELATION darstellen.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.

SQL-Fragment:

```
SELECT dept.city, emp_city
FROM   dept LEFT JOIN emp
       ON dept.city <> emp_city
```

Ausschnitt aus der Erklärung:

```
      left_outer_join  [
         left : table_scan [
           base: ...DEPT
        ]
         right : rest [
           from: table_scan [
              base: ...EMP
           ]
           pred: (CITY #1 <> EMP_CITY #2)
        ]
         column_s : <
           CITY ::= CITY #1,
           EMP_CITY ::= EMP_CITY #2
        >
     ]
```

7.2.9 left_outer_mjoin

Semantik: Kreuzprodukt mit Bedingung, zusätzlich alle Zeilen der linken Tabelle (aufgefüllt mit NULL), die darin nicht vorkommen.

Der Interpreter geht von einer Sortierung der beiden Operanden gemäß dem Join-Prädikat aus. Dieses entspricht einer Gleich-Bedingung.
Beide Operanden werden sequenziell gelesen, bis ein Paar gefunden wird, das die Join-Bedingung erfüllt. Dann wird eine Ergebniszeile aus der linken und rechten Zeile gebildet. Für Zeilen des linken Operanden, die in keinem Paar vorkommen, wird eine Ergebniszeile aus der linken Zeile, aufgefüllt mit NULL, gebildet.
Hat einmal eine Zeile des rechten Operanden (in den Spalten gemäß der Join-Bedingung) dieselben Werte wie die vorangehende, so wird der linke Operand um soviele Zeilen zurückgesetzt, wie zur vorangehenden Zeile des rechten Operanden gelesen wurden.

Attribute:

column_s	Liste von Ausdrücken, die die Berechnung der einzelnen Spalten angeben.
joincond	Die Equi-Join-Bedingung zwischen den beiden Operanden. Die Row-Komponenten von „joincond“ sind entsprechend der garantierten Sortierreihenfolge der Join-Columns angeordnet. Man beachte, dass hier ein Vergleich von Row-Expressions stattfindet, die mit dem Operator **value** gebildet werden.
left, right	Operanden, die jeweils eine RELATION darstellen, sortiert nach den Spalten, die in joincond vorkommen.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.
right_is_unique	Ist True, wenn der linke Operand niemals zurückgesetzt werden muss.

SQL-Fragment:

```
SELECT professors.pname ,  papers.title
FROM   professors LEFT JOIN papers
       ON professors.pnr = papers.pnr
```

Ausschnitt aus der Erklärung:

```
left_outer_mjoin  [
   left : table_scan [
     column_s: <
        PNR,
        PNAME #1
     >
   ]
    right : table_scan [
     column_s: <
        PNR #2,
        TITLE #3
     >
   ]
    column_s : <
     PNAME ::= PNAME #1,
     TITLE ::= TITLE #3
   >
    joincond : (row(PNR #2) = row(PNR))
    right_is_unique : False
]
```

7.2.10 mjoin

Semantik: Kreuzprodukt mit Bedingung.

Der Interpreter geht von einer Sortierung der beiden Operanden gemäß dem Join-Prädikat aus. Dieses entspricht einer Gleich-Bedingung.
Beide Operanden werden sequenziell gelesen, bis ein Paar gefunden wird, das die Join-Bedingung erfüllt.
Hat einmal eine Zeile des rechten Operanden (in den Spalten gemäß der Join-Bedingung) dieselben Werte wie die vorangehende, so wird der linke Operand um soviele Zeilen zurückgesetzt, wie zur vorangehenden Zeile des rechten Operanden gelesen wurden.

Attribute:

joincond	Die Equi-Join-Bedingung zwischen den beiden Operanden. Die Row-Komponenten von „joincond“ sind entsprechend der garantierten Sortierreihenfolge der Join-Columns angeordnet. Man beachte, dass hier ein Vergleich von Row-Expressions stattfindet, die mit dem Operator **value** gebildet werden.
left, right	Operanden, die jeweils eine RELATION darstellen, sortiert nach den Spalten, die in joincond vorkommen.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.
right_is_unique	Ist True, wenn der linke Operand niemals zurückgesetzt werden muss.

SQL-Fragment:

```
DECLARE c1 CURSOR FOR
        account INNER JOIN history
        ON accnr = haccnr
```

Ausschnitt aus der Erklärung:

```
cursor_select_stmt [
    output: <
        ACCNR,
        ...
    >
    query:  mjoin  [
         left : table_scan [
            column_s: <
                HACCNR,
                ...
            >
            sortorder: <
                HACCNR ascending
            >
            ...
            pred: ((HACCNR IS NOT NULL))
        ]
         right : table_scan [
            column_s: <
                ACCNR,
                ...
            >
            sortorder: <
                ACCNR ascending
            >
            ...
        ]
         joincond : (row(HACCNR) = row(ACCNR))
         right_is_unique : true
    ]
    ...
]
```

7.2.11 nljoin

Semantik: Nested-Loop-Join zwischen den beiden Operanden left und right.

Der rechte Operand wird Zeile für Zeile gelesen. Für jede Zeile des rechten Operanden wird der linke Operand eröffnet und jede seiner Zeilen mit der rechten Zeile kombiniert.

Attribute:

left, right	Operanden, die jeweils eine RELATION darstellen. Der linke Operand enthält die Join-Bedingung mit Referenzen auf den rechten Operanden.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.

SQL-Fragment:

```
SELECT accnr, balance, amount
FROM (
      account AS a
      INNER JOIN
      history AS h
      ON a.accnr = h.haccnr
     ) AS x
```

Ausschnitt aus der Erklärung:

```
query: eproj [
   from:  nljoin  [
       left : table_scan [
         outer_reference_s: <
            (expr #1) ::= ACCNR #2
         >
         column_s: <
            AMOUNT #3,
            HACCNR
         >
         pred: (HACCNR = (expr #1))
      ]
       right : table_scan [
         column_s: <
            BALANCE #4,
            ACCNR #2
         >
      ]
   ]
   column_s: <
      ACCNR ::= ACCNR #2,
      BALANCE ::= BALANCE #4,
      AMOUNT ::= AMOUNT #3
   >
]
```

7.2.12 One

Semantik: Die Tabelle mit genau einer leeren Zeile.
(In der Algebra der Relationen ist es das Eins-Element.)

SQL-Fragment:

```
--%PRAGMA SIMPLIFICATION ON
SELECT COUNT (*)
FROM   t
WHERE  1 = 2
```

Ausschnitt aus der Erklärung:

```
eproj [
   from:  One
   column_s: <
      (column #2) ::= 0
   >
]
```

7.2.13 pre_rest

Semantik: Die Operanden-Tabelle wird eingeschränkt auf die Sätze, die das Restriktionsprädikat erfüllen. Es ist unabhängig vom Inhalt der Tabelle. Darum ist das Ergebnis entweder gleich dem Operanden oder die leere Tabelle.

Vor dem Lesen der ersten Zeile wird das Restriktionsprädikat ausgewertet. Wenn es erfüllt ist, werden die Zeilen des Operanden gelesen. Andernfalls ist die Tabelle leer.
Attribute wie bei rest.

SQL-Fragment:

```
SELECT sal
       FROM   emp
       WHERE  sal <= :p AND :p <= 10000;
```

Ausschnitt aus der Erklärung:

```
       pre_rest  [
          outer_reference_s : <
            (pred #2) ::= (:P\1 <= 10000)
          >
           from : table_scan [
             base: SICAT.SIETEC_DDLOPT.EMP
             pred:
                   ((SAL <= :P\1))
           ]
           pred : (pred #2)
      ]
```

7.2.14 rest

Semantik: Die Operanden-Tabelle wird eingeschränkt auf die Sätze, die das Restriktionsprädikat erfüllen.

Attribute:

from	Operand der einstelligen Operation.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.
pred	das Restriktionsprädikat.

SQL-Fragment:

```
SELECT balance FROM account
WHERE balance > ALL (SELECT haccnr FROM history)
```

Ausschnitt aus der Erklärung:

```
rest  [
   from : table_scan [
     base: SICAT.SIETEC_DC.ACCOUNT
     column_s: <
        BALANCE
     >
   ]
    pred :
     all [
        outer_reference_s: <
           (expr #1) ::= BALANCE,
           (pred #2) ::= ((expr #1) IS NULL)
        >
        arg: table_scan [
           outer_reference_s: <
              (expr #3) ::= (expr #1)
              (pred #4) ::= (pred #2)
           >
           base: SICAT.SIETEC_DC.HISTORY
           pred:
                 (
                    ((HACCNR >= (expr #3)) OR (HACCNR IS NULL))
                    OR (pred #4)
                 )
        ]
        pred: false
     ]
]
```

7.2.15 sort

Semantik: Sortierte Tabelle.

Vor dem Lesen des ersten Satzes wird die Operandentabelle in eine Zwischendatei geschrieben und sortiert. Dann wird die Zwischendatei gelesen.

Attribute:

from	Operand der einstelligen Operation.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.
sort_spec_s	Die Spalten, nach denen sortiert wird.

SQL-Fragment:

```
SELECT tellerid+1 AS tid
FROM teller
ORDER BY tid
```

Ausschnitt aus der Erklärung:

```
      sort  [
         from : eproj [
           from: table_scan [
               ...
           ]
           column_s: <
              TID ::= (TELLERID + 1)
           >
        ]
         sort_spec_s : <
           TID ascending
        >
      ]
```

7.2.16 sorted_group

Semantik: Gruppierungsoperation.

Eine Gruppe besteht aus allen Zeilen der Operandentabelle, deren in group_col_s angegebene Spalten dieselben Werte enthalten. Für jede Gruppe wird eine Ergebnis-Zeile ermittelt, die die Spalten aus group_col_s und die Spalten aus aggregate_s enthält, wobei die Letzteren über die Gruppe gebildet werden. Ist aggregate_s leer, degeneriert die Operation zur Duplikatentfernung. Ist group_col_s leer, wird die ganze Operandentabelle als eine einzige Gruppe angesehen.
Die Zeilen der Operandentabelle müssen nach den Gruppierungsspalten sortiert sein. Die Zeilen werden sequenziell gelesen. Die zu aggregierenden Spalten werden dabei nach Bedarf gezählt, aufsummiert usw. Beim Gruppenwechsel wird die Ergebniszeile gebildet, indem die Aggregatfunktionen berechnet werden.

Attribute:

aggregate_s	Definiert neue Spalten aus den Elementen einer Gruppe. Dafür gibt es folgende Spezialoperatoren: **card**, **all_count**, **all_min**, **all_max**, **all_sum**, **all_avg**, **dist_count**, **dist_sum**, **dist_avg**
from	Operand der einstelligen Operation, sortiert nach den group_col_s.
group_col_s	Die Spalten, die die Gruppierung definieren.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.

SQL-Fragment:

```
SELECT t1_dec_1,
       COUNT(*),
       COUNT(DISTINCT t1_num_1),
       MIN  (t1_num_1),
       MAX  (t1_num_1),
       SUM  (t1_num_1),
       AVG  (t1_num_1)
  FROM t1
  GROUP BY t1_dec_1
```

Ausschnitt aus der Erklärung:

```
sorted_group  [
   from : table_scan [
     ...
     sortorder: <
        t1_DEC_1 ascending,
        T1_NUM_1 ascending
     >
   ]
    group_col_s : <
     T1_DEC_1 #24 ::= T1_DEC_1
   >
    aggregate_s : <
     (column #25) ::= card,
     (column #26) ::= dist_count(T1_NUM_1),
     (column #27) ::= all_min(T1_NUM_1),
     (column #28) ::= all_max(T1_NUM_1),
     (column #29) ::= all_sum(T1_NUM_1),
     (column #30) ::= all_avg(T1_NUM_1)
   >
]
```

7.2.17 store_temp

Semantik: Identität.

Die Operanden-Tabelle wird in einer Datei zwischengespeichert.
Da das Anlegen der Zwischendatei eine teure Operation ist, wird sie nach Möglichkeit nicht doppelt ausgeführt. Der Interpreter prüft vor dem wiederholten Öffnen der Tabelle store_temp das Attribut outer_reference_s. Wenn sich keine Werte der outer_reference_s geändert haben, so wird auf die bereits vorhandene Zwischendatei zurückgegriffen.
U.a. bedeutet das, dass store_temp ohne outer_reference_s stets nur einmal berechnet wird.

Attribute:

from	Operand der einstelligen Operation.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.

SQL-Fragment:

```
SELECT ...
FROM ...
GROUP BY ...
HAVING xxx IN  ( SELECT SUM ( sal )
                 FROM   emp AS e2
                 GROUP BY dept
                 HAVING yyy  > MIN ( e2.sal )  )
```

Ausschnitt aus der Erklärung:

```
some [
   outer_reference_s: <
      (expr #1) ::= Ausdruck für xxx
      (expr #2) ::= Ausdruck für yyy
   >
   from: rest [
      outer_reference_s: <
         (expr #7) ::= (expr #1),
         (expr #8) ::= (expr #2)
      >
      from:  store_temp  [
          from : sorted_group [
            from: ...
            group_col_s: <
               DEPT #24 ::= DEPT #21
            >
            aggregate_s: <
               (column #25) ::= all_min(SAL #22),
               (column #26) ::= all_sum(SAL #23)
            >
         ]
      ]
      pred: (
         ((expr #7) = (column #26))
         AND ((expr #8) > (column #25))
         )
   ]
   pred: true
]
```

Anmerkung zum Beispiel

Die beiden HAVING-Klauseln sind zu einem Prädikat zusammengefasst. Die unter dem rest liegende Tabelle sorted_group hängt von keinen äußeren Werten ab. Durch den eingeschobenen Operator store_temp wird sie nur einmal berechnet (da keine outer_reference_s) und dann zwischengespeichert.

7.2.18 table_function_scan

Semantik: Lesen von Spaltenwerten aus einer CSV-Datei mit der Tabellenfunktion CSV()

CSV-Dateien können mit der Tabellenfunktion CSV() gelesen, ihre Werte mit SESAM/SQL-Mitteln interpretiert werden. Der Zugriff auf CSV-Dateien wird im Pragma EXPLAIN als Knoten „table_function_scan“ ausgegeben.

Derartige Dateizugriffe können performance-kritisch sein, da beim Zugriff auf Dateien keine Zugriffspfade wie Primärschlüssel oder Sekundärindizes benutzt werden können. Dateien werden stets sequentiell gelesen.

Attribute:

column_s	Die aus der CSV-Eingabedatei benötigten Spalten (die ausgegebenen Spaltennamen werden intern generiert)
input file	Name der CSV-Eingabedatei
field separator	Delimiter-Zeichen für die Spaltenwerte der CSV-Datei
quote character	Quote-Zeichen, in das die Spaltenwerte in der CSV-Datei eingeschlossen werden können
escape character	Escape-Zeichen zum Entwerten von Sonderzeichen

SQL-Fragment:

```
SELECT *
FROM TABLE (CSV(…))
WHERE …
```

Ausschnitt aus der Erklärung:

```
from: table_function_scan [
        column_s: <
             _TblCSVCol00001_,
             _TblCSVCol00007_,
             _TblCSVCol00004_,
             _TblCSVCol00003_
         >
          input file:'MYFILE.CSV'
          field separator:';'
          quote character:'"'
          escape character:'\'
 ]
...
```

7.2.19 table_scan

Semantik: Alle Sätze einer Basistabelle, die das angegebene Prädikat erfüllen, davon aber nur die angegebenen Spalten.

Falls das Prädikat Werte enthält, die erst zur Interpretationszeit bekannt sind (Parameter, Referenzen auf Spalten anderer Tabellen), werden vor dem Lesen der ersten Zeile die zu benutzenden Indizes ausgewählt.
Die Zeilen der Basistabelle werden dann entsprechend dem ausgewählten Zugriffspfad gelesen.

Attribute:

base	Die Basistabelle, angegeben in der Form Katalogname.Schemaname.Tabellenname
column_s	Liste der benötigten Spalten der Basistabelle
cost_ratio	Die angezeigten Zahlen dienen den Entwicklern zur Feinabstimmung des Optimizers
has_at_most_one_row	Ist True, falls es höchstens einen Treffer geben kann.
isolation level	Fest eingestellte Werte: uncommitted consistency level 1 read committed repeatable read serializable Wert wird zur Interpretationszeit bestimmt: set at runtime at least repeatable read (siehe auch Abschnitt „Synchronisation“ auf Seite 45)
lock mode	shared exclusive
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.
pred	Die ausgewählten Zeilen müssen das Prädikat erfüllen.
sortorder	Spalten, deren Ordnung (ascending oder descending) garantiert werden soll.
used index	Liste der Indizes, die für den Zugriff auf die Basistabelle infrage kommen (siehe Abschnitt „Zugriff auf Basistabellen“ auf Seite 36).

SQL-Fragment:

```
UPDATE account
SET balance = balance + :betrag
WHERE accnr >= :acc_id + 1
```

Ausschnitt aus der Erklärung:

```
update_stmt [
   object_rows:  table_scan  [
       outer_reference_s : <
         (expr #1) ::= (:ACC_ID\\2 + 1)
       >
       base : DCCAT.DC_SCHEMA.ACCOUNT
       column_s : <
         ACCNR,
         BALANCE #3
       >
       sortorder : <
         ACCNR ascending
       >
       used index : < Primary Key >
       isolation level : at least repeatable read
       lock mode : exclusive
       cost_ratio : 0.000000E+00/0.000000E+00
       pred : (ACCNR >= (expr #1))
   ]
   ...
]
```

7.2.20 union

Semantik: Union zwischen den beiden Operanden left und right.

Der linke und der rechte Operand werden nacheinander sequenziell gelesen.

Attribute:

column_s	definiert die Spalten der neuen Tabelle als Paare von Spalten der Operanden.
left, right	Operanden, die jeweils eine RELATION darstellen.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.

SQL-Fragment:

```
SELECT *
FROM  (
       SELECT dept FROM emp
           UNION
       SELECT dept_no AS dept FROM dept
       ) AS u
```

Ausschnitt aus der Erklärung:

```
sorted_group [
   from: sort [
      from:  union  [
          left : eproj [
            from: ...
            column_s: <
               DEPT #2 ::= ...
            >
         ]
          right : eproj [
            from: ...
            column_s: <
               DEPT #3 ::= ...
            >
         ]
          column_s : <
            DEPT #4 ::= (DEPT #2, DEPT #3)
         >
      ]
      sort_spec_s: <
         DEPT #4 ascending
      >
   ]
   group_col_s: <
      DEPT ::= DEPT #4
   >
]
```

Anmerkung zum Beispiel

Für die SQL-Operation UNION müssen Duplikate entfernt werden, deshalb werden die Operationen sort und sorted_group auf die union angewandt.

7.2.21 virtual_scan

Semantik: Lesen von Metainformationen, die nicht im Catalog-Space abgespeichert sind.

Beim Zugriff auf bestimmte Views des INFORMATION_SCHEMA werden Metadaten benötigt, die nicht im Catalog-Space abgespeichert sind. Beispielsweise ist die Information, die über die View SQL_FEATURES ermittelt werden kann, nicht fest im Catalog-Space abgespeichert. Diese Information wird versionsspezifisch vom SESAM/SQL Data Base Handler verwaltet. In Zugriffsplänen für Views des INFORMATION_SCHEMA, die derartige interne, nicht im Catalog-Space verwaltete Informationen nutzen, werden diese Zugriffe als Operator virtual_scan dargestellt. Dieser Operator repräsentiert einen Zugriff auf eine nur virtuell vorhandene und nicht explizit ansprechbare Metadaten-Tabelle im Catalog-Space.

Attribute:

base	Die angesprochene virtuelle Tabelle in der Form *catalog*.DEFINITION_SCHEMA.*tabelle*.
column_s	Die von der virtuellen Tabelle benötigten Spalten.
cost_ratio	Geschätzte Kosten des Zugriffs; benötigt für die Erstellung des Plans durch den Optimizer.

SQL-Fragment:

```
SELECT *
FROM   INFORMATION_SCHEMA.SQL_FEATURES
WHERE  FEATURE_ID < 'E020'
```

Ausschnitt aus der Erklärung:

```
...
   from: virtual_scan [
      base: TESTCAT.DEFINITION_SCHEMA.SQL_FEATURES
      column_s: <
         SUB_FEATURE_NAME,
         SUB_FEATURE_ID,
         IS_VERIFIED_BY,
         IS_SUPPORTED,
         FEATURE_SUBFEATURE_PACKAGE_CODE,
         FEATURE_NAME,
         FEATURE_ID,
         COMMENTS
      >
      cost_ratio: 1.000000E+00/3.590000E+02
    ]
...
```

7.3 DML-Anweisungen

7.3.1 UPDATE-Anweisung

Semantik: Ändern von Zeilen einer Basistabelle.

Die Zeilen von object_rows werden bestimmt und geändert. check_on_the_fly wird verifiziert und die Zeile zurückgeschrieben.
Zum Abschluss check_after_all verifizieren.

Attribute:

check_after_all	Ein Prädikat, dessen Richtigkeit auf Grund von Integritätsbedingungen getestet werden muss. Die Prüfung erfolgt am Ende der Anweisung, weil es sich auf mehrere/alle Sätze der Basistabelle bezieht. (Steht hier true, braucht nichts geprüft zu werden.)
check_on_the_fly	Ein Prädikat, dessen Richtigkeit auf Grund von Integritätsbedingungen getestet werden muss. Die Prüfung erfolgt vor jeder einzelnen Satzänderung. (Steht hier true, braucht nichts geprüft zu werden.)
const_col_s	Siehe updated_col_s.
cursor_id_ref	Wenn die object_rows cursor_scan enthalten, so steht hier eine Referenz auf den Cursor in der Form Modulname.Cursorname
object_rows	Tabelle, die die zu ändernden Sätze bestimmt.
target	Die zu ändernde Basistabelle. Der Knoten Base-Table wird stets in der Form Katalogname.Schemaname.Tabellenname angezeigt.
updated_col_s, const_col_s	Sie definieren die Spalten der Basistabelle, die zu ändern sind. const_col_s enthält diejenigen Werte, die sich während der Abarbeitung nicht ändern, also für jede zu ändernde Zeile gleich sind. Die auszuführenden Änderungen erscheinen in der Erklärung als eine Liste von „column_set“s: links steht die zu ändernde Spalte, rechts der einzutragende Wert. (Zur Knotenform siehe Abschnitt „Ausdrücke, Funktionsaufrufe und Prädikate“ auf Seite 195.) Als Werte sind alle Arten von Ausdrücken möglich, darunter auch diese Größen, die der Interpreter ermittelt: **user, system_user, current_catalog, current_isolation_level, current_referenced_catalog, current_schema, current_date, current_time, current_timestamp** für CURRENT_USER, SYSTEM_USER usw.

SQL-Fragment:

```
UPDATE account
SET balance = balance + :amount
WHERE accnr = :acc_id
```

Ausschnitt aus der Erklärung:

```
update_stmt  [
   check_after_all : true
   target : DCCAT.DC_SCHEMA.ACCOUNT
   object_rows : rest [
     from: ...
     pred: (ACCNR = :ACC_ID\\2)
  ]
   check_on_the_fly : true
   cursor_id_ref : Void
   const_col_s : <>
   updated_col_s : <
     BALANCE ::= (BALANCE #2 + :AMOUNT\\1)
  >
]
```

7.3.2 INSERT-Anweisung

Semantik: Einfügen von Zeilen in eine Basistabelle.

Für jede Zeile der Tabelle object_rows wird check_on_the_fly verifiziert. Dann wird die Zeile in die Tabelle target eingefügt.
Zum Abschluss wird check_after_all verifiziert.

Attribute:

check_after_all
: Ein Prädikat, dessen Richtigkeit auf Grund von Integritätsbedingungen getestet werden muss. Die Prüfung erfolgt am Ende der Anweisung, weil es sich auf mehrere/alle Sätze der Basistabelle bezieht. (Steht hier true, braucht nichts geprüft zu werden.)

check_on_the_fly
: Ein Prädikat, dessen Richtigkeit auf Grund von Integritätsbedingungen getestet werden muss. Die Prüfung erfolgt vor jeder einzelnen Satzänderung. (Steht hier true, braucht nichts geprüft zu werden.)

const_col_s
: Siehe inserted_col_s.

inserted_col_s, const_col_s
: Sie definieren die Spalten der Basistabelle, die zu setzen sind. const_col_s enthält diejenigen Werte, die sich während der Abarbeitung nicht ändern, also für jede einzufügende Zeile gleich sind. Die auszuführenden Einfügungen erscheinen in der Erklärung als eine Liste von „column_set“s: links steht die zu setzende Spalte, rechts der einzutragende Wert. (Zur Knotenform siehe Abschnitt „Ausdrücke, Funktionsaufrufe und Prädikate“ auf Seite 195.)
Als Werte sind alle Arten von Ausdrücken möglich, darunter auch diese Größen, die der Interpreter ermittelt: **user, system_user, current_catalog, current_isolation_level, current_referenced_catalog, current_schema, current_date, current_time, current_timestamp** für CURRENT_USER, SYSTEM_USER usw.
In der Insert-Anweisung kann auch der Ausdruck **generate_key** erscheinen. Er veranlasst die automatische Generierung eines Schlüssels.

object_rows
: Tabelle, die die einzufügenden Sätze bestimmt.

output
: Wenn „*“ als Wert für eine Spalte angegeben wurde, steht hier eine Referenz auf diese Spalte, deren Wert dem Ausgabeparameter zugeordnet wird.

target
: Die zu ändernde Basistabelle. Der Knoten Base-Table wird stets in der Form Katalogname.Schemaname.Tabellenname angezeigt.

SQL-Fragment:

```
INSERT INTO account
VALUES (*, :abranchid       )
      RETURN INTO :gen
```

Ausschnitt aus der Erklärung:

```
       insert_stmt  [
          output : <
            ACCNR
          >
          check_after_all : true
          target : DCCAT.DC_SCHEMA.ACCOUNT
          object_rows : One
          check_on_the_fly : true
          const_col_s : <
            ACCNR ::= generate_key,
            ABRANCHID ::= :ABRANCHID\\1
          >
      ]
```

SQL-Fragment:

```
INSERT INTO history (haccnr, amount, htellerid, hrest)
SELECT * FROM account
```

Ausschnitt aus der Erklärung:

```
insert_stmt  [
   check_after_all : true
   target : DCCAT.DC_SCHEMA.HISTORY
   object_rows : eproj [
     from: table_scan [
        base: DCCAT.DC_SCHEMA.ACCOUNT
        ...
     ]
     column_s: <
        ACCNR #1 ::= ACCNR,
        ABRANCHID #2 ::= ABRANCHID,
        BALANCE #3 ::= BALANCE,
        AREST #4 ::= AREST
     >
  ]
   check_on_the_fly : true
   inserted_col_s : <
     HACCNR ::= ACCNR #1,
     AMOUNT ::= ABRANCHID #2,
     HTELLERID ::= BALANCE #3,
     HREST ::= AREST #4
  >
]
```

7.3.3 MERGE-Anweisung

Semantik:
Einfügen von Zeilen in eine Basistabelle oder Ändern von Zeilen in einer Basistabelle.

Zunächst werden die object_rows berechnet; dann wird für jede Zeile der object_rows entweder der INSERT in die Zieltabelle oder der UPDATE auf der Zieltabelle durchgefuehrt, je nachdem ob es zu einer Zeile aus der Quelltabelle bzgl. der ON-Bedingung korrespondierende Zeilen in der Zieltabelle gibt oder nicht.

Attribute:

check_after_all — Ein Prädikat, dessen Richtigkeit auf Grund von Integritätsbedingungen getestet werden muss. Die Prüfung erfolgt am Ende der Anweisung, weil es sich auf mehrere/alle Sätze der Basistabelle bezieht. (Steht hier true, braucht nichts geprüft zu werden.)

target — Die zu ändernde Basistabelle. Der Knoten Base-Table wird stets in der Form Katalogname.Schemaname.Tabellenname angezeigt.

object_rows — Berechnete Zwischentabelle, aus der Werte fuer den UPDATE bzw. INSERT genommen werden können und auf Grund der entschieden wird, welche Sätze in der Zieltabelle eingefügt bzw. geändert werden.

inserted_col_s, inserted_const_col_s
Sie definieren die Spalten der Basistabelle, die im INSERT-Fall des MERGE zu setzen sind.
inserted_const_col_s enthält diejenigen Werte, die sich während der Bearbeitung nicht ändern, also für jede einzufügende Zeile gleich sind.
Die auszuführenden Einfügungen erscheinen in der Erklärung als eine Liste von „column_set“s: links steht die zu setzende Spalte, rechts der einzutragende Wert.
In den Zuweisungen kann auch der Ausdruck generate_key erscheinen. Er veranlasst die automatische Generierung eines Zählfeldes.

ins_check_on_the_fly — Ein Prädikat, dessen Richtigkeit auf Grund von Integritätsbedingungen für jeden Satz getestet werden muss, der per INSERT eingefügt wird. (Steht hier true, braucht nichts geprüft zu werden.)

updated_col_s, updated_const_col_s
: Sie definieren die Spalten der Basistabelle, die im UPDATE-Fall zu ändern sind.
updated_const_col_s enthält diejenigen Werte, die sich während der Bearbeitung nicht ändern, also für jede zu ändernde Zeile gleich sind.
Die auszuführenden Änderungen erscheinen in der Erklärung als eine Liste von „column_set“s: links steht die zu ändernde Spalte, rechts der einzutragende Wert.

upd_check_on_the_fly
: Ein Prädikat, dessen Richtigkeit auf Grund von Integritätsbedingungen für jeden Satz getestet werden muss, der per UPDATE geändert wird. (Steht hier true, braucht nichts geprüft zu werden.)

SQL-Fragment:

```
merge into t10
      using t9
      on t10.c001 = t9.c001
when matched then update set c004 = t10.c004 + t9.c004
                           , c005 = 17
when not matched then insert (c001, c003) values (t9.c007, 27)
```

Ausschnitt aus der Erklärung:

```
       merge_stmt [
          check_after_all: true
          target: CAT.SCH.T10
          object_rows: sort [
             from: left_outer_join [
                left: table_scan [
                   base: CAT.SCH.T9
                   column_s: <
                      C001 #3,
                      C007,
                      C004 #4
                   >
                   isolation level: set at runtime
                   lock mode: shared
                   cost_ratio: 6.000000E+00/1.000000E+01
                ]
```

```
            right: table_scan [
               outer_reference_s: <
                  (expr #5) ::= C001 #3
               >
               base: CAT.SCH.T10
               column_s: <
                  C001 #6,
                  row_id,
                  C004 #7
               >
               used index: <Primary Key, NoName>
               ...
               has_at_most_one_row: True
               pred:

                        ((C001 #6 = (expr #5)))
            ]
            column_s: <
               C004 #8 ::= C004 #7,
               C004 #9 ::= C004 #4,
               C007 #10 ::= C007,
               (column #11) ::= row_id
            >
         ]
         sort_spec_s: <
            (column #11) descending
         >
      ]
      ...
      inserted_const_col_s: <
         C003 ::= 27
      >
      inserted_col_s: <
         C001 ::= C007 #10
      >
      ins_check_on_the_fly: true
      updated_const_col_s: <
         C005 ::= 17
      >
      updated_col_s: <
         C004 ::= (C004 #8 + C004 #9)
      >
      upd_check_on_the_fly: true
   ]
```

7.3.4 DELETE-Anweisung

Semantik: Löschen von Zeilen einer Basistabelle.

Die Zeilen der Tabelle object_rows werden bestimmt. Jedes Mal wird verifiziert, dass check_on_the_fly ohne diese Zeile gilt und dann die Zeile gelöscht.
Zum Abschluss wird check_after_all verifiziert.

Attribute:

check_after_all	Ein Prädikat, dessen Richtigkeit auf Grund von Integritätsbedingungen getestet werden muss. Die Prüfung erfolgt am Ende der Anweisung, weil es sich auf mehrere/alle Sätze der Basistabelle bezieht. (Steht hier true, braucht nichts geprüft zu werden.)
check_on_the_fly	Ein Prädikat, dessen Richtigkeit auf Grund von Integritätsbedingungen getestet werden muss. Die Prüfung erfolgt vor jeder einzelnen Satzänderung. (Steht hier true, braucht nichts geprüft zu werden.)
cursor_id_ref	Wenn die object_rows cursor_scan enthalten, so steht hier eine Referenz auf den Cursor in der Form Modulname.Cursorname
object_rows	Tabelle, die die zu streichenden Sätze bestimmt.
target	Die zu ändernde Basistabelle. Der Knoten Base-Table wird stets in der Form Katalogname.Schemaname.Tabellenname angezeigt.

SQL-Fragment:

```
DELETE FROM account
WHERE accnr > :acc_id
```

Ausschnitt aus der Erklärung:

```
delete_stmt  [
   check_after_all : true
   target : DCCAT.DC_SCHEMA.ACCOUNT
   object_rows : rest [
     from: ...
     pred: (ACCNR > :ACC_ID\\1)
  ]
   check_on_the_fly : true
   cursor_id_ref : Void
]
```

7.3.5 DECLARE CURSOR-Anweisung

Semantik: Deklaration eines Cursors.

Wenn der Cursor mit SCROLL eröffnet wird, so wird bei komplexen Abfragen eine Zwischendatei eröffnet (siehe Abschnitt „Zwischendateien“ auf Seite 203).
Beim Positionieren wird die Tabelle query an der entsprechenden Position gelesen.
Falls eine Zwischendatei existiert:
Solange die Tabelle sequenziell gelesen wird, wird jeder aus der Tabelle gelesene Satz in die Zwischendatei geschrieben. Bei irgendeiner anderen als der sequenziellen Positionierung wird zuerst der gesamte noch nicht gelesene Rest der Tabelle in die Zwischendatei geschrieben. Danach wird dann die Positionierung auf der Zwischendatei ausgeführt und von dort gelesen.

Attribute:

is_for_update	true genau dann, wenn die FOR UPDATE-Klausel spezifiziert ist.
output	Eine Liste von Ausdrücken, die den Ausgabeparametern der Reihe nach zugeordnet werden.
query	Die zu durchlaufende Tabelle.
read_only	true, wenn der Cursor für UPDATE... CURRENT und DELETE... CURRENT nicht zulässig ist.

SQL-Fragment:

```
DECLARE any_cursor CURSOR FOR
SELECT accnr FROM account
```

Ausschnitt aus der Erklärung:

```
cursor_select_stmt  [
   output : <
     ACCNR
  >
   query : table_scan [
     ...
   ]
   is_for_update : false
   read_only : false
]
```

7.3.6 Single-SELECT-Anweisung

Semantik: Lesen einer einzeiligen Tabelle.

Eine Zeile der Tabelle query wird gelesen und verifiziert, dass keine weitere Zeile vorhanden ist.

Attribute:

output	Eine Liste von Ausdrücken, die den Ausgabeparametern der Reihe nach zugeordnet werden.
query	Die einzeilige Tabelle.

SQL-Fragment:

```
SELECT accnr
INTO :accnr
FROM account
WHERE balance = 2000
```

Ausschnitt aus der Erklärung:

```
single_select_stmt  [
   output : <
     ACCNR
  >
   query : rest [
     ...
   ]
]
```

7.3.7 Dynamische SELECT-Anweisung

Ein dynamic_select_stmt wird vom Optimizer nur dann erzeugt, wenn eine Unterscheidung von cursor_spec und single_select_stmt bei dynamischer SQL zur Übersetzungszeit nicht möglich ist.
Erst bei der Anwendung wird klar, als was es verwendet wird. Danach bestimmen sich dann Semantik und Abarbeitung.
Die Attribute sind so wie die für cursor_select_stmt bzw. single_select_stmt, je nachdem, als was die Anweisung verwendet wird.

SQL-Fragment:

```
SELECT accnr FROM account WHERE  ? = 1
```

Ausschnitt aus der Erklärung:

```
dynamic_select_stmt  [
   output : <
     ACCNR
  >
   query : pre_rest [
     ...
  ]
   is_for_update : false
   read_only : false
]
```

7.3.8 Interne EXPORT DATA-Anweisung

Semantik: Bestimmung des Zugriffsplans für die Ermittlung der zu exportierenden Sätze bei einer EXPORT-Anweisung mit WHERE-Klausel.

Bei einer EXPORT-Anweisung können mit der WHERE-Klausel die zu exportierenden Sätze einer Tabelle ausgewählt werden. In diesem Fall wird bei der Verarbeitung der EXPORT-Anweisung intern eine so genannte EXPORT DATA-Anweisung eingegeben, mit der die betroffenen Sätze bestimmt werden. Der Optimizer erzeugt auch für diese interne Anweisung einen Zugriffsplan. Durch Angabe des Pragmas EXPLAIN bei der Utility-Anweisung EXPORT TABLE kann die Ausgabe dieses Plans für die interne EXPORT DATA-Anweisung auf Datei erzwungen werden.

Im Zugriffsplan einer EXPORT DATA-Anweisung liefert der zugrunde liegende table_scan unter anderem die Satznummer der Treffersätze (als spezielles Attribut row_id), da diese für die interne Abarbeitung der Anweisung benötigt wird.

Der Plan für die interne EXPORT DATA-Anweisung wird erst während der Ausführung (Execute) der Utility-Anweisung EXPORT erzeugt und nicht schon bei deren Übersetzung (Prepare).

Attribute:

output	Besteht nur aus dem beschriebenen speziellen Attribut row_id.
query	Die aus den Treffersätzen bestehende Ergebnistabelle.
target	Die Tabelle, aus der Daten exportiert werden.

SQL-Fragment:

```
EXPORT TABLE T10 INTO FILE 'EXPORTFILE.T10' WHERE C001 < 100
```

Ausschnitt aus der Erklärung
(der dabei intern eingegebenen EXPORT DATA-Anweisung):

```
export_data_stmt [
   output: <
      _ROW_ID
   >
   query: eproj [
      from: table_scan [
         base: CAT.SCH.T10
         column_s: <
            row_id,
            ...
         >
         sortorder: <
            C001 ascending
         >
         sort_index: Primary Key
         used index: <Primary Key>
         ...
         pred:
               ((C001 < 100))
      ]
      column_s: <
         _ROW_ID ::= row_id,
         C001 #1 ::= C001
      >
   ]
   target: CAT.SCH.T10
]
```

7.3.9 Interne SELECT FOR UNLOAD-Anweisung

Semantik: Ermittlung der zu entladenden Sätze bei einer Utility-Anweisung UNLOAD ONLINE mit WHERE-Klausel.

Bei einer Utility-Anweisung UNLOAD ONLINE können mit der WHERE-Klausel die zu entladenden Sätze einer Tabelle oder eines Views ausgewählt werden. In diesem Fall wird bei der Verarbeitung der Anweisung intern eine so genannte SELECT FOR UNLOAD-Anweisung aufgerufen, mit der die betroffenen Sätze bestimmt werden. Der Optimizer erzeugt auch für diese interne Anweisung einen Zugriffsplan. Durch Angabe des Pragmas EXPLAIN bei UNLOAD ONLINE wird der Plan für die interne SELECT FOR UNLOAD-Anweisung in eine Datei ausgegeben.

Der Plan für die interne SELECT FOR UNLOAD-Anweisung wird erst während der Ausführung (Execute) von UNLOAD ONLINE erzeugt, nicht schon bei deren Übersetzung (Prepare).

Attribute:

output	Werte, die entladen werden sollen
query	Die aus den Treffersätzen bestehende Ergebnistabelle

SQL-Fragment:

```
UNLOAD ONLINE TABLE T INTO FILE 'MYFILE.TXT'
DELIMITER_FORMAT TERMINATED BY ';' WHERE C001 < 100
```

Ausschnitt aus der Erklärung:

```
unload_data_stmt [
   output: <
      C001,
      ...
   >
   query: eproj [
      from: table_scan [
         base: CAT.SCH.T
         column_s: <
            C001 #1,
            ...
         >
         used index: <Primary Key>
         ...
         pred:
               ((C001 #1 < 100))
      ]
      column_s: <
         C001 ::= C001 #1,
         ...
      >
   ]
]
```

7.3.10 CALL-Anweisung

Semantik: Aufruf einer Prozedur (Stored procedure).

Die Prozedur `myproc` hat folgende Definition:

```
create procedure myproc(in par1 integer, in par2 integer, out par3 integer)
modifies sql data
begin
   declare var1 integer default -1;
   declare var2 integer default null;
   set var2 = (select max(c001) from t1) + par1 * par2 * var1;

   if ((select min(c001) from t2) + var2 > 0)
      then set par3 = var2 * 1000;
      elseif (var2 = 0)
        then set par3 = 0;
        else set par3 = 1000000 * var2;
   end if;
end
```

Der Aufruf dieser Prozedur erfolgt mit:

```
--%pragma explain into 'MYPROC.EXPLAIN'
call myproc(17,18,?)
```

Es wird die nachfolgende Erklärung erzeugt. Diese enthält die üblichen Erklärungen der in der Prozedur enthaltenen DML-Anweisungen INSERT, UPDATE, DELETE, MERGE, SELECT und DECLARE CURSOR.

Sie enthält auch die Erklärungen der Prozedur-Anweisungen IF, LOOP, LEAVE und SET:

`if_stmt`
: Erklärung der IF-Anweisung mit einem `if`-Block, mit keinem, einem oder mehreren `elseif`-Blöcken und evtl. einem `else`-Block.
Ein `if`- bzw. `elseif`-Block besteht aus einer Bedingung und den dazu gehörenden Anweisungen im `then`-Block. Diese werden durchlaufen, wenn die Bedingung zutrifft.
Der `else`-Block enthält die Anweisungen, die durchlaufen werden, wenn keine Bedingung in den `if`- bzw. elseif-Blöcken erfüllt ist.

`assign_stmt`
: Erklärung der SET-Anweisung. Neben der Zuweisung `target` wird eine Erklärung des Ausdrucks `source` geliefert. `source` kann Subqueries enthalten, die dann wie in gewöhnlichen DML-Anweisungen erklärt werden.

`loop_stmt`
: Erklärung der LOOP-Anweisung. Sie enthält die Folge von Anweisungen `stmt_s` des Schleifenrumpfs.

`leave_stmt`
: Erklärung der LEAVE-Anweisung. Sie beschreibt, dass eine Schleife verlassen wird. Bei geschachtelten Schleifen beschreibt sie auch welche Schleife verlassen wird.

Es werden auch die Ausgabevariablen der Prozedur, die lokalen Variablen mit Defaultwerten sowie die Eingabewerte der CALL-Anweisung ausgegeben. Die Eingabewerte können Unterabfragen enthalten, deren Berechnung dann wie in anderen DML-Anweisungen erklärt wird.

Ausschnitt aus der Erklärung:

```
--%pragma explain into 'MYPROC.EXPLAIN'
call myproc(17,18,?)
call_stmt [
   output: <
      :par->PAR3
   >
   stmt: compound_stmt [
      is_atomic: False
      variable_s: <
         :var->VAR1,
         :var->VAR2
      >
      stmt_s: <
         assign_stmt [
            target: :var->VAR2
            source: plus [
               left: cast_rel_to_row [
                  from: eproj [
                     from: sorted_group [
                        from: table_scan [
                           base: THUCAT.JOINSCHEMA.T1
                           column_s: <
                              C001
                           >
                           isolation level: set at runtime
                           lock mode: shared
                           cost_ratio: 1.000000E+01/9.200000E+01
                        ]
                        aggregate_s: <
                           (column #1) ::= all_max(C001)
                        >
                     ]
                     column_s: <
                        (column #2) ::= (column #1)
                     >
```

```
                  ]
               ]
               right: ((:par->PAR1 * :par->PAR2) * :var->VAR1)
            ]
         ],
         if_stmt [
            if:
                  gt [
                     left: cast_rel_to_row [
                        from: sorted_group [
                           from: table_scan [
                              base: THUCAT.JOINSCHEMA.T2
                              column_s: <
                                 C001 #3
                              >
                              isolation level: set at runtime
                              lock mode: shared
                              cost_ratio: 1.000000E+00/1.000000E+00
                              has_at_most_one_row: True
                           ]
                           aggregate_s: <
                              (column #4) ::= all_min(C001 #3)
                           >
                        ]
                     ]
                     right: (- :var->VAR2)
                  ]
            then: <
               assign_stmt [
                  target: :par->PAR3
                  source: (:var->VAR2 * 1000)
               ]
            >,
            elseif: (:var->VAR2 = 0)
            then: <
               assign_stmt [
                  target: :par->PAR3
                  source: 0
               ]
            >
            else: <
               assign_stmt [
                  target: :par->PAR3
                  source: (1000000 * :var->VAR2)
               ]
            >
         ]
      >
```

```
      proc_var_default_s: <
         -1,
         null
      >
   ]
   value_s: <
      17,
      18,
      :\1
   >
]
```

7.4 Quantoren

some, all - Quantoren. Es wird geprüft, ob ein Prädikat auf wenigstens einen Satz bzw. auf alle Sätze einer Tabelle zutrifft.

Wenn alle outer_reference_s bei einer nachfolgenden Auswertung dieselben Werte haben wie zuvor, so unterbleibt die erneute Auswertung des Quantors, und es wird das alte Ergebnis verwendet. Hat die Tabelle EMP im Folgenden Beispiel mehrere aufeinanderfolgende Zeilen mit demselben Wert SAL, so wird der Quantor dafür nur einmal berechnet. (Ein Quantor ohne outer_refence_s wird nur ein einziges Mal ausgewertet.)

Attribute:

arg	Die zu prüfende Tabelle.
outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können.
pred	Die Prüfbedingung. Mit dem Ziel, die Treffermengen aller Tabellen möglichst klein zu halten, versucht der Optimizer, die Bedingung in das Argument zu schieben. Darum hat some meistens das Prädikat true; der Wert des Quantors ist dann also true, wenn die Tabelle nicht leer ist. Bei all wird versucht, das negierte Prädikat in das Argument zu schieben, darum hat all meistens das Prädikat false; der Wert des Quantors ist dann also true, wenn die Tabelle leer ist.

SQL-Fragment:

```
SELECT sal
FROM   emp
WHERE  sal = ANY ( SELECT 10*s FROM ...)
```

Ausschnitt aus der Erklärung:

```
rest [
   from: table_scan [
      ...
   ]
   pred: ((SAL IS NOT NULL)
      AND
       some  [
          outer_reference_s : <
            (expr #1) ::= SAL
         >
          arg : rest [
            ...
            pred: (
               ...
               AND ((expr #1) = (10 * S))
               )
         ]
          pred : true
      ]
   )
]
```

7.5 Wertabfrage

Eine Wertabfrage ist ein Ausdruck, der aus dem Datenbankinhalt berechnet wird. Sie erscheint in der Erklärung als Knoten **cast_rel_to_row**.

Attribute:

outer_reference_s	Liste von Ausdrücken und Prädikaten, die unabhängig von den übrigen Attributen des Knotens berechnet werden können. Wenn alle outer_reference_s bei einer nachfolgenden Auswertung dieselben Werte haben wie zuvor, so unterbleibt die erneute Auswertung der Wertabfrage, und es wird das alte Ergebnis verwendet.
from	Auszuwertende Tabelle (1 Zeile, 1 Spalte).

SQL-Fragment:

```
WHERE  ( SELECT subject
         FROM lectures AS l
         WHERE l.dnr = d.dnr
       )
     =
       ( SELECT ... )
```

Ausschnitt aus der Erklärung:

```
      eq [
         left:  cast_rel_to_row  [
             outer_reference_s : <
               (expr #1) ::= DNR
            >
             from : rest [
               ...
               pred: (DNR #2 = (expr #1))
            ]
         ]
         right: cast_rel_to_row [
            ...
         ]
      ]
```

7.6 Ausdrücke, Funktionsaufrufe und Prädikate

Ausdrücke, Funktionsaufrufe und Prädikate sind in der Regel nicht performance-kritisch, daher dient ihr Auftreten in der Erklärung vorrangig dazu, bestimmte Teile der SQL-Anweisung in der Erklärung wiederzufinden.

Ausdrücke, Prädikate usw. erscheinen in der Erklärung in ähnlicher Form, wie sie in der ursprünglichen SQL-Anweisung standen. Davon gibt es drei Ausnahmen:

1. Mehrere gleichnamige Spaltenreferenzen

   Treten Referenzen auf verschiedene Spalten gleichen Namens auf oder beziehen sich verschiedene Tabellen auf eine Spalte, so wird dem Spaltennamen eine Nummer angefügt, die den Bezug eindeutig macht.

   Beispiel

   ```
   ACCOUNT #3
   ```

2. Referenzen auf gemeinsame Teilausdrücke/-prädikate

   Referenzen auf gemeinsame Teilausdrücke/-prädikate werden in den outer_reference_s oder column_s definiert.

   Beispiel

   ```
   (expr #1) ::= SAL
   (pred #2) ::= ((expr #1) IS NULL)
   (column #3) ::= ('mnopqr' || T1_CHAR)
   ```

   Andere gemeinsame Teilausdrücke/-prädikate werden mit einer Marke versehen, die aus dem Knotennamen und einer Nummer besteht.

   Beispiel

   ```
   atom_def #11: (PNAME = :PROF1\\1)
   (SAL / scalar_literal #2: 4) > 5
   null #46: null = AMOUNT
   ```

3. Wertabfrage

   Wenn ein Wert aus dem Datenbankinhalt berechnet wird, so enthält der entsprechende Ausdruck einen tabellenwertigen Knoten, der die Syntax eines einfachen Ausdrucks sprengt (siehe Abschnitt „Wertabfrage“ auf Seite 194). Dann wird der Ausdruck in Knotenform ausgegeben.

   Beispiel

```
balance + (SELECT btotal FROM branch WHERE branchid = 1)

plus [
   left: BALANCE #2
   right:  cast_rel_to_row  [
       from : rest [
          ...]
      ]
    ]
```

Knotenformen bei Wertabfragen

Im folgenden werden alle Knotenformen aufgezählt, die in Zusammenhang mit Wertabfragen auftreten können.

In den Attributen const_col_s, inserted_col_s und updated_col_s:

```
column_set  [
   column :
   expr :
]
```

Ausdrücke

Die Knoten **plus, minus, times, divide, concat, negation**, **cast** stehen für +, binäres -, *, /, ||, unäres - und die Cast-Funktion.

Beispiel

```
plus  [
   left :
   right :
 ]
  negation  [
     arg :
 ]


cast  [
   arg :
 ]
```

Die Knoten **position, char_length, substring, upper, lower, trim_both, trim_leading und trim_trailing** stehen für die Textfunktionen POSITION, CHAR_LENGTH, SUBSTRING, UPPER, LOWER, TRIM(BOTH...), TRIM(LEADING...) und TRIM(TRAILING...).

Beispiel

```
position  [
    arg :
    part :
]


substring  [
   arg :
   from :
   for :
]
```

Ebenso werden im Zusammenhang mit Wertabfragen bedingte Ausdrücke u.U. in Knotenform ausgegeben.

Dabei stehen die Knoten **nullif** und **coalesce** für die entsprechenden Schlüsselwörter.

Beispiel

```
nullif  [
   left :
    right :
]
```

und

```
coalesce  [
    expr_s : <
                -- comma-delimited list of WHEN/THEN expressions
>
]
```

Der Knoten **case** steht für die CASE-Klausel.

Beispiel

```
case  [
    arg :       -- Optional CASE - argument
    default :   -- Value of the ELSE clause
    case_s : <
                -- comma-delimited list of WHEN/THEN expressions
   >
]
```

Dabei können die WHEN/THEN-Ausdrücke je nach Komplexität der Ausdrücke als SQL-Äquivalente oder wieder als Knotenform **when_then** auftreten.

Beispiel

```
when_then  [
when :
then :
]
```

Funktionen wie ABS(), MOD() usw. werden normalerweise wie im SQL-Text ausgegeben. Allerdings kann es je nach Komplexität des Arguments auch zu einer Ausgabe als Knoten mit dem Namen „specific_value_call“ kommen.

Funktionsaufrufe

Beispiel

Der einfache Funktionsaufruf `select fff(c001+c002) from t10` liefert:

```
dynamic_select_stmt [
   output: <
      (column #1)
   >
   query: eproj [
      from: table_scan [
         base: THUCAT.JOINSCHEMA.T10
         column_s: <
            C001,
            C002
         >
         isolation level: set at runtime
         lock mode: shared
         cost_ratio: 6.000000E+00/9.000000E+00
      ]
      column_s: <
         (column #1) ::= THUCAT.JOINSCHEMA.FFF((C001 + C002))
      >
   ]
   is_for_update: False
   read_only: True
```

Der Funktionswert wird also wie erwartet als einfacher Funktionsaufruf mit vollqualifiziertem Funktionsnamen dargestellt: `THUCAT.JOINSCHEMA.FFF((C001 + C002))`

Wenn die Parameter der Funktion einen komplexen Ausdruck enthalten, der die Syntax eines einfachen Ausdrucks sprengt (siehe Abschnitt „Wertabfrage“ auf Seite 194), dann wird der Funktionsaufruf evtl. in Knotenform ausgegeben.

Für den Funktionsaufruf in der Anweisung
`select fff((select max(c001) from t2 where c001 < t1.c001)) from t1`
wird der Wert als `function_value` Knoten so ausgegeben:

```
function_value [
      function: THUCAT.JOINSCHEMA.FFF
      value_s: <
         cast_rel_to_row [
            outer_reference_s: <
               (expr #2) ::= C001
            >
            from: eproj [
               from: sorted_group [
                  from: table_scan [
                     outer_reference_s: <
                        (expr #3) ::= (expr #2)
                     >
                     base: THUCAT.JOINSCHEMA.T2
                     column_s: <
                        C001 #4
                     >
                     used index: <Primary Key, Interval>
                     isolation level: set at runtime
                     lock mode: shared
                     cost_ratio: 1.000000E+00/1.000000E+00
                     pred:
                              ((C001 #4 < (expr #3)))
                  ]
                  aggregate_s: <
                     (column #5) ::= all_max(C001 #4)
                  >
               ]
               column_s: <
                  (column #6) ::= (column #5)
               >
            ]
         ]
      >
   ]
```

Prädikate

Die Knoten **eq, ne, lt, gt, le, ge** stehen für =, <>, <, >, <=, >=.

Beispiel

```
eq  [
   left :
   right :
]
```

Die Knoten **is_null, is_not_null** stehen für IS NULL, IS NOT NULL.

Beispiel

```
is_null  [
   arg :
]
```

Die Knoten **is_like, is_not_like** stehen für IS LIKE, IS NOT LIKE.

Beispiel

```
is_like  [
    match_value :       - String to be checked
    pattern :           - Pattern
    escape :            - Wildcard (joker character)
 ]
```

Die Knoten **between, not_between** stehen für BETWEEN, NOT BETWEEN.

Beispiel

```
not_between  [
   mid :
   left :
    right :
]
```

Der Knoten **reg_exp_like_pred** steht für LIKE_REGEX.

Beispiel

```
reg_exp_like_pred [
   match_value :
   pattern_automaton :
   op :
]
```

Der Knoten **castable_pred** steht für IS CASTABLE.

Beispiel

```
castable_pred [
  arg :
  op :
]
```

Für AND oder OR gibt es keine Knotenform, auch wenn die Operanden Knotenform haben.

7.7 Zwischendateien

Zwischendateien werden angelegt, wenn eine Tabelle mehrfach gelesen werden muss. Andere Gründe für Zwischenspeicherungen sind technischer Art, wie beispielsweise das Sortieren einer Tabelle (Operation sort).

Zwischendateien werden nach Möglichkeit vermieden. Vielmehr wird unter Ausnutzen der SESAM-Fähigkeiten versucht, bereits die Basistabelle geeignet zu lesen. Andererseits kann es sinnvoll sein, eine mehrfach benötigte und teuer zu erstellende Zwischenergebnis-Tabelle nur einmal zu berechnen. Dann erzeugt der Optimizer die Operation store_temp.

Auf Grund der Architektur des SQL-Handlers gibt es einen Fall, bei dem der Optimizer nicht entscheiden kann, ob eine Zwischendatei benötigt wird oder nicht. Denn dem Optimizer werden DECLARE CURSOR-Anweisungen nicht übergeben, sondern nur die darin enthaltenen Abfragen. Daher ist bei der Optimierung nicht bekannt, ob der Cursor scrollbar ist, und so müssen alle Tabellen prinzipiell zum Scrollen vorbereitet sein. Um nun nicht unnötigerweise Zwischendateien anzulegen, entscheidet der Interpreter anhand der ihm übergebenen Parameter, ob es tatsächlich erforderlich ist.

Die Zwischenspeicherung führt der Interpreter auf zwei verschiedene Weisen aus.
Zum Zeitpunkt des Eröffnens wird bei den Operationen store_temp und sort die gesamte Tabelle in die Zwischendatei geschrieben (und sortiert). Das Lesen der Tabelle erfolgt dann aus der Zwischendatei.
Wird ein Cursor im Scroll-Modus eröffnet, so wird beim Öffnen der Tabelle die Zwischendatei zwar angelegt, aber noch nicht gefüllt. Beim sequenziellen Lesen der Tabelle wird Zeile für Zeile in die Zwischendatei geschrieben. Bei der ersten vom sequenziellen Lesen abweichenden Positionierung jedoch wird die Tabelle bis ans Ende gelesen und in die Zwischendatei geschrieben. Danach wird auf der Zwischendatei positioniert und von dort gelesen.

Das Anlegen von Scroll-Zwischendateien erfolgt jedoch nicht blind. Wenn die darunter liegende Tabelle bereits zwischengespeichert oder eine Basistabelle ist, ist das Scrollen ohne Zwischendatei möglich.

8 Performance-relevante Aspekte von Utility-Funktionen und DDL-Anweisungen

Dieses Kapitel gibt Hinweise zum optimierten Einsatz von Utility-Funktionen und DDL-Anweisungen.

8.1 LOAD - Anwenderdaten in eine Basistabelle laden

Neben der Utility-Anweisung LOAD OFFLINE gibt es die Utility-Anweisung LOAD ONLINE.

Diese beiden Utility-Anweisungen unterscheiden sich voneinander

- durch ihr Sperrverhalten, also in der Art der Synchronisation mit anderen Anweisungen, die auf die gleichen Spaces, Tabellen und Indizes zugreifen
- durch ihren internen Ablauf

Daraus ergibt sich in Abhängigkeit von bestimmten Randbedingungen auch ein unterschiedliches Verhalten bezüglich der Performance.

Die folgenden Abschnitte liefern Anhaltspunkte, in welchen Situationen die Klauseln OFFLINE bzw. ONLINE gewählt werden sollten, um eine LOAD-Anweisung möglichst schnell und effizient durchzuführen.

Eine abgebrochene LOAD-Anweisung können Sie komfortabel und performant fortsetzen. Im Abschnitt „Fortsetzen einer abgebrochenen LOAD-Anweisung“ auf Seite 211 finden Sie performance-relevante Hinweise dazu.

8.1.1 Bearbeitung von LOAD in SESAM/SQL

Dieser Abschnitt stellt kurz die Unterschiede in der Bearbeitung von LOAD OFFLINE bzw. LOAD ONLINE in SESAM/SQL dar.

LOAD OFFLINE

Wie die meisten Utility-Funktionen sperrt LOAD OFFLINE die aktuelle Tabelle und ihre Indizes exklusiv. Auf sonstige Tabellen und Indizes der betroffenen Spaces kann parallel nur lesend zugegriffen werden. Die Datenzugriffe finden zum größten Teil in einer Servicetask statt.

Für die Verarbeitung einer Tabelle mit Primärschlüssel wird intern die Sortierung der Ladesätze nach dem Primärschlüsselwert benötigt. Falls die Ladedatei bereits entsprechend sortiert ist (d.h. der Parameter SORTED ist angegeben), werden die Ladesätze in dieser Reihenfolge aufbereitet. Andernfalls erzeugt SESAM/SQL eine Arbeitsdatei, in der alle Ladesätze in aufbereiteter (und möglicherweise expandierter) Form abgespeichert werden, und einen SORT-Puffer, in den alle Primärschlüsselwerte (mit Verweis auf den zugeordneten Satz in der Arbeitsdatei) eingetragen werden. Nachdem die Ladedatei vollständig bearbeitet ist, wird der SORT-Puffer sortiert.
Die aufbereiteten Sätze werden in die (im Allgemeinen nicht leere) Tabelle an der jeweils passenden Stelle eingemischt. Zu diesem Zweck werden die Primärdaten der Tabelle vollständig eingelesen, und bei Bedarf wird an der dem Primärschlüssel entsprechenden Stelle jeweils ein Satz eingefügt oder geändert. Parallel dazu werden die Zugriffspfade völlig neu aufgebaut.

Bei einer Tabelle ohne Primärschlüssel werden die Sätze in der Reihenfolge ihres Auftretens in der Ladedatei aufbereitet und vor den bisherigen Primärdaten in die Tabelle eingefügt. Die Zugriffspfade werden nur für den neu eingefügten Bereich aktualisiert.

Die Pflege der Indizes kann unterdrückt werden (Parameter NO INDEX, Voreinstellung); alle Indizes der Tabelle werden dann als „ungültig“ markiert. Bei Angabe des Parameters GENERATE INDEX werden alle Indizes neu aufgebaut, nachdem alle Ladesätze in die Primärdaten der Tabelle eingemischt sind. Im Prinzip läuft intern die Anweisung RECOVER INDEX ab; in diesem Zeitraum sind die Indexspaces exklusiv gesperrt.

Die Prüfung von Integritätsbedingungen kann unterdrückt werden (Parameter NO CONSTRAINT CHECK, Voreinstellung). Die Tabelle wird dann in den Zustand „check pending“ versetzt. Bei Angabe des Parameters CONSTRAINT CHECK werden alle Integritätsbedingungen geprüft, nachdem alle Ladesätze in die Primärdaten der Tabelle eingearbeitet sind. Im Prinzip läuft intern die Anweisung CHECK CONSTRAINTS ab.

Wenn sich der Anwender-Space, auf dem die Tabelle liegt, im Logging befindet, wird das Logging unterbrochen. Der Space ist danach im Zustand „copy pending“.

Wird die Anweisung LOAD OFFLINE während der Verarbeitung in der Servicetask abgebrochen, z.B. wegen eines Betriebsmittelengpasses, so ist die Tabelle inkonsistent und erhält den Zustand „load running“. Auf sonstige Tabellen und Indizes auf dem gleichen Space kann danach nur lesend zugegriffen werden. Der Anwenderspace muss dadurch repariert werden, dass er mit einer RECOVER-Anweisung auf einen Stand vor dem LOAD zurückgesetzt wird.

LOAD ONLINE

LOAD ONLINE sperrt die aktuelle Tabelle und ihre Indizes nur shared und verhält sich hier wie eine DML-Änderungsanweisung. Auf der betroffenen Tabelle ist paralleles Lesen und Ändern erlaubt. Die Datenzugriffe finden vollständig in einer DBH-Task statt.

Die Sätze werden einzeln in der Reihenfolge ihres Auftretens in der Ladedatei in die Tabelle eingefügt. Die Zugriffspfade werden pro Satz aktualisiert.

Die Pflege der Indizes kann nicht unterdrückt werden. Die Indizes der Tabelle werden jeweils für die eingefügten und geänderten Sätze aktualisiert. Ungültige Indizes werden ignoriert.

Die Prüfung von Integritätsbedingungen kann unterdrückt werden (Parameter NO CONSTRAINT CHECK). Die Tabelle wird dann in den Zustand „check pending“ versetzt. Ohne diese Parameterangabe werden die Integritätsbedingungen für die eingefügten und geänderten Sätze geprüft. Nur wenn die Tabelle bereits im Zustand „check pending“ war, werden alle Integritätsbedingungen für alle Sätze geprüft.

Das Logging wird nicht unterbrochen.

Auch nach einem Abbruch während der Verarbeitung, z.B. wegen eines Betriebsmittelengpasses, ist die Tabelle formal korrekt aufgebaut. Sie kann aber den Zustand „check pending“ annehmen, falls Integritätsbedingungen nicht vollständig geprüft werden konnten.

8.1.2 Performance-relevante Aspekte von LOAD

Aus der im letzten Abschnitt beschriebenen, unterschiedlichen Bearbeitung der LOAD-Anweisungen ergeben sich Folgerungen für die Performance einer LOAD-Anweisung.

LOAD OFFLINE

Bei einer Tabelle mit Primärschlüssel hängt die Performance von LOAD OFFLINE stark von der Größe der Tabelle (nach dem LOAD) und nur in relativ geringem Maße von der Menge der Ladesätze ab.
Eine hohe Grundlast (Einlesen der gesamten bereits existierenden Primärdaten, Neuaufbau der Zugriffspfade, ggf. Neuaufbau der Indizes und Prüfung der Integritätsbedingungen) liegt immer vor, auch wenn nur ein Satz geladen werden soll.

Bei einer Tabelle ohne Primärschlüssel ist die Performance von LOAD OFFLINE nicht ganz so stark von der Größe der Tabelle abhängig, da das Einlesen der existierenden Primärdaten und der vollständige Neuaufbau der Zugriffspfade entfallen. Falls Indizes oder Integritätsbedingungen existieren, liegt aber auch hier eine gewisse Grundlast vor.

In zwei weiteren Fällen, die aus Übersichtlichkeitsgründen im vorigen Abschnitt nicht beschrieben wurden, nimmt SESAM/SQL auch bei Tabellen mit Primärschlüssel im Rahmen von LOAD OFFLINE eine Optimierung vor:

– wenn alle Sätze der Ladedatei einen höheren Primärschlüsselwert haben als der bisher höchste Primärschlüsselwert der Tabelle

– wenn der erste Teil eines zusammengesetzten Primärschlüssels als Zählfeld verwendet wird

In diesen Fällen werden alle einzumischenden Sätze an die bisherigen Primärdaten angefügt. Es unterbleibt ebenfalls das Einlesen der existierenden Primärdaten und der vollständige Neuaufbau der Zugriffspfade.

LOAD ONLINE

LOAD ONLINE verhält sich weitgehend wie eine DML-Änderungsanweisung. Die Performance ist im Wesentlichen von der Anzahl der geladenen Sätze abhängig. Es gibt bei LOAD ONLINE keine nennenswerte Grundlast.
Die Bearbeitungszeit pro Satz ist bei LOAD ONLINE länger als bei LOAD OFFLINE, da die Zugriffspfade und die Indizes satzweise gepflegt werden und auch ein gewisser Aufwand für die Synchronisation und das Logging anfällt.

8.1.3 Einsatzempfehlungen für LOAD

Dieser Abschnitt gibt Hinweise zur Verbesserung der Performance von LOAD. Insbesondere finden Sie hier Entscheidungshilfen, in welchen Fällen LOAD OFFLINE bzw. LOAD ONLINE zum Erzielen einer optimalen Performance vorzuziehen ist.

LOAD OFFLINE

LOAD OFFLINE ist beim Laden in eine leere Tabelle performanter als LOAD ONLINE.

Bei LOAD OFFLINE sollte vor allem bei großen Datenmengen immer die Ladedatei nach Primärschlüsselwert sortiert und die LOAD-Anweisung mit Parameter SORTED aufgerufen werden. Abgesehen von der schnelleren Verarbeitung wird auch keine Arbeitsdatei angelegt, die durchaus größer sein kann als die Ladedatei selbst.

Werden mehrere Anweisungen LOAD OFFLINE hintereinander durchgeführt, so sollten die Parameter GENERATE INDEX und CONSTRAINT CHECK nur beim letzten Aufruf von LOAD OFFLINE angegeben werden. Dies spart überflüssige und aufwändige Aktionen.

LOAD OFFLINE unterbricht das Logging für den Anwenderspace, auf dem sich die Tabelle befindet, und für die zugehörigen Indexspaces. Für diese Spaces muss anschließend mit COPY eine SESAM-Sicherung oder eine Fremdkopie erstellt werden. Der Zeitaufwand für das Kopieren und der Speicherplatz für die Sicherungsdateien kann durch die Verwendung von LOAD ONLINE eingespart werden. Zu beachten ist aber, dass bei LOAD ONLINE für die Logging-Informationen ein gewisser zusätzlicher Speicherplatz benötigt wird.

LOAD ONLINE

LOAD ONLINE ist beim Laden von relativ wenigen Sätzen performanter als LOAD OFFLINE. Dies trifft besonders bei einer Tabelle mit Primärschlüssel und vielen Indizes zu. „Relativ wenig“ heißt hier, dass die Anzahl der Sätze in der Ladedatei im Verhältnis zu den bereits in der Tabelle existierenden Sätzen gering ist.
Konkrete Zahlen können nicht genannt werden, da hier die Menge der Integritätsbedingungen und vor allem die Menge der Indizes eine große Rolle spielt. Als grobe Faustregel gilt, dass LOAD ONLINE dann schneller ist, wenn die Anzahl der Sätze in der Ladedatei weniger als 10 Prozent der Anzahl der Sätze in der Tabelle beträgt.

LOAD ONLINE nutzt eine Sortierung der Ladedatei nicht explizit aus. Dennoch kann dadurch ein besseres Pufferverhalten und somit eine gewisse Performancesteigerung erzielt werden. Enthält z.B. die Ladedatei viele Sätze, deren Primärschlüsselwerte im gleichen Wertebereich liegen oder sogar benachbart sind, dann kann eine Sortierung deutliche Vorteile bringen.

Bei LOAD ONLINE können die Integritätsbedingungen meistens effektiver geprüft werden als bei LOAD OFFLINE. Daher kann durch die Angabe von NO CONSTRAINT CHECK bei LOAD ONLINE im Allgemeinen auch nicht viel eingespart werden. Ferner ist zu bedenken,

dass eine später erneut durchzuführende Prüfung (im Rahmen von LOAD oder von CHECK CONSTRAINTS) dann wieder die gesamte Tabelle auf ihre Integrität hin prüfen muss. Falls aber absehbar ist, dass die Tabelle durch LOAD ONLINE sowieso in den Zustand „check pending“ geraten wird, z.B. wenn nicht alle Spalten auf einmal zugeladen werden können, ist die Unterdrückung der Prüfungen natürlich sinnvoll.

Nach Abbruch der Anweisung im Fehlerfall bleibt die Tabelle bei LOAD ONLINE immer formal korrekt aufgebaut, während sie durch LOAD OFFLINE in den Zustand „load running“ versetzt werden kann. In letzterem Fall ist das Reparieren des Table-Spaces durch RECOVER erforderlich, was sich bei Nutzung von LOAD ONLINE also vermeiden lässt. Bei einer Tabelle mit Primärschlüssel kann der abgebrochene LOAD ONLINE im Allgemeinen mit Parameter OVERWRITE wiederholt werden. Nötigenfalls sind vor der Wiederholung die bereits zugeladenen Sätze aus der Ladedatei zu entfernen.

8.1.4 Fortsetzen einer abgebrochenen LOAD-Anweisung

Sie können mit dem Parameter SKIP FIRST *n* RECORDS eine abgebrochene LOAD-Anweisung auf komfortable und performante Weise fortsetzen.

Dazu wird nach einer abgebrochenen LOAD-Anweisung in der Fehlerdatei vermerkt, wie viele Sätze der Eingabedatei bis zum Abbruchzeitpunkt gelesen und bereits in die Tabelle eingetragen wurden (Eintrag „*n* RECORDS PROCESSED“).

Mit dem Parameter SKIP FIRST *n* RECORDS erreichen Sie in der nachfolgenden LOAD-Anweisung, dass die ersten *n* Sätze der Eingabedatei überlesen werden.

Eine abgebrochene LOAD-Anweisung (insbesondere LOAD ONLINE) kann z.B. dann sinnvoll fortgesetzt werden, wenn der Abbruch durch einen Betriebsmittelengpass verursacht wurde und dieser Engpass mittlerweile beseitigt ist.

LOAD ONLINE

Nach dem vorzeitigen Abbruch einer LOAD ONLINE-Anweisung haben Sie folgende Möglichkeiten der Fortsetzung:

- Die LOAD-Anweisung wird mit dem zusätzlichen Parameter SKIP FIRST *n* RECORDS wiederholt, wobei der Wert *n* direkt der Fehlerdatei entnommen werden kann.

  Diese Variante ist die komfortabelste und performanteste Fortsetzung einer abgebrochenen LOAD ONLINE-Anweisung.

- Die LOAD-Anweisung kann erneut mit der vorliegenden Eingabedatei durchgeführt werden, wenn gewährleistet ist, dass eine Wiederholung zum gleichen Ergebnis führt wie beim ersten Mal.
  Voraussetzung ist also, dass Sätze, die bei der ersten LOAD-Anweisung neu angelegt wurden, bei der Wiederholung der LOAD-Anweisung überschrieben werden. Die Basistabelle muss demnach einen Primärschlüssel haben und bei der ersten LOAD-Anweisung darf kein Zählfeld angegeben worden sein. Ferner ist bei der Verwendung multipler Felder darauf zu achten, dass ein mit 1 beginnender lückenloser Ausprägungsbereich vorliegt und nur signifikante Werte in die multiplen Felder geladen werden.

- Vor dem Wiederholen der LOAD-Anweisung werden die ersten, bereits verarbeiteten *n* Sätze der Eingabedatei gelöscht.
  Die Anzahl der bereits verarbeiteten Sätze kann dem Eintrag „*n* RECORDS PROCESSED“ der Fehlerdatei entnommen werden. Allerdings ist die Bearbeitung von Dateien (speziell bei Sätzen einer Länge von mehr als 256 Zeichen) zeitaufwändig.

Eine abgebrochene LOAD ONLINE-Anweisung kann auch durch eine LOAD OFFLINE-Anweisung mit dem Parameter SKIP FIRST *n* RECORDS fortgesetzt werden.

LOAD OFFLINE

Bei LOAD OFFLINE wird das Laden der Anwenderdaten entweder ganz oder gar nicht durchgeführt.

Auch hier kann dem Eintrag „*n* RECORDS PROCESSED“ der Fehlerdatei entnommen werden, wie viele Sätze bereits in die Tabelle eingetragen wurden.

Der Eintrag „0 RECORDS PROCESSED“ kann folgende Ursachen haben:

- Die LOAD-Anweisung wurde abgebrochen bevor die Tabelle selbst modifiziert wurde. In diesem Fall kann die LOAD OFFLINE-Anweisung einfach wiederholt werden.
- Die LOAD-Anweisung wurde abgebrochen nachdem die Tabelle bereits modifiziert wurde. In diesem Fall befindet sich der Anwender-Space im Zustand „load running“ und muss mit einer RECOVER-Anweisung auf einen Stand vor der LOAD-Anweisung zurückgesetzt werden. Danach kann dann die LOAD OFFLINE-Anweisung wiederholt werden.

Wurden laut Fehlerdatei bereits Sätze bearbeitet (n>0), dann wurde das Laden der Primärdatensätze erfolgreich abgeschlossen.
Der Abbruch erfolgte demnach beim Erstellen der Sekundärindizes
(Parameter GENERATE INDEX) oder bei der Prüfung der Integritätsbedingungen
(Parameter CONSTRAINT CHECK).

Im Prinzip kann auch hier die LOAD-Anweisung mit Parameter SKIP FIRST *n* RECORDS wiederholt werden. Da in diesem Fall aber (unnötigerweise) die gesamte Eingabedatei noch einmal gelesen wird, ist es performanter, explizit die Anweisungen RECOVER INDEX ON TABLE bzw. CHECK CONSTRAINTS ON TABLE auszuführen.

8.2 UNLOAD - Anwenderdaten aus einer Tabelle ausgeben

Neben der Utility-Anweisung UNLOAD OFFLINE gibt es die Utility-Anweisung UNLOAD ONLINE.

Diese beiden Utility-Anweisungen unterscheiden sich voneinander

- durch ihr Sperrverhalten, also in der Art der Synchronisation mit anderen Anweisungen, die auf die gleichen Spaces, Tabellen und Indizes zugreifen
- durch ihren internen Ablauf
- durch ihren Funktionsumfang

Daraus ergibt sich in Abhängigkeit von bestimmten Randbedingungen auch ein unterschiedliches Verhalten bezüglich der Performance. Die folgenden Abschnitte liefern Anhaltspunkte, in welchen Situationen die Klauseln OFFLINE bzw. ONLINE gewählt werden sollten.

UNLOAD OFFLINE

Wie die meisten Utility-Anweisungen sperrt UNLOAD OFFLINE die betroffene Tabelle exklusiv. Indizes werden nicht ausgewertet und werden daher nicht gesperrt. Auf sonstige Tabellen und Indizes auf den betroffenen Spaces können andere Anwender parallel nur lesend zugreifen.

Die Datenzugriffe finden zum größten Teil in einer Service-Task statt.

Es können nur Basistabellen angesprochen werden.

UNLOAD ONLINE

UNLOAD ONLINE sperrt die betroffenen Sätze und Indexwerte nur shared und verhält sich hier wie eine lesende DML-Anweisung. Auf der betroffenen Tabelle ist daher paralleles Lesen und Ändern möglich.

Die Datenzugriffe finden vollständig in einer DBH-Task statt.

Sowohl Basistabellen als auch Views können angesprochen werden. Die Treffermenge kann sortiert und mit einer Suchbedingung eingeschränkt werden.

Performance-relevante Aspekte von UNLOAD

UNLOAD ONLINE verhält sich ähnlich einer Datenbankabfrage per DML. Der Unterschied besteht darin, dass bei UNLOAD ONLINE die Ergebnisse nicht in Benutzervariablen, sondern in einer Ausgabedatei abgelegt werden. Insbesondere wenn der Hauptaufwand in der Ermittlung der Treffermenge liegt, etwa bei komplexen WHERE-Klauseln oder Views, dann ist die Performance von UNLOAD ONLINE mit der Performance der entsprechenden Datenbankabfrage vergleichbar.

UNLOAD OFFLINE sperrt die Tabelle exklusiv, so dass der Synchronisationsaufwand entfällt. Es wird die gesamte Basistabelle gelesen. Sie wird dann grundsätzlich sequenziell bearbeitet, wodurch sehr effiziente Algorithmen genutzt werden können. UNLOAD OFFLINE ist daher wesentlich performanter als UNLOAD ONLINE.

Einsatzempfehlungen für UNLOAD

Wenn eine vollständige Basistabelle ausgegeben werden soll und das restriktive Sperrverhalten vertretbar ist, dann sollte UNLOAD OFFLINE verwendet werden.

UNLOAD ONLINE kann dagegen vielseitiger eingesetzt werden (siehe WHERE-Klausel, ORDER-BY-Klausel, Views) und behindert parallele Anwendungen höchstens durch Lesesperren auf einzelnen Sätzen.

Die Ausgabeformate bei UNLOAD ONLINE und UNLOAD OFFLINE sind identisch. Wenn beim Ausgeben sehr großer Basistabellen das Layout der Ausgabedatei zuerst getestet werden soll (insbesondere bei einem selbst-definierten Format), dann können UNLOAD ONLINE und UNLOAD OFFLINE gut kombiniert werden. Eine sinnvolle Vorgehensweise besteht z.B. darin, zunächst verschiedene Varianten mit UNLOAD ONLINE (mit einer kleinen Treffermenge bei entsprechender WHERE-Klausel) auszuprobieren. Für das endgültige Erzeugen der vollständigen Ausgabedatei kann dann die performantere Variante UNLOAD OFFLINE eingesetzt werden.

8.3 SESAM-Sicherungsbestand (COPY) und Fremdkopie erstellen

Der Benutzer kann Sicherungen seiner Datenbank mit folgenden Mitteln erzeugen:

- mit der Utility-Anweisung COPY
- mit dem BS2000-Kommando COPY-FILE
- mit den Replikations-Funktionen der Plattenspeicher-Systeme

Mit der Utility-Anweisung COPY können Sie Sicherungskopien auf Platte anlegen. Sie können auch mit Hilfe von ARCHIVE oder HSMS eine Langzeitarchivierung durchführen. Langzeitarchivierung bezeichnet die langfristige Auslagerung auf Magnetbandkassetten oder sonstigen Speichersystemen (z.B. CentricStor).

Weiter können Sie angeben, ob die Sicherung ONLINE oder OFFLINE erfolgen soll. Bei COPY OFFLINE ist die zu sichernde Datenbank gegen ändernde Zugriffe während der Dauer der Sicherung gesperrt.
Während COPY ONLINE sind nach einer Synchronisationsphase sowohl lesende als auch ändernde DML-Anweisungen auf die Datenbank erlaubt.
Eine ONLINE-Sicherung repräsentiert immer den Stand zu Beginn der Sicherung. Alle während der Dauer der Sicherung durchgeführten Änderungen sind nicht in der Sicherung enthalten, sondern nur in der Datenbank.

Im Folgenden werden die performance-relevanten Aspekte der Utility-Anweisung COPY ONLINE beschrieben. Diese Aspekte gelten mit Ausnahme der Aussagen zur PBI-Datei bzw. zur HSMS-Arbeitsdatei auch für COPY OFFLINE.

8.3.1 Bearbeitung der COPY-Anweisung in SESAM/SQL

Bei allen SESAM-Sicherungsverfahren werden die Dateinamen der Spaces nach ihrer Dateigröße beginnend mit dem größten Space sortiert. In dieser Reihenfolge werden die Spacenamen an die Komponente übergeben, die die Sicherung durchführt.

Durch dieses Verfahren dauert die Sicherung im günstigsten Fall nur so lange wie zur Sicherung des größten Spaces benötigt wird.

8.3.2 Sicherung auf Platte

Bei der Sicherung auf Platte werden bis zu zehn Spaces im „Wrap Around“-Verfahren gleichzeitig gesichert. Jedem Space ist ein 960 KB großer Wechselpuffer zugeordnet. In diesen Wechselpuffer wird asynchron in Einheiten von bis zu 480 KB aus dem Original gelesen und in die Sicherungskopie geschrieben. Die Größe der Schreib-/Lese-Einheit ist abhängig vom Plattentyp. SESAM/SQL nutzt die maximal mögliche I/O-Länge für jeden zu kopierenden Space. Die I/O-Länge liegt im Bereich 32 bis 480 KB.
Information über die maximale I/O-Länge in Half Pages (eine Half Page entspricht 2 KB) erhalten Sie mit folgendem BS2000-Kommando:

```
/SHOW-PUBSET-CONFIGURATION PUBSET=<catid>,INFORMATION=*PUBSET-FEATURES)
```

Anschließend wird pro Space auf das Ende der Schreiboperation „gewartet“. Wenn die Datenbankdateien auf verschiedene Platten verteilt sind und die Platten über unterschiedliche Kanäle angeschlossen sind, werden Wartesituationen minimiert. Wenn die Sicherung für einen Space beendet ist, wird sofort der nächste Space aus der Liste bearbeitet. Im besten Fall werden immer zehn Spaces gleichzeitig gesichert.

Werden während der Sicherung ändernde DML-Anweisungen auf die Datenbank durchgeführt, so werden die physikalischen Before Images von geänderten Datenbankblöcken in eine PBI-Datei geschrieben. Diese PBI-Datei wird am Ende der Sicherung in die Sicherungskopie eingebracht.

8.3.3 Sicherung mit ARCHIVE

Um Spaces parallel zu sichern, benötigen Sie redundante Peripherie. Im besten Fall steht pro Space ein Bandgerät zur Verfügung. Die Dauer der Sicherung wird dann vom größten Space bestimmt. SESAM führt eine Paketierung durch, falls in der ARCHIVE-Parameterdatei der Parameter DRIVE, der die Anzahl der Parallelläufe vorgibt, einen Wert größer als 1 besitzt.

Im ungünstigsten Fall steht nur ein Bandgerät zur Verfügung. Die Spaces werden dann sequenziell gesichert werden.

Werden während der Sicherung ändernde DML-Anweisungen auf die Datenbank durchgeführt, so werden die physikalischen Before Images von geänderten Datenbankblöcken in eine PBI-Datei geschrieben. Diese PBI-Datei wird nach der Sicherung der Spaces ebenfalls auf Band gesichert. Erst wenn die Sicherung wieder eingespielt wird, wird auch die PBI-Datei eingespielt und die PBI-Datei in die Sicherungskopie eingebracht.

Zusätzlich zur Sicherung der Spaces kommt also hier noch die Zeit für die Sicherung der PBI-Datei hinzu.

8.3.4 Sicherung mit HSMS

Bei der Sicherung mit HSMS gibt es zwei verschiedene Verfahren: Sicherung mit Arbeitsdatei in ein HSMS-Archiv oder Sicherung von der Spiegelplatte eines lokalen oder entfernten Plattenspeichersystems in ein HSMS-Archiv.

Sicherung mit Arbeitsdatei in ein HSMS-Archiv

Bei der Sicherung mit Arbeitsdatei nutzt SESAM/SQL die Funktion „Concurrent Copy“ (CCOPY) von HSMS. Werden während der Sicherung ändernde DML-Anweisungen auf die Datenbank durchgeführt, so schreibt HSMS die Before Images von geänderten Datenbankblöcken in eine Arbeitsdatei. Beim Sichern der Datenbankdateien in das HSMS-Archiv liest HSMS den zu sichernden Block aus der Datenbankdatei oder aus der Arbeitsdatei. Bei diesem Verfahren entfällt das SESAM-seitige Schreiben der PBI-Datei und die Sicherung der PBI-Datei auf Band. Ebenso entfällt auch beim Einspielen der Sicherung das Einspielen der PBI-Datei und das Einbringen der PBI-Datei in die Sicherungskopie.

Sicherung von Spiegelplatten in ein HSMS-Archiv

Bei der Sicherung in ein HSMS-Archiv von einer Spiegelplatte nutzt SESAM/SQL implizit über HSMS die Funktion TimeFinder/Mirror der Symmetrix-Systeme. Nachdem die Spiegelplatte abgesplittet ist, werden die Datenbankdateien von der Spiegelplatte in das HSMS-Archiv gesichert. Anschließend wird die Spiegelplatte wieder mit der Originalplatte synchronisiert.

8.3.5 Sicherung mit Fremdkopie

Der Anwender kann seine Datenbank auch unabhängig von SESAM/SQL sichern. Mit Hilfe von Administrationsanweisungen (LOCK-SEQUENCE, PREPARE-FOREIGN-COPY) muss ein transaktionsfreier Zustand auf der Datenbank erzeugt werden, und die veränderten Datenbankblöcke müssen auf Platte geschrieben werden. Dann kann vom Anwender eine Kopie der Datenbank erzeugt werden.

Falls die Datenbank auf einer Spiegelplatte liegt, kann, nachdem ein transaktionsfreier Zustand erzeugt und die Datenbankblöcke auf Platte geschrieben wurden, die Spiegelplatte abgesplittet werden. Anschließend kann auf die Datenbank auf der Originalplatte wieder zugegriffen werden. Die Datenbank auf der Spiegelplatte kann nun mit SESAM-Mitteln formal geprüft (CHECK FORMAL) und als SESAM-Sicherung oder als Fremdkopie gesichert werden. Nach der Sicherung wird die Spiegelplatte wieder mit der Originalplatte synchronisiert, siehe „Basishandbuch“, Abschnitt „Erzeugen einer Fremdkopie“.

Der Nachteil bei der Erzeugung einer Fremdkopie mit dem BS2000-Kommando COPY-FILE oder von einer Spiegelplatte ist, dass während der Dauer der Kopie kein paralleler (weder lesender noch ändernder) DML-Zugriff möglich ist. Die Fremdkopie verhält sich hier ähnlich wie die Utility-Anweisung COPY OFFLINE. Die Vorteile der Fremdkopie liegen bei der Reparatur (siehe Abschnitt „Verkürzung der Einspielzeit“ auf Seite 220).

Hinweis zu PREPARE-FOREIGN-COPY

Die Administrationsanweisung PREPARE-FOREIGN-COPY ..., CLOSE=*NO schließt die Datenbank logisch, aber nicht physikalisch. Sie ist für Sicherungsverfahren geeignet, die eine logisch geschlossene Datenbank voraussetzen, z.B. die Replikationsfunktion TimeFinder/Mirror.

Für Sicherungsverfahren, die eine physikalisch geschlossene Datenbank voraussetzen, wie z.B. die Replikationsfunktion TimeFinder/Snap oder das BS2000-Kommando COPY-FILE kann die Administrationsanweisung PREPARE-FOREIGN-COPY ..., CLOSE=*YES verwendet werden.
Die Datenbank wird dann auch physikalisch geschlossen.

8.3.6 Auswahlkriterien für Sicherungsverfahren

Die Sicherung auf Platte benötigt Speicherplatz in derselben Größenordnung wie die Datenbank. Die Dauer der Sicherung hängt ab von der Verteilung der Dateien auf den Platten und der verwendeten Peripherie.

Bei einer Sicherung mit ARCHIVE oder HSMS wird nur Speicherplatz im verwendeten Langzeitarchiv benötigt. Die Dauer der Sicherung hängt sehr stark von den verwendeten Archivierungsmedien und dem Parallelitätsgrad ab. Bei ARCHIVE muss zusätzlich zu den Dateien der Datenbank noch die PBI-Datei gesichert werden.

Die eigentliche Sicherung der Datenbank findet in einer Service-Task statt. Durch die Sicherung selbst wird also der DBH nicht belastet. Das Schreiben in die PBI-Datei (bei Plattensicherung und Sicherung mit ARCHIVE) findet im DBH statt.

Die SESAM Sicherung in einem HSMS-Archiv kann auch mit HSMS-Mitteln eingespielt werden. Sie repräsentiert einen konsistenten Stand und kann als Fremdkopie genutzt werden. Bei der Sicherung mit ARCHIVE ist das nicht möglich.

Bei der Sicherung von der Spiegelplatte in ein HSMS-Archiv repräsentiert die Spiegelplatte nach dem Absplitten immer den Stand der Datenbank zum Zeitpunkt des Absplittens. Die Sicherung in ein HSMS-Archiv kann beliebig oft wiederholt werden, falls z.B. eine Sicherung fehlerhaft beendet wird. Dieses Verfahren ermöglicht es, eine „revisionssichere“ Sicherung zu erzeugen, d.h. die Sicherung repräsentiert immer den Zeitpunkt, zu dem die COPY-Anweisung gestartet wurde.
Kunden, für die eine „revisionsichere“ Sicherung zwingend erforderlich ist, können damit die Anweisung COPY ONLINE verwenden.

Bei der Sicherung einer Datenbank mit dem BS2000-Kommando COPY-FILE oder mit Hilfe von Replikationsfunktionen des Plattenspeichersystems, also ohne SESAM-Mittel, ist Folgendes zu beachten:

- Der Anwender ist selbst für die Erzeugung einer konsistenten Sicherung verantwortlich.
- Während der Dauer der Kopieerstellung ist kein paralleler Zugriff möglich.

Die Sicherungen müssen vom Anwender verwaltet werden. Sie müssen zur Recovery vom Anwender eingespielt werden.

Die Fremdkopie kann durch Umbennen für eine schnelle Reparatur verwendet werden.

8.4 RECOVER - Reparieren und Zurücksetzen

Beim Ablauf der Utility-Anweisung RECOVER werden in der Regel folgende Verarbeitungsschritte durchgeführt:

- Einspielen der Sicherung der zu reparierenden Spaces
- Einbringen der protokollierten Änderungen in die Spaces (Nachfahren der CAT-LOG- bzw. DA-LOG-Dateien)
- Eventuell Sekundärindizes neu aufbauen

Entsprechend hängt der Zeitbedarf für die Reparatur von folgenden Faktoren ab:

- Größe der einzuspielenden Sicherungsdateien
- Anzahl der Änderungen (inklusive Einfügungen und Löschungen), die seit der Sicherung durchgeführt wurden
- Größe und Anzahl der Tabellen, für die Sekundärindizes neu aufgebaut werden müssen

Anhand der aufgeführten Verarbeitungsschritte soll in den folgenden Abschnitten gezeigt werden, mit welchen Maßnahmen die Dauer der einzelnen Verarbeitungsschritte verkürzt werden kann. Bei den beschriebenen Maßnahmen wird immer von einer Reparatur der Datenbank oder einzelner Spaces ausgegangen. Die Aussagen gelten im Wesentlichen auch für das Zurücksetzen, wobei nicht immer alle Schritte durchgeführt werden müssen.

8.4.1 Verkürzung der Einspielzeit

Beim Reparieren werden zuerst die Sicherungen auf die zu reparierenden Spaces kopiert. Dieses Verfahren wird bei SESAM-Sicherungen immer angewendet. Bei Replikaten kann der Datenbankverwalter festlegen, ob die Dateien kopiert oder umbenannt werden sollen (Parameter COPY oder RENAME). Bei Fremdkopien ist der Datenbankverwalter für das Einspielen verantwortlich, d.h. er kann die Dateien kopieren oder umbenennen. Eine deutliche Verkürzung der Einspielzeit lässt sich erreichen, wenn die Dateien umbenannt werden.

RECOVER mit Replikat und Parameter RENAME

Bevor diese RECOVER-Anweisung eingegeben wird, müssen die zu reparierenden Spaces vom Datenbankverwalter gelöscht werden.

Während der Verarbeitung der Utility-Anweisung RECOVER … USING REPLICATION … RENAME werden die angegebenen Spaces (oder der gesamte Catalog) des Replikats auf die Dateien der Datenbank umbenannt. Anschließend werden die in den Logging-Dateien protokollierten Änderungen in die Spaces eingebracht.

Die Dauer der Einspielzeit ist reduziert auf die Zeit, die benötigt wird, um die Spaces zu löschen und die Spaces des Replikats umzubenennen. Dabei ist zu beachten, dass beim Löschen ein Space mit Binär Null überschrieben wird, falls der Space in der CREATE SPACE-Anweisung mit dem Parameter DESTROY angelegt wurde. Es empfiehlt sich, die Spaces mit dem Parameter NO DESTROY anzulegen.

Da die Dateien umkatalogisiert wurden, liegen die reparierten Spaces jetzt auf den Medien des Replikats.

Die angegebenen Spaces des Replikats sind nach dieser Anweisung nicht mehr vorhanden. Das Replikat muss neu erzeugt werden bzw. die Spaces müssen wieder in das Replikat aufgenommen werden.

RECOVER mit einer Fremdkopie auf Platte

Eine ähnliche Vorgehensweise wie beim Replikat beschrieben, lässt sich mit Hilfe einer Fremdkopie realisieren. Der Datenbankverwalter löscht die zu reparierenden Spaces und benennt die entsprechenden Spaces der Fremdkopie auf die Spaces der Datenbank um. Die anschließende Utility-Anweisung RECOVER … USING FOREIGN COPY verwendet die bereitgestellte Sicherung als Basis für die Reparatur.

Da die Dateien umkatalogisiert wurden, liegen die reparierten Spaces jetzt auf den Medien der Fremdkopie.

Die angegebenen Spaces der Fremdkopie sind nach dieser Anweisung nicht mehr vorhanden. Dieser Stand der Fremdkopie kann auch nicht neu erzeugt werden.

Daher empfiehlt es sich bei der Nutzung dieses Verfahrens, in regelmäßigen Abständen eine Fremdkopie der Datenbank zu erzeugen und diese anschließend in ein Langzeitarchiv zu sichern. Die jeweils letzte Fremdkopie bleibt auf einer Platte bis zur nächsten Sicherung erhalten.

RECOVER mit einer Fremdkopie von einer Spiegelplatte

Wenn Sie Plattenspeichersysteme mit Replikationsfunktionen verwenden, dann können Sie die Replikationsfunktionen dieser Plattenspeichersysteme zur Erstellung von Fremdkopien unter SESAM/SQL nutzen.
Die Bedienung der Replikationsfunktionen erfolgt über die Host-Komponente SHC-OSD, siehe Handbuch „Storage Management für BS2000“.

Im Folgenden wird als Beispiel ein Verfahren mit Hilfe von zwei Spiegelplatten (Mirror-Units) beschrieben. Hier wird nur auf die Besonderheiten dieses Verfahrens eingegangen. Informationen zu den notwendigen Serialisierungen und Vorbereitungen finden Sie im Abschnitt „Erzeugen einer Fremdkopie“ im „Basishandbuch“.

Mit Hilfe von zwei Spiegelplatten lässt sich folgende Konfiguration aufbauen.
Dabei wird vorausgesetzt, dass die Datenbankdateien auf der Original-Unit liegen. Die Logging-Dateien (CAT-LOG- und DA-LOG-Dateien) und die CAT-REC-Datei dürfen nicht mit dieser Spiegelplatte gespiegelt werden.

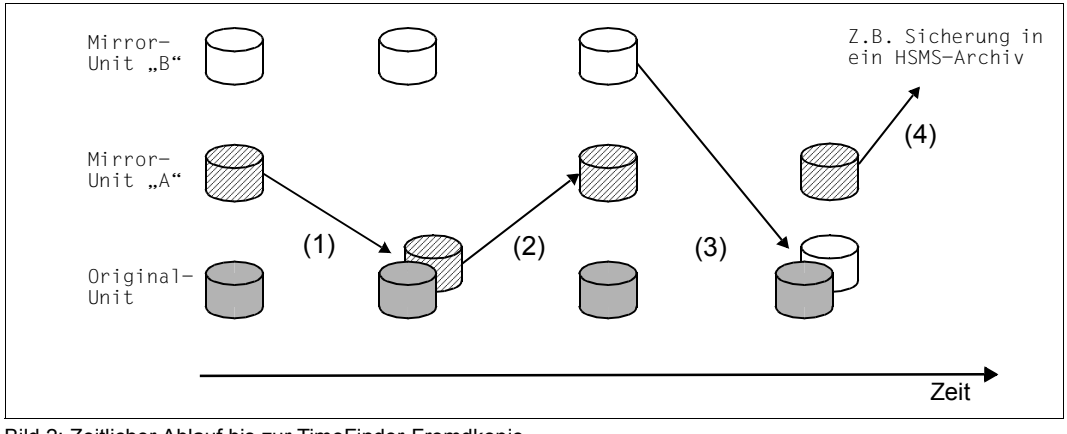


Bild 2: Zeitlicher Ablauf bis zur TimeFinder-Fremdkopie

(1) Der Original-Unit, auf der die Datenbank liegt, wird zunächst die Mirror-Unit „A“ zugeschaltet. Die Änderungen der Datenbank werden ab diesem Zeitpunkt auf der Mirror-Unit „A“ gespiegelt.

(2) Zu einem späteren Zeitpunkt wird für die Sicherung die Mirror-Unit „A“ abgetrennt.

(3) Die Mirror-Unit „B“ wird für den weiteren Betrieb zugeschaltet. Die Änderungen der Datenbank werden ab diesem Zeitpunkt auf der Mirror-Unit „B“ gespiegelt.

(4) Die abgetrennte Mirror-Unit „A“ wird anschließend als eigenes Pubset importiert. Die Datenbank kann nun mit SESAM-Mitteln bearbeitet werden. Sie kann z.B. mit CHECK FORMAL auf formale Konsistenz geprüft werden und mit HSMS in ein Langzeitarchiv gesichert werden. Weiterhin steht die Datenbank auf der Mirror-Unit „A“ für eine schnelle Reparatur zur Verfügung.

Eine Reparatur auf Basis der letzten, auf der Mirror-Unit „A“ erstellten Sicherung ist nun auf folgende Weise möglich:

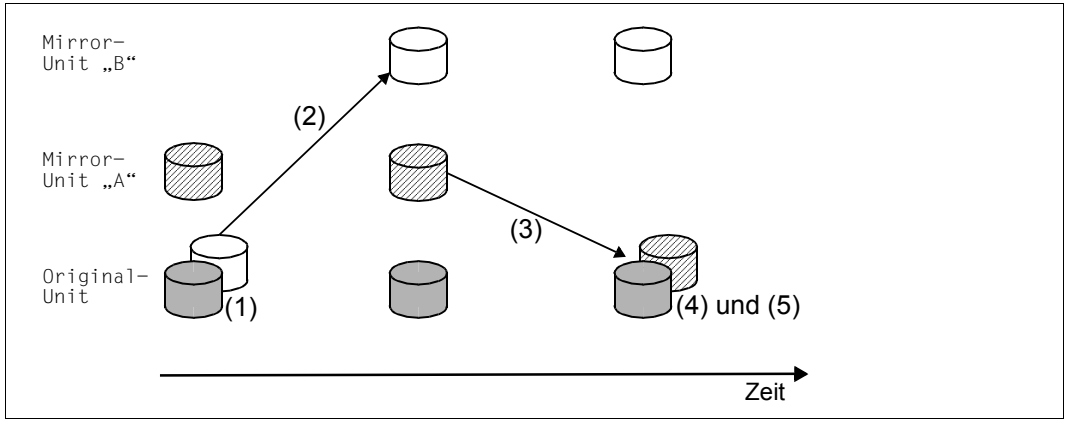


Bild 3: Zeitlicher Ablauf der schnellen Recovery

(1) Datenbank schließen mit der Administrationsanweisung
`SET-SQL-DB-CATALOG-STATUS STATUS=*FREE,SELECT=...`

(2) Abtrennen der aktiven Mirror-Unit „B“

(3) Mirror-Unit „A“ (mit der Sicherung) zuschalten

Die geänderten Daten der Mirror-Unit „A“ werden zur Original-Unit kopiert.
Die Änderungen der Datenbank werden ab diesem Zeitpunkt (wieder) auf der Mirror-Unit „A“ gespiegelt.

(4) Datenbank öffnen mit mit der Administrationsanweisung
`SET-SQL-DB-CATALOG-STATUS STATUS=*ACTIVE,SELECT=...`

(5) Reparieren der Datenbank und Einbringen der protokollierten Änderungen in die Datenbank mit der Utility-Anweisung
`RECOVER ... USING FOREIGN COPY`

RECOVER mit dem Parameter SCOPE PENDING

Die Dauer der Reparatur kann auch dadurch verkürzt werden, dass nur diejenigen Spaces repariert werden, die SESAM/SQL als defekt erkennt. Ein Space wird als defekt erkannt, wenn mindestens eine der folgenden Bedingungen zutrifft:

- Während einer DBH-Session trat ein Consistency Check mit Fehlergewicht 37 (Space defekt) oder 38 (Catalog-Space der Datenbank defekt) auf.
- Während einer DBH-Session trat ein Consistency Check für eine Basistabelle auf dem aktuellen Space mit Fehlergewicht 36 (Tabelle defekt) auf.
- Die Utility-Funktion CHECK FORMAL hat einen Fehler erkannt (CHECK FORMAL ohne Parameter NO ACTION).
- Der Space wurde durch eine abgebrochene Utility Anweisung (LOAD OFFLINE oder RECOVER) auf defekt gesetzt (Space-Zustand „load running“ oder „recover pending“).
- Der Space entspricht nicht dem aktuellen Stand der Metadaten im Catalog-Space.
- Beim Öffnen des Space tritt ein DMS-Fehler auf.

Anmerkung

Ein Space mit einem oder mehreren defekten Indizes ist in diesem Sinne nicht defekt.

Bei der Utility-Anweisung RECOVER CATALOG laufen folgende Verarbeitungsschritte ab:

1. Einspielen der Sicherung des Catalog-Space
2. Einbringen der protokollierten Änderungen in den Catalog-Space
3. Ermitteln und einspielen der Sicherungen der Spaces
4. Einbringen der protokollierten Änderungen in die Spaces
5. Eventuell Indizes neu aufbauen

Ist der Parameter SCOPE PENDING bei RECOVER CATALOG angegeben, so werden die Verarbeitungsschritte 1. und 2. immer durchgeführt. Im Verarbeitungsschritt 3. wird zusätzlich geprüft, welcher Space defekt ist. Nur für die als defekt erkannten Spaces werden die Verarbeitungsschritte 3. und 4. durchgeführt. Anschließend wird eventuell Verarbeitungsschritt 5. durchgeführt.

Bei RECOVER SPACE und RECOVER SPACESET mit oder ohne dem Parameter SCOPE PENDING entfallen die Verarbeitungsschritte 1. und 2. Die Verarbeitungsschritte 3. und 4. (und eventuell 5.) werden wie oben beschrieben ausgeführt.

8.4.2 Beschleunigung der Nachfahrzeiten

Alle Änderungen von Tabellen und Indizes eines Space werden in DA-LOG-Dateien protokolliert, sofern für diesen Space die logische Datensicherung eingeschaltet ist.
Eine DA-LOG-Datei ist in so genannte DA-LOG-Einheiten (DA-LOG-UNITS) unterteilt. Diese DA-LOG-Einheiten werden in der DA_LOGS-Tabelle verwaltet.

Ein Wechsel der DA-LOG-Einheit erfolgt bei folgenden Ereignissen:

- Wenn eine DA-LOG-Datei voll ist, wird implizit auf die nächste DA-LOG-Datei gewechselt.
- Mit der Administrationsanweisung CHANGE-DALOG wird die DA-LOG-Datei explizit gewechselt.
- Die DA-LOG-Einheit wird bei den Anweisungen CREATE SPACE und COPY SPACE implizit gewechselt.
- Die DA-LOG-Datei wird bei den Anweisungen COPY CATALOG, RECOVER CATALOG und RECOVER SPACE implizit gewechselt.
- Bei der ersten Änderung auf einem Space wird die DA-LOG-Einheit implizit gewechselt.
- Beim Wiederanlauf des DBH wird die DA-LOG-Einheit implizit gewechselt.

In der DA_LOGS-Tabelle sind pro DA-LOG-Einheit noch diejenigen Spaces vermerkt, für die in dieser DA-LOG-Einheit Änderungen protokolliert sind.

Wenn die protokollierten Änderungen in einen Space nachgefahren werden sollen, wird vorher aus der DA_LOGS-Tabelle ermittelt, welche DA-LOG-Einheiten dazu notwendig sind. Nur die benötigten DA-LOG-Einheiten werden beim Nachfahren gelesen.

Die Zeit für das Nachfahren von protokollierten Änderungen ist im Wesentlichen bestimmt durch die Anzahl Sätze, für die Änderungen protokolliert sind. Die Anzahl der zu ändernden Spalten in einem Satz spielt hier nur eine untergeordnete Rolle.

Paralleles Nachfahren einzelner Spaces

Mehrere RECOVER SPACE-Anweisungen können auf unterschiedliche Spaces der gleichen Datenbank gleichzeitig von verschiedenen Programmen eingegeben werden. Diese RECOVER SPACE-Anweisungen werden parallel im DBH verarbeitet. Dazu müssen die DA-LOG-Dateien auf Platte liegen, da bei Banddateien kein paralleler Zugriff möglich ist.

Die parallele Verarbeitung ist dann vorteilhaft, wenn die Änderungen für die nachzufahrenden Spaces in möglichst disjunkten DA-LOG-Einheiten protokolliert sind.

Ein weiterer Vorteil ist, dass für jeden nachzufahrenden Space eine Service-Task mit all ihren Ressourcen für diese Aufgabe zur Verfügung steht. Die SYSTEM-DATA- und USER-DATA-Buffer stehen jetzt einmal pro nachzufahrenden Space zur Verfügung. Auf diese Weise wird die Verdrängung von Datenbankblöcken aus dem Buffer deutlich reduziert.

Die parallelen RECOVER SPACE-Anweisungen sollten nur verwendet werden, wenn die Sicherungen der Spaces auf Platte liegen. Bei Sicherungen auf Magnetbandkassette kann durch entsprechende Parameter in der HSMS- bzw. ARCHIVE-Parameterdatei das parallele Einspielen der Dateien auf mehreren Bandgeräten gesteuert werden.

Wenn auf den Spaces Partitionen einer partitionierten Tabelle liegen, dann sollten diese Spaces in **einer** Anweisung RECOVER Space-Set / Space-Liste repariert werden, um mögliche Deadlock-Situationen zu vermeiden.

Nachfahren mit einem möglichst aktuellen Replikat

Mit den Funktionen CREATE REPLICATION und REFRESH REPLICATION bietet SESAM/SQL die Möglichkeit eine Schattendatenbank zu erzeugen. Diese Schattendatenbank kann relativ aktuell sein. Immer wenn auf eine neue Logging-Datei in der Original-Session gewechselt wird, kann die Vorgänger-Logging-Datei per REFRESH REPLICATION in das Replikat eingebracht werden. Bei der Reparatur mit der Anweisung RECOVER … USING REPLICATION RENAME müssen nur noch die auf der aktuellen Logging-Datei protokollierten Änderungen nachgefahren werden.

8.4.3 Reparatur von Index-Spaces

Datenbankverwalter können Ihre Datenbank so organisieren, dass es Spaces gibt, auf denen nur Indizes und keine Tabellen definiert sind. Diese so genannten Index-Spaces können schnell repariert werden. Index-Spaces können auch von der logischen Datensicherung ausgenommen werden, da die Indizes jederzeit aus den Daten der zugehörigen Tabellen neu aufgebaut werden können.

Wenn die Tabelle relativ wenige Sätze enthält und viele Änderungen durchgeführt werden, dann wird folgendes Vorgehen empfohlen:

- Der Index-Space wird von der logischen Datensicherung ausgenommen. Der Neuaufbau von Indizes kann bei RECOVER CATALOG durch Angabe des Parameters GENERATE INDEX ON NO LOG INDEX SPACE erfolgen. Für den Neuaufbau der Indizes einer Tabelle werden die Daten der gesamten Tabelle gelesen, die Daten werden sortiert und anschließend werden die Indizes aufgebaut.
- Soll ein Index-Space einzeln repariert werden, so ist dies unabhängig davon, ob er der logischen Datensicherung unterliegt oder nicht. Dies ist mit der Utility-Anweisung RECOVER SPACE *index_space* TO COPY_FILE '*DUMMY' möglich. Dabei wird der Index-Space neu angelegt und die Indizes werden neu aufgebaut.

Wenn die Tabelle relativ viele Sätze enthält, dann ist es vorteilhafter, den Index-Space mit logischer Datensicherung zu prozessieren und die protokollierten Änderungen nachzufahren. Das kann mit der Anweisung RECOVER CATALOG ohne den Parameter GENERATE INDEX ON NO LOG INDEX SPACE erfolgen. Der Index-Space kann einzeln mit der Utility-Anweisung RECOVER SPACE *index_space* repariert werden.

8.5 REORG - Spaces und Basistabellen reorganisieren

Neben der Utility-Anweisung REORG SPACE gibt es die Utility-Anweisung REORG ONLINE TABLE.

Diese beiden Utility-Anweisungen unterscheiden sich voneinander

- durch ihren internen Ablauf
- durch ihr Sperrverhalten

Daraus ergibt sich in Abhängigkeit von bestimmten Randbedingungen auch ein unterschiedliches Verhalten bezüglich der Performance.

Bei REORG SPACE und REORG ONLINE TABLE berücksichtigt SESAM/SQL auch die Freiplatzreservierung, die zuvor in den Anweisungen CREATE SPACE bzw. ALTER SPACE mit der Klausel PCTFREE angegeben wurde.

REORG SPACE

SESAM/SQL reorganisiert einen Anwender-Space mithilfe einer Arbeitsdatei auf Platte in zwei Phasen:

1. Die Arbeitsdatei wird mit den reorganisierten Blöcken des Space aufgebaut. Jede Basistabelle und jeder Index des Space wird in der Arbeitsdatei neu aufgebaut. Logisch zusammenhängende Blöcke einer Tabelle oder eines Index sind dann auch physikalisch zusammenhängend. In dieser Phase können Anwender mit DML-Anweisungen lesend auf die Tabellen und Indizes des Original-Space zugreifen.
2. Die Arbeitsdatei wird umbenannt (RENAME) oder kopiert (COPY). In dieser Phase kann auf den betroffenen Space nicht zugegriffen werden. Der Anwender kann die Sperrzeit kurz halten, wenn er RENAME anstatt COPY verwendet.

Wenn die REORG SPACE-Anweisung in der ersten Phase abgebrochen wird, dann ist der Space in einem konsistenten Zustand. Die Anweisung kann wiederholt werden.

REORG ONLINE TABLE

SESAM/SQL reorganisiert eine Basistabelle durch Modifizieren und Kopieren der Blöcke innerhalb des Anwender-Space im laufenden Betrieb („online“). Die Basistabelle wird stückweise neu aufgebaut. Dafür werden abschnittsweise zusammenhängende freie Blöcke verwendet. Die durch die Reorganisation frei werdenden Blöcke werden wieder verwendet. Wenn zu Beginn der Reorganisation einer stark zerstückelten Tabelle nicht genügend zusammenhängende freie Blöcke zur Verfügung stehen, dann wird der Space erweitert.

Da für die Reorganisation keine exklusiven Transaktionssperren angefordert werden, können andere DML-Anwendungen lesend und ändernd auf die Basistabelle zugreifen. Es wird nur die Basistabelle reorganisiert, nicht aber die zur Basistabelle gehörenden Indizes.

Wenn eine REORG ONLINE TABLE Anweisung abgebrochen wird, dann ist der Space in einem konsistenten Zustand. Die Anweisung kann wiederholt werden.

Einsatzempfehlungen für REORG

Betrachtet man einen Anwender-Space, auf dem eine einzige Basistabelle liegt, so ist die Laufzeit von REORG SPACE performanter als von REORG ONLINE TABLE. Während eines REORG SPACE sind aber keine ändernden DML-Anweisungen erlaubt, die während eines REORG ONLINE TABLE möglich sind. Die Laufzeit von REORG ONLINE TABLE ist größer als die Laufzeit von REORG SPACE.

8.6 Optimierter Indexaufbau für partitionierte Tabellen

Die CREATE INDEX-Anweisung wird in drei Phasen bearbeitet:

1. Lesen der Werte aus den Primärdaten
2. Sortieren der Werte
3. Aufbau der Sekundärindizes

Bei partitionierten Tabellen kann die erste Phase parallelisiert werden, wenn die zu erzeugenden Indizes nicht auf einem der Spaces liegen sollen, die die Partitionen der Tabelle enthalten.

In der ersten Phase wird parallel gelesen, wenn genügend Service-Tasks zur Verfügung stehen. Die gelesenen Daten werden in einem Memory Pool zwischengespeichert. Parallel dazu gibt eine weitere Service-Task die Daten aus dem Memory Pool an die BS2000-Sortierung mit SORT. Die Sortierung startet, wenn alle Daten gelesen und übergeben sind. Nach Ende der Sortierung werden die Sekundärindizes aus den sortierten Daten aufgebaut. Dieser Aufbau kann nicht parallelisiert werden. Er wird in einer Service-Task sequenziell ausgeführt.

Für die optimale Performance müssen Abhängigkeiten zu den DBH-Optionen beachtet werden. Die maximale Parallelität wird erreicht, wenn die Anzahl der zur Verfügung stehen Service-Tasks wenigstens um eins größer ist als die Anzahl der Partitionen. Für weitere parallele Aufträge sollte die Anzahl der Service-Tasks aber höher gewählt werden. Die DBH-Option THREADS muss wenigstens um eins größer sein als die Anzahl der Partitionen. Der Wert des Parameters INITIAL der DBH-Option SERVICE-TASKS muß genügend groß sein.

Wenn nicht genügend Service-Tasks zur Verfügung stehen, dann werden mehrere Arbeitsschritte von einer Service-Task bearbeitet. Die mögliche Parallelität sinkt.

Der optimierte CREATE INDEX steht im linked-in DBH nicht zur Verfügung.

8.7 DROP TABLE DEFERRED / DROP INDEX DEFERRED

Die physikalische Struktur einer Tabelle oder eines Index hat zwei Bereiche:

- den zusammenhängenden Teil
- die so genannten Auslagerungen

Wenn eine Tabelle mit der SQL-Anweisung CREATE TABLE angelegt wird, dann werden die einzufügenden Sätze zunächst im zusammenhängenden Teil gespeichert. Wenn dieser Teil nicht mehr ausreicht, dann wird eine so genannte Auslagerung angelegt; von der Freiplatzverwaltung wird ein freier Block angefordert. Dieser Block gehört logisch zur Tabelle, liegt aber physikalisch nicht mehr im zusammenhängenden Teil.

Wenn ein Anwender-Space mit der Anweisung REORG SPACE reorganisiert wird, dann wird die Tabelle wieder vollständig in einem zusammenhängenden Bereich aufgebaut.

Wenn eine Tabelle mit der SQL-Anweisung DROP TABLE gelöscht wird, dann müssen der zusammenhängende Teil und die Auslagerungen gelöscht werden. Das Löschen des zusammenhängenden Teils geht sehr schnell, da hier bekannt ist, welche Blöcke zu löschen sind. Um die Auslagerungen zu löschen, müssen all diese Blöcke einzeln ermittelt und gelöscht werden. Die Zeit für das Freigeben hängt von der Größe der Tabelle ab. Bei sehr großen Tabellen, deren Anwender-Space lange nicht reorganisiert wurde, kann das Löschen mehrere Stunden in Anspruch nehmen.

Das beschriebene Verhalten gilt analog für Indizes.

Bei den Anweisungen DROP TABLE und DROP INDEX gibt es den Parameter DEFERRED.

Wenn der Parameter DEFERRED spezifiziert ist, dann wird bei einer Tabelle oder einem Index nur der zusammenhängende Teil gelöscht. Die Auslagerungen bleiben erhalten. Damit laufen die Anweisungen DROP TABLE DEFERRED und DROP INDEX DEFERRED bei großen, selten reorganisierten Tabellen oder Indizes deutlich schneller ab als die Anweisungen ohne den Parameter DEFERRED. Bei der nächsten Reorganisation des Anwender-Spaces mit REORG SPACE werden alle existierenden Tabellen und Indizes auf dem Anwender-Space neu aufgebaut. Damit verschwinden die „nicht gelöschten“ Auslagerungen. Erst dann können die frei gewordenen Blöcke wieder verwendet werden.

8.8 ALTER TABLE ... ADD COLUMN mit Indexbeschreibung

Die SQL-Anweisung ALTER TABLE ... ADD COLUMN verfügtüber eine ADD INDEX-Klausel (optional). Damit können mit einer SQL-Anweisung Spalten hinzugefügt und die zugehörigen Indizes definiert werden.

Die Indizes werden sehr schnell aufgebaut, wenn für keine der neuen Spalten eine DEFAULT-Klausel mit einem Wert ungleich NULL angegeben ist. In diesem Fall enthalten die zugehörigen Spalten keine Daten und die Tabelle muss für den Indexaufbau nicht gelesen werden.

Werden die Indizes aber in einer separaten CREATE INDEX-Anweisung definiert, dann muss dafür die gesamte Tabelle gelesen werden. Die Dauer des Lesens ist abhängig von der Anzahl der Sätze in der Tabelle. Diese Zeit kann durch ALTER TABLE ... ADD COLUMN ... ADD INDEX eingespart werden.

In der ADD INDEX-Klausel können nur Indizes für Spalten definiert werden, die auch in der ADD COLUMN-Liste angegeben sind.

Insgesamt ergibt sich mit dieser Erweiterung eine Schreib- und eine Laufzeitersparnis.

8.9 Medienbeschreibung für DDL-TA-LOG-Dateien

Bei der Ausführung von DDL- und SSL-Anweisungen werden die physikalischen Before Images von geänderten Blöcken in der DDL-TA-LOG-Datei (DDLTA-Datei) protokolliert. Diese Information wird zum Rücksetzen der DDL- und SSL-Anweisungen benötigt. Die DDLTA-Datei ist space-spezifisch mit dem Namen:
`<catalog_name>.<space_name>.DDLTA`

Für diese Datei kann eine Medienbeschreibung angegeben werden. Es ist sinnvoll, die Medienbeschreibung anzupassen, wenn DDL-Anweisungen ausgeführt werden sollen, die viele Daten in die DDLTA-Datei protokollieren, wie

- ALTER TABLE ... ADD COLUMN mit DEFAULT-Klausel
- ALTER TABLE ... ALTER COLUMN, wenn die Spaltenlänge verkürzt wird
- ALTER TABLE ... DROP COLUMN

Bei diesen Anweisungen hängt die zu protokollierende Datenmenge von der Anzahl der Sätze der Tabelle ab.

Bei den übrigen DDL- und SSL-Anweisungen ist die zu protokollierende Datenmenge gering.

9 Performance-relevante Aspekte von Administrationsanweisungen

Die Laufzeit einiger Administrationskommandos lässt sich durch günstige Einstellungen für bestimmte DBH-Optionen oder durch bestimmte Maßnahmen verkürzen.

9.1 RECONFIGURE-DBH-SESSION und RELOAD-DBH-SESSION

Eine große Anzahl geänderter Blöcke, für die das Schreiben auf die Spaces der Datenbanken verzögert wird, verlängert möglicherweise die Laufzeit der Aktionen, die bei diesen Administrationskommandos notwendig sind.

Diese Anzahl lässt sich durch die Wahl niedriger Werte für die Parameter BUFFER-LIMIT und TALOG-LIMIT der DBH-Option RESTART-CONTROL minimieren.
Diese Werte können mit der Administrationsanweisung MODIFY-RESTART-CONTROL auch während der DBH-Session, also vor der Ausführung von RECONFIGURE-DBH-SESSION oder RELOAD-DBH-SESSION, modifiziert werden.
Zur Beschreibung der DBH-Option RESTART-CONTROL und der Administrationsanweisung MODIFY-RESTART-CONTROL siehe das Handbuch „Datenbankbetrieb“.

Die Laufzeit der Administrationskommandos wird ebenfalls verringert, wenn Kommandos, die das Durchführen der verzögerten Schreibaufträge erzwingen (z.B. PREPARE-FOREIGN-COPY), noch nicht lange zurück liegen.

Eine große Anzahl offener Transaktionen verlängert die Zeit, bis die Administrationskommandos zur Ausführung kommen.

9.2 SET-SQL-DB-CATALOG-STATUS (STATUS=FREE)

Sätze, die in einer Transaktion gelöscht werden, werden zunächst nur logisch gelöscht. Die Sätze werden asynchron zu einem späteren Zeitpunkt physikalisch gelöscht, spätestens beim Schließen des Spaces.

Wenn eine große Anzahl von Sätzen nicht in einer Transaktion, sondern in mehreren Transaktionen gelöscht wird, dann wird ein Teil der Sätze bereits in dieser Phase physikalisch gelöscht.

Jede Änderung auf dem Space löst physikalische Schreibaufträge aus, die asynchron durchgeführt werden. Beim Schließen eines Space müssen alle noch offenen Schreibaufträge ausgeführt werden.

Diese Anzahl offener Schreibaufträge lässt sich durch die Wahl niedriger Werte für die Parameter BUFFER-LIMIT und TALOG-LIMIT der DBH-Option RESTART-CONTROL minimieren.
Diese Werte können mit der Administrationsanweisung MODIFY-RESTART-CONTROL auch während der DBH-Session, also vor der Ausführung von RECONFIGURE-DBH-SESSION oder RELOAD-DBH-SESSION, modifiziert werden.
Zur Beschreibung der DBH-Option RESTART-CONTROL und der Administrationsanweisung MODIFY-RESTART-CONTROL siehe das Handbuch „Datenbankbetrieb“.

Die Laufzeit der Administrationskommandos wird ebenfalls verringert, wenn Kommandos, die das Durchführen der verzögerten Schreibaufträge erzwingen
(z.B. PREPARE-FOREIGN-COPY), noch nicht lange zurück liegen.

Eine große Anzahl offener Transaktionen verlängert die Zeit, bis die Administrationskommandos zur Ausführung kommen.

9.3 STOP-DBH

Im Rahmen der Beendigung des DBHs werden zunächst offene Transaktionen zurückgesetzt. Eine große Anzahl offener Transaktionen oder auch offene, langlaufende Transaktionen verlängern die Ausführungszeit dieses Administrationskommandos.

Da außerdem alle Datenbanken ordnungsgemäß geschlossen werden, gelten zusätzlich die Aussagen im obigen Abschnitt „SET-SQL-DB-CATALOG-STATUS (STATUS=FREE)“.

Literatur

Die Handbücher finden Sie im Internet unter *http://manuals.ts.fujitsu.com*. Handbücher, die mit einer Bestellnummer angezeigt werden, können Sie in auch gedruckter Form bestellen.

SESAM/SQL-Server (BS2000)
SQL-Sprachbeschreibung Teil 1: SQL-Anweisungen
Benutzerhandbuch

SESAM/SQL-Server (BS2000)
SQL-Sprachbeschreibung Teil 2: Utilities
Benutzerhandbuch

SESAM/SQL-Server (BS2000)
CALL-DML Anwendungen
Benutzerhandbuch

SESAM/SQL-Server (BS2000)
Basishandbuch
Benutzerhandbuch

SESAM/SQL-Server (BS2000)
Datenbankbetrieb
Benutzerhandbuch

SESAM/SQL-Server (BS2000)
Utility-Monitor
Benutzerhandbuch

SESAM/SQL-Server (BS2000)
Fachwörter und Masterindex
Benutzerhandbuch

SESAM/SQL-Server (BS2000)
Meldungen
Benutzerhandbuch

ESQL-COBOL (BS2000)
ESQL-COBOL für SESAM/SQL-Server
Benutzerhandbuch

SESAM-DBAccess
Server-Installation, Administration (nur auf dem Handbuch-Server verfügbar)

openUTM
Konzepte und Funktionen
Benutzerhandbuch

openUTM (BS2000, UNIX-Systeme, Windows NT)
Anwendungen programmieren mit KDCS für COBOL, C und C++
Basishandbuch

openUTM (BS2000)
Anwendungen generieren und betreiben
Benutzerhandbuch

BS2000 OSD/BC
Einführung in die Systembetreuung
Benutzerhandbuch

BS2000 OSD/BC
Performance-Handbuch
Benutzerhandbuch

SHC-OSD / SCCA-BS2
Storage Management für BS2000
Benutzerhandbuch

openSM2 (BS2000)
Software Monitor
Benutzerhandbuch

SORT (BS2000)
SDF-Format
Benutzerhandbuch

SPACEOPT (BS2000)
Optimierung und Reorganisation von Platten
Benutzerhandbuch

Stichwörter

Im Stichwortverzeichnis verweisen **halbfette** Seitenzahlen auf die Hauptfundstellen von Stichwörtern und *kursive* Seitenzahlen auf Beispiele. Es gilt folgende Sortierreihenfolge: Symbole vor Ziffern vor Buchstaben. Satzzeichen sind Symbole.