**Universität Leipzig**
**Fakultät für Mathematik und Informatik**
**Institut für Informatik**

# Analyse des SINAUT Spectrum Graphikdatenmodells und Neuimplementierung unter dem Gesichtspunkt der Granularität und Performanz

**Diplomarbeit von**
**Do Hong Hai**
**August, 1998**

# Danksagung

An dieser Stelle möchte ich mich ganz herzlich beim Herrn Professor Dr. Erhard Rahm und Herrn Dr. Dieter Sosna für die Betreuung meiner Arbeit und für zahlreiche kritische Hinweise bedanken, die mir dabei sehr geholfen haben, meine Arbeit zu verbessern.

Ich möchte mich auch sehr bei der Firma Siemens und ihren Mitarbeitern für die Unterstützung bei der Durchführung der Arbeit und für die Bereitstellung der Hard- und Software bedanken. Besonders danke ich Frau Henriette Zöller, meiner Siemens-Betreuerin, für die ständige Bereitschaft, sich Zeit für meine Fragen und Ideen zu nehmen und mir dabei sehr nützliche Informationen sowie Vorschläge gegeben zu haben.

# Abkürzungsverzeichnis

| | |
|---|---|
| B | Bytes |
| CKPT | Checkpoint Process (ORACLE) |
| CPU | Central Processing Unit |
| DBMS | Database Management System, Datenbankmanagement-System |
| DBWR | Database Writer Process (ORACLE) |
| DDL | Data Definition Language |
| DLM | Distributed Lock Manager (ORACLE) |
| DMS | Distribution Management System |
| EMS | Energy Management System |
| EVU | Energieversorgungsunternehmen |
| I/O | Input/Output |
| KB | Kilobytes |
| KV | Kilovolts |
| LGWR | Log Writer Process (ORACLE) |
| LRU | Least Recently Used |
| MB | Megabytes |
| ODB | Operational Database, Prozeßdatenbank (Siemens) |
| PL/SQL | Procedural Language Extension to SQL (ORACLE) |
| PMON | Process Monitor Process (ORACLE) |
| RDBMS | Relational Database Management System |
| SCADA | Supervisory Controll And Data Acquisation |
| SDB | Source Database, Quelldatenbank (Siemens) |
| SDM | Source Data Management, Quelldatenverwaltung (Siemens) |
| SINAUT | Siemens Netzautomatisierung |
| SMON | System Monitor Process (ORACLE) |
| SQL | Structured Query Language |
| V | Volts |

# Abbildungsverzeichnis

# Inhaltsverzeichnis

# 1 Einführung

## 1.1 Aufgabenstellung

In den Leitsystemen werden komplexe Versorgungsnetze auf zweidimensionale Darstellungen abgebildet, um sie zentral und automatisch überwachen und steuern zu können.

Die Struktur des elektrischen Netzwerks unterliegt einem ständigen Änderungsprozeß. Neue Einrichtungen werden installiert, existierende werden ersetzt oder zusätzliche Komponenten werden eingebaut, um bestimmte Umstände zu kontrollieren. Parallel dazu muß die Datenbank des Kontrollsystems kontinuierlich aktualisiert werden. Die Änderungen im Netzwerk können für lange Zeit im voraus geplant werden oder auch einfach aus den Modifizierungen im täglichen Betrieb resultieren. Dazu gehören kleine Änderungen eines Betriebsparameters sowie komplexe Definitionen von neuen Unterstationen. Die Einteilung der Netzführung in Überwachungs- und Planungsaufgabe bringt die Trennung der Datenbank des Systems mit sich. Die Datenbank für die Überwachungsaufgabe wird in Echtzeit betrieben. Die Applikation muß auf alle möglichen Ereignisse im Netz sofort reagieren können. Die Datenbank für die Planungsaufgabe ermöglicht, Änderungen im Netz während des Netzführungsbetriebs in Echtzeit zu simulieren. Außerdem spielt ihre Standardisiertheit eine entscheidende Rolle dafür, daß die Daten eines kompletten Netzwerks zwischen verschiedenen Leitsystemen exportiert und importiert werden können.

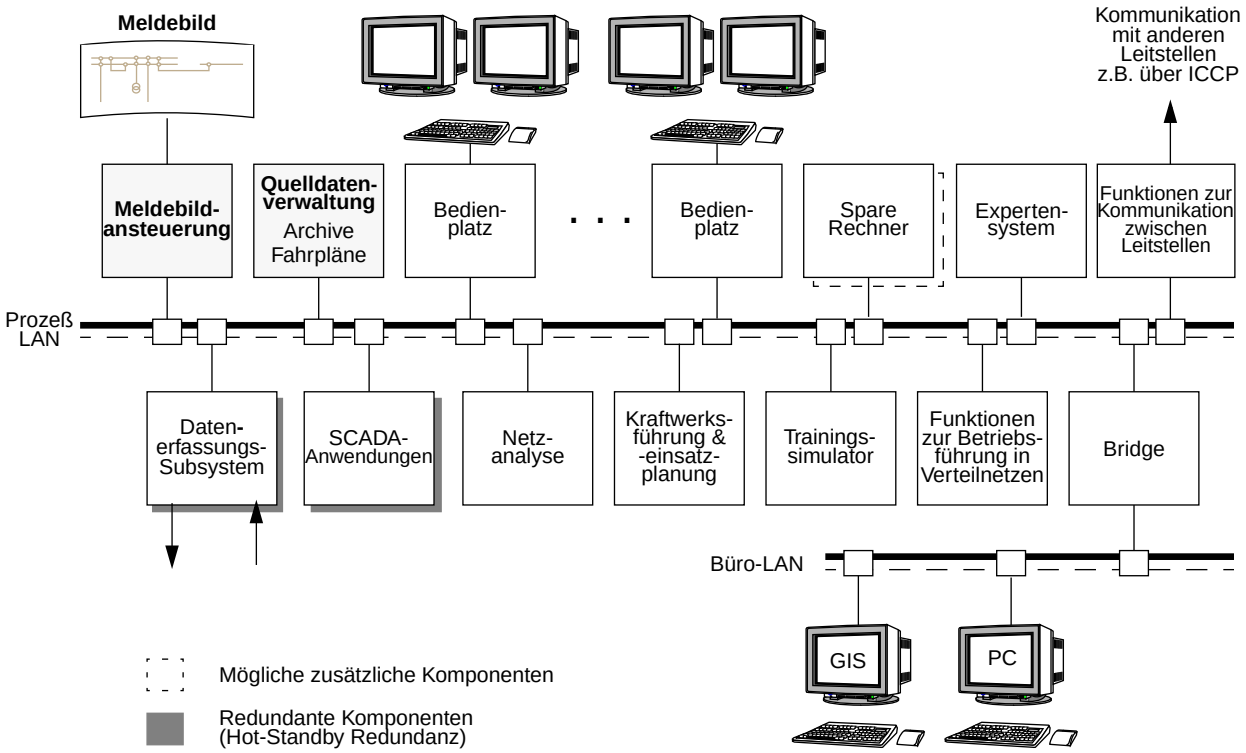
Die Aufgabe der Diplomarbeit ist zuerst zu untersuchen, welche schwerwiegende Nachteile und Probleme die bisherige Modellierung und Implementierung der Repräsentationsdaten der elektrischen Netze auf den Planungs- und Wartungsbetrieb hat und wie man durch neue Modellierung diese Probleme lösen kann.

Als Ergebnis der Arbeit sollen die Kenntnisse darüber gewonnen werden, wie die neu entworfene Struktur der graphischen Darstellung des elektrischen Netzwerks hinsichtlich der Performanz auf die Applikationen wirkt, die die gleichen Aufgaben wie in der alten Struktur haben. Konkrete Implementierungen werden benötigt, um darauf entsprechende Performanzuntersuchungen durchführen zu können. Dazu zählen auch ausführliche Untersuchung der Implementierungsumgebung, um Maßnahmen für Performanzoptimierung ausfindig zu machen.

## 1.2 Das SINAUT Spectrum Netzführungssystem

SINAUT Spectrum ist ein Produkt der Firma Siemens, das zur Führung von elektrischen Netzen sowie Gas-, Wasser- und Fernheiznetzen entwickelt wurde. Es zeichnet sich in erster Linie durch eine offene, modulare und verteilte Systemarchitektur aus. SINAUT Spectrum beinhaltet eine ganze Palette netzleittechnischer Funktionen, zu denen die Basis- und Anwenderfunktionen für SCADA (Supervisory Control And Data Acquisition), EMS (Energy Management System) und DMS (Distribution Management System), sowie Trainingsfunktionen für Betriebsführungspersonal zu zählen sind ([SieNetzleit98]).

Abbildung 1 Beispiel eines SINAUT Spectrum SCADA/EMS/DMS-Systems




Den Hauptgegenstand der Netzführung stellen die zu verwaltenden Netzkomponenten und die Informationen über ihre Zustände dar, die sich im Netzbetrieb dynamisch ändern können. Diese Informationen werden über Leitstellen an die Zentrale übermittelt und dort verarbeitet. Die zentrale Überwachung und Steuerung erfolgen somit durch fernmeldende und fernwirkende Mechanismen und Betriebsmittel.

Im Lebenszyklus der Daten werden zuerst Bilder zur Visualisierung mit Hilfe von Editoren erstellt und beschreibende Daten eingegeben. Es steht z.B. der Graphikeditor zur Erfassung von Netzbildern zur Verfügung. Die Netzdaten und Bilddaten werden dann in verschiedenen Datenbanken untergebracht. Sie unterliegen von nun an einer kontinuierlichen Wartung (Quelldatenverwaltung oder Source Data Management - SDM) und sind Grundlage für die Netzbilder im Echtzeits-Betrieb. Die Datenverwaltung wird durch eine einheitliche, auf drei Phasen basierte Arbeitsweise charakterisiert:

- Eingabe bzw. Änderung der Daten, Überprüfung innerhalb der Quelldatenbank
- Datentest
- Aktivierung der Modifikationen, Übertragung der neuen Daten von der Quelldatenbank in die Prozeßdatenbank, endgültige Aufnahme in die Netzbilder

Alle Daten, die sich auf Erweiterungen bzw. Modifizierungen vom elektrischen Netzwerk beziehen, können dabei in Betracht kommen. In den Modulen der SDM werden verschiedene Funktionen bereitgestellt, die nach Bedarf Aktionen mit den entsprechenden Daten ausführen. Folgende Datengruppen werden in der Netzführung gebildet:

- Technologische Daten, die das Netzwerk beschreiben
- Daten für Telekommunikation

- Daten für die Repräsentation des Netzwerks am Bildschirm
- Daten für die Verarbeitung in den Applikationen

Dabei spielt das Sicherheitsprinzip eine große Rolle, deshalb muß jede Manipulationen an Daten auf Relevanz geprüft und für eventuelle Restaurierung protokolliert werden. Das Änderungsprotokoll ermöglicht die Überwachung der Anzahl der Änderungen sowie die rückblickende Betrachtung der Geschichte von Datenänderungen.

Die Reihenfolge der Änderungen und deren Aktivierungen kann beliebig sein, da sie verschiedenen zufallsbedingten Faktoren unterliegen, die auf die Dauer der am realen Netz durchzuführenden Änderungsarbeiten zurückgehen.

## 1.3 Datenbanksysteme und Anwendung

Seit Anfang der 80er Jahren hat sich ein neues Konzept im Design der Datenbanksysteme durchgesetzt und stellt heute das am meisten angewandten Modell im Gebiet der Informationsverwaltung dar: nämlich das Konzept der relationalen Systeme. Sie werden durch eine wesentlich klarere Unterscheidung zwischen einem physikalischen und logischen Datenmodell im Vergleich zu Systemen in der Vergangenheit wie z.B. Filesystemen charakterisiert. Die relationalen Datenbanksysteme basieren auf einem einfachen konzeptionellen Modell, das aus Relationen bzw. Tabellen besteht. Außerdem wird eine für Administratoren, Entwickler sowie Benutzer einheitliche Sprache für Datendefinition und -manipulation verwendet ([Vossen94], [Meier95]).

Mit unterschiedlichsten Funktionalitäten kann ein Datenbanksystem die meiste Arbeit in der Informationsverwaltung übernehmen. Die Einschränkung ist jedoch, daß Anwendungen mit speziellen Aufgaben unter dem Gesichtspunkt der Performanz nur teilweise unterstützt werden. In SINAUT Spectrum ist es z.B. die Aufgabe der Behandlung von Signalen im automatisierten Echtzeitsbetrieb der Netzführung.

Die zentrale Netzüberwachung fordert schnelle Kommunikations- bzw. Ereignis-Behandlungsmechanismen. Bei einer Störung können im Sekundenbereich tausende Signale von Leitstellen zum Kontrollzentrum strömen und Entscheidungen müssen so schnell wie möglich getroffen werden. Deshalb muß die Prozeßdatenbank, die die statischen (Verarbeitungsparametrierung) sowie dynamischen Prozeßdaten für SCADA-Anwendungen (Meldungen, Meßwerte, Zählwerte, usw.) und Anwendungsdaten für verschiedene EMS- und DMS-Aufgaben enthält, in solchen Fällen die erwartete Performanz liefern. Die Prozeßdatenbank (auch Operational Database - ODB) ist eine eigene Entwickung von Siemens. Mit rationalem Ansatz bei reduziertem Funktionsumfang wird sie für optimalen Zugriff von Applikationen entworfen.

Die Quelldatenbank (auch Source Database - SDB) enthält die Netz- und Sachdaten sowie die Graphikdaten des Netzleitsystems. Sie bietet die Möglichkeit dazu an, im ständigen Betrieb der Netzführung Änderungen im Netz (z.B. neue Installationen) vorauszuplanen und zu testen. Diese Datenbank wird in der Regel innerhalb eines relationalen Datenbankmanagement-Systems (RDBMS) untergebracht (ORACLE). Der Einsatz eines RDBMS für die Datenbank bringt den Vorteil, daß Daten von bestehenden Netzleitsystemen auf gängige RDBMS-Strukturen exportiert und in SINAUT Spectrum wieder importiert werden können. Die dabei verwendeten standardisierten Schnittstellen des RDBMS sind darüber hinaus die Voraussetzung für die Integration von Netzleitsystemen und anderen Informationssystemen im EVU (Energieversorgungsuntermehmen).

Die Diplomarbeit beschäftigt sich vor allem mit den Repräsentationsdaten des elektrischen Netzwerkes. Der Arbeitsbereich wird dem Gebiet der SDM zugeschrieben, weil ein neues Datenmodell für die dynamische Repräsentation der elektrischen Netze entstehen soll und es dabei auf verschiedene Aspekte der Performanz für späteren Einsatz in der SDB untersucht werden soll.

# 2 Analyse der Daten

## 2.1 Das Energienetzwerk und seine Charakteristika

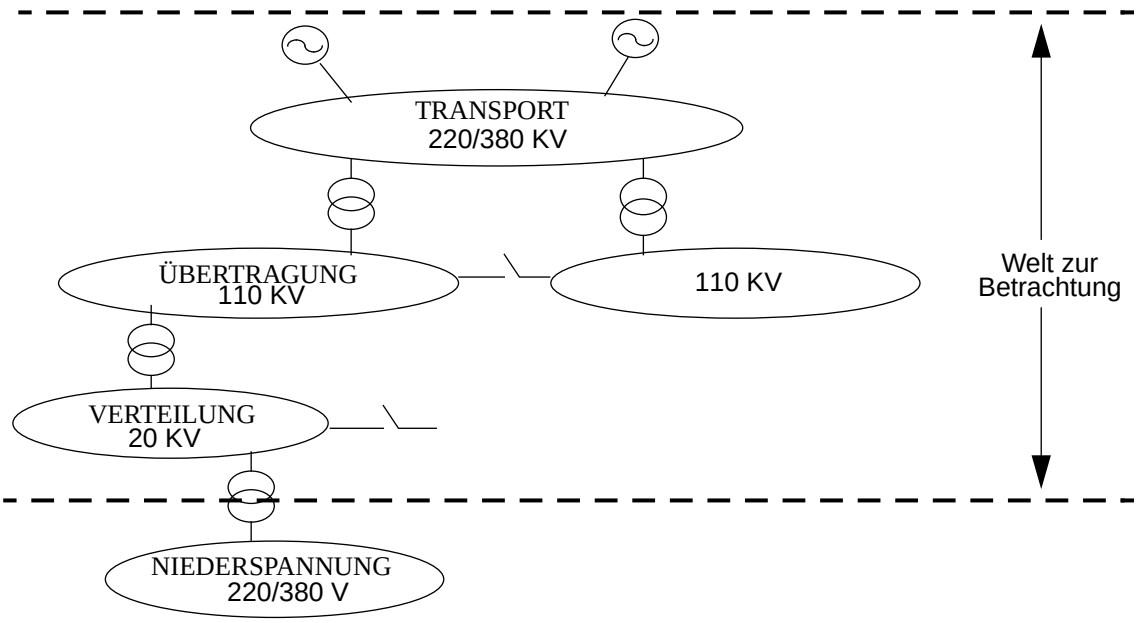
(Literatur zu diesem Abschnitt: [Beißler96])

Um elektrische Netze insbesondere und andere Versorgungsnetze im allgemeinen (z.B. Gasnetze, Wassernetze) zentral kontrollieren zu können, werden sie auf zweidimensionalen Darstellungen abgebildet und mittels dynamischer Kommunikations- und Kontrollmechanismen visualisiert bzw. steuerbar gemacht.

Die Energieversorgung wird heutzutage durch ein Verbundnetz gewährleistet. An ein Verbundnetz sind in der Regel mehrere Kraftwerke angeschlossen, die die elektrische Energie ins Netz einspeisen. Am anderen Ende des Verbundnetzes befinden sich die Verbraucher, die mit elektrischer Energie versorgt werden. Dazwischen existieren verschiedene Netzelemente wie Leitungen, Transformatoren, Schalter. Sie dienen dazu, die elektrische Energie von gewünschter Spannung und Stärke an gewünschte Stellen zu transportieren. Ein Verbundnetz wird dadurch charakterisiert, daß es sich in mehrere Spannungsebenen gliedern läßt. Das Transportnetz ist die Netzebene mit der Höchstspannung (220/380 KV). In diesem Teil des Netzes erfolgen die Einspeisung der Kraftwerke und Verbindungen zu anderen Verbundnetzen. Die Netzebene mit dem nächst-niedrigeren Spannungsniveau (110 KV) ist die Ebene der Hochspannungsnetze. Im diesem Teil des Verbundnetzes erfolgt nur teilweise Einspeisung von Kraftwerken. Verbindungen zu Nachbarnetzen stellen auf dieser Netzebene wenige Sonderfälle dar. An ein Übertragungsnetz schließen sich mehrere Verteilnetze an, die sog. Mittelspannungsnetze (6-20 KV). In diesen Netzen erfolgt keine Einspeisung von den Kraftwerken mehr. Die Verteilnetze können bei Bedarf miteinander verbunden werden. Große Verbraucher (z.B. die Industrie) werden über diese Netze versorgt. Auf der nächsten Netzebene befinden sich die Niederspannungsnetze (220/380 V). Darin sind vor allem die normalen Endverbraucher wie Haushalte zu finden, die mit Niederspannungsstrom versorgt werden.

Ein Verbundnetz hängt immer mit einem geographischen Gebiet zusammen. Für ein geographisch begrenztes Gebiet hat die Netzebene der Höchstspannung die kleinste Menge von Betriebsmitteln im Vergleich zu anderen Netzebenen. Die Komplexität des Netzes, der zu verwaltende Informationsbestand und der Grad der Netzdynamik wachsen, wenn man in der Hierarchie des Verbundnetzes von einer Netzebene mit der höheren Spannung auf eine mit der niedrigeren Spannung heruntergeht. Der Einsatzbereich der Betriebsmittel, die der Fernmeldung und Fernwirkung an Komponenten dienen, verkleinert sich aber auch parallel dazu. In der Netzebene der Mittelspannung ist es oft der Fall, daß die Komponenten nicht direkt mit Leitstellen verbunden werden können, so daß eine automatische Behandlung von Störsignalen in dem betroffenen Netzbereich nicht möglich ist. Die Netzebene der Niederspannung ist für unsere Betrachtung völlig uninteressant, weil der Aufbau von Leitstellen zur Übertragung der Anlageninformationen an die Zentrale hier nicht möglich ist. Zum anderen fehlt jeder Anspruch zur zentralen, automatisierten Überwachung und Steuerung in diesem Netzteil.

Abbildung 2 Spannungsebenen des Verbundnetzes




Ein elektrisches Netz wird zum Zwecke der Netzbetriebsführung in seinem Aufbau und seinem statischen und dynamischen Zustand durch eine Menge von Komponenten beschrieben. Eine Komponente ist die kleinste selbständige Einheit, die eine bestimmte Bedeutung und Funktion im Netz hat. Man gruppiert alle an einem Ort befindlichen Komponenten (Einrichtungen zu Schalten, Umformen und Verteilen der elektrischen Energie einschließlich der Meß-, Schutz- und Hilfseinrichtungen) zu einer Unterstation (auch Station genannt). Unterstation ist die größte selbständige Einheit in der Netzbetrachtung.

Jedes Netz besteht aus einer Vielzahl von Netzkomponenten wie Transformatoren, Leitungen, Leistungsschaltern, Meßpunkten. Jede Netzkomponente kann zu jeder Zeit einen anderen Zustand annehmen, der ihrer Position im Netz und den Zuständen ihrer naheliegenden Netzkomponenten entspricht. Daraus folgt, die Visualisierung eines solchen Netzes ist in erster Linie eine dynamische Darstellung aller im Netz vorhandenen Komponenten. Um die Darstellung des Netzes für den Menschen erfaßbar machen zu können, müssen noch andere visuale Hilfsmittel zugelassen werden, die das Netz zusätzlich mit geographischen Grundrissen oder mit Kommentaren, Symbolen beschreiben. Also neben dem dynamischen Teil des Netzes existiert noch ein statischer Teil in der Netzdarstellung, der über die Veränderungen in Zustand von Netzkomponenten hinweg konstant bleibt.

Um die Aufgaben der zentralisierten Überwachung und Kontrolle zu erfüllen, muß auf jede Komponente zugreifbar bzw. jeder Informationsaustausch zwischen Zentrale und Netzkomponenten protokollierbar sein. Dadurch entsteht die Anforderung, die Komponenten eindeutig zu adressieren. In der Realität wählt man eine mehrstufige (üblicherweise dreistufige) Ortsangabe zusammen mit einer Typangabe zur Identifizierung von Netzkomponenten. Der Vorteil liegt darin, daß jede beliebige Komponente in der Netzhierarchie adressiert werden kann. Das ganze Netz bekommt einen gewissen Grad an geographischer Transparenz, die im zentralisierten Überwachungs- und Kontrollbetrieb des Netzes vorausgesetzt wird.

Die Aufgabe der Netzvisualisierung kann man grob in zwei Teilaufgaben einteilen, einmal das ganze Netz mit allen seinen Komponenten graphisch darzustellen und zum anderen die Kommunikation zwischen dem Nutzer und den einzelnen Komponenten und umgekehrt zu gewährleisten. Das heißt, jede Zustandsänderung (Event) bei den Netzkomponenten muß dynamisch durch graphische Mittel dem Nutzer mitgeteilt werden und Befehle vom Nutzer können jederzeit an einzelne Netzkomponenten übermittelt werden.

Die Auffassung der Netzvisualisierung in zwei Teilaufgaben bringt die Einteilung der Daten in zwei Gruppen mit sich. Die erste Gruppe sind die Daten, die benötigt werden, um das Netzwerk graphisch am Bildschirm darzustellen. Ein Standardverfahren ist, Einheiten aus einfachen graphischen Elementen (sog. Graphikprimitiven: Linienzügen, Polygonzügen, Kreisbögen, Texten) zu konstruieren. Der Prozeß, Netzbilder zu zeichnen, wird auf Routinen vereinfacht, die die einzelnen Graphikprimitive im Bildschirm zeichnen. Die zweite Gruppe sind die Daten, die benötigt werden, um reale Netzkomponenten mittels ihrer graphisch darstellenden Objekte zu steuern. Netzkomponenten können Meldungen über ihren Zustand an Zentrale machen. Netzkomponenten können Befehle von Zentrale ausführen und Bestätigung zurücksenden. Netzkomponenten können Werte liefern, um Protokolle bzw. Prüfungen auf Grenzwerte zu ermöglichen. Ein Transformator kann verschiedene Stufenstellungen haben. Diese Daten kann man den Charakteristika und den Funktionalitäten der verschiedenen Typen der realen Netzkomponenten entnehmen. Daraus werden entsprechende Datentypen gebildet, um den darstellenden Objekten zuweisen zu können.

Die dynamische Darstellung des Netzes bedeutet nun, Zustandsänderungen einzelner Netzkomponenten zu registrieren und sie dynamisch auf dem Bildschirm darzustellen.

## 2.2 Das SINAUT Spectrum Datenmodell und Implementierung

(Literatur zu diesem Abschnitt: [SieGraEdit97])

### 2.2.1 Das SINAUT Spectrum Datenmodell - Kriterien und Definitionen

Das reale Netzwerk kann sich in Größe und Komplexität von Fall zu Fall unterscheiden. Es ist aber nicht immer möglich, das gesamte Netz als eine und einzige Einheit zu verwalten. Das Verbundnetz stellt selbst eine heterogene Struktur dar: es besteht aus Subnetzen unterschiedlicher Spannungsniveaus. Deshalb ist die erste Aufgabe der Modellierung, ein Kriterium zu finden, das eine angemessene Einteilung des realen Netzwerks ermöglicht. Dadurch müssen technologisch sinnvolle Substrukturen entstehen, die unabhängig voneinander überwacht und kontrolliert werden können. Für die Modellierung des Verbundnetzes ist eine Einteilung nach dem Kriterium der Spannungsniveaus der einzige Weg. Alle Daten, die relevant für ein Spannungsniveau sind, werden zusammengefaßt. Die Signale werden auch spezifisch innerhalb des Spannungsniveaus abgearbeitet.

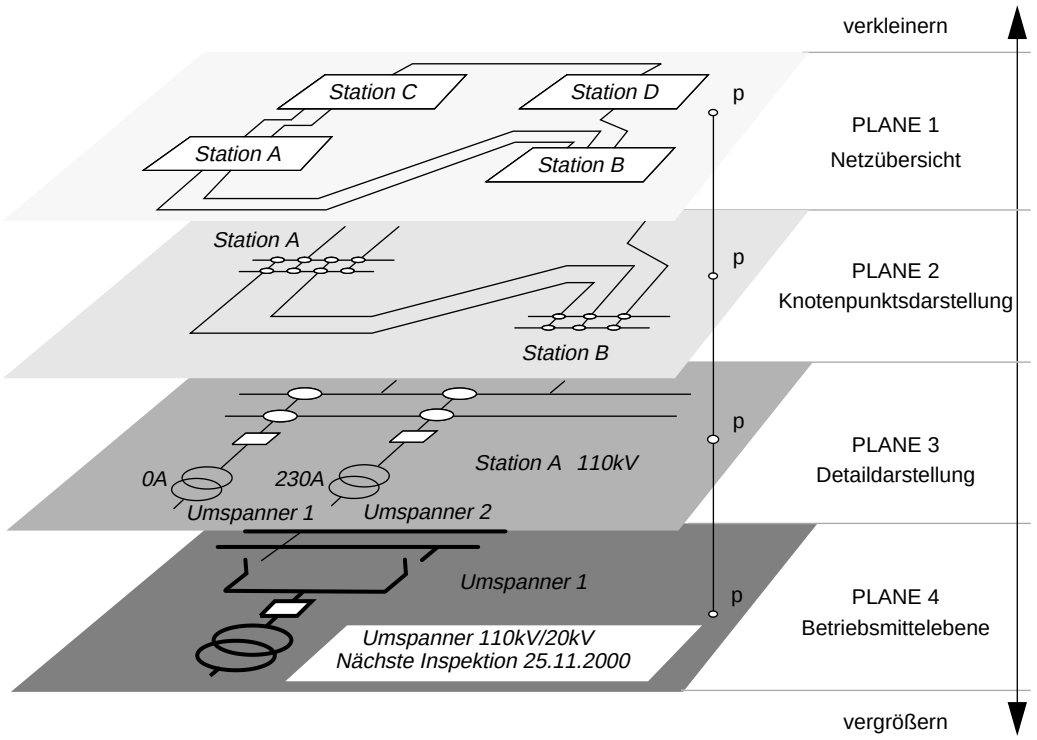
*Definition 1)* *Alle Bilder des SINAUT Spectrum Vollgraphiksystems sind Worldmaps. Eine Worldmap ist eine zweidimensionale Darstellung eines Teils der realen Welt.*

Eine Worldmap stellt ein elektrisches Netz von Komponenten eines einheitlichen Spannungsniveaus, das sich weit über geographische Grenzen erstrecken kann, in einem zweidimensionalen Koordinatensystem dar. Jeder Punkt der Worldmap wird durch ein eindeutiges Paar von X,Y-Koordinaten (auch Weltkoordinaten genannt) definiert.

Um die Aufgabe der Überwachung zu erfüllen, muß die Visualisierung des Netzes in einer Worldmap eine Übersicht über das ganze Netz wiedergeben können, in der alle Stationen als auswählbare Symbole mit Signalfunktionalität angezeigt werden. Um die Aufgabe der Kontrolle zu erfüllen, muß die Worldmapdarstellung auch detaillierte Informationen über Zustand einzelner Stationen und ihrer Komponenten enthalten. Daraus ergibt sich die Notwendigkeit, die Sicht über eine Worldmap in Planes (Ebenen) zu verfeinern.

*Definition 2)* *Planes bilden in verschiedenen Maßstäben den gleichen Teil der Welt ab, und zwar in einem Höhenbereich zwischen einem maximalen und einem minimalen Vergrößerungsfaktor. Eine Worldmap besteht mindestens aus einer Plane.*

Abbildung 3 Ausschnitt einer Worldmap




Jede Plane umfaßt also den gesamten 2-D Bereich mit dem gesamten Umfang an eindeutigen Weltkoordinaten. Der Punkt P (Abbildung 3) hat in allen Planes dieselben Weltkoordinaten (X,Y). Die Darstellung einer Worldmap in Planes unterliegt dem Kriterium der Detailliertheit, mit der man die Komponenten der Worldmap betrachten will. Alle Planes sind nur in dem ihr zugewiesenen Vergrößerungsbereich sichtbar. Sie enthalten unterschiedliche graphische Darstellungen derselben realen technologischen Komponenten. Da die Vergrößerungsbereiche der Planes kontinuierlich sind, kann man mit der Vergrößerung der Planes logischerweise mehr Details der Worldmap erfahren.

Es gibt mehrere Gründe dafür, Planes weiter zu zerlegen. Die größten Substrukturen, die dargestellt werden müssen, sind die Unterstationen. Der direkte Zugriff auf einzelne Unterstationen muß im Überwachungs- und Kontrollablauf unterstützt werden. Außerdem sprechen die enorme Größe einer Unterstation und die Komplexität der darin enthaltenen Informationen auch für eine separate Darstellung.

*Definition 3)* *Ein Segment bezeichnet einen geographischen Bereich innerhalb einer Plane und umfaßt eine genau definierte Menge von graphischen Elementen.*

Ein Segment ist im allgemeinen über eine Taste anwählbar, damit ein schneller Zugriff auf Netzkomponenten erfolgen kann. Das ist besonders wichtig, wenn Störungen im Netz auftreten. Der Koordinatenbereich eines Segments ist eine Untermenge der Weltkoordinaten. Die Komponenten der Stationen werden in Segmenten zusammengefaßt. Die Anzahl der Komponenten ist pro Segment vom System limitiert, was manchmal dazu führt, das ein Stationbild nicht in einem einzigen Segment untergebracht werden kann.

Alle Netzkomponenten, die im realen Netzwerk existieren, werden in Objekte der Segmente abgebildet. Das graphische Aussehen dieser Objekte verändert sich dem Zustand der dargestellten Komponente entsprechend. Neben solchen Objekten existieren auch andere Objekte, die dazu da sind, der Worldmap eine bessere Verständlichkeit zu geben, z.B. zusätzliche Symbole, Kommentare, Namen. Sie unterliegen keiner Veränderung, die während des betrieblichen Ablaufs zugefügt werden kann.

*Definition 4)* *Ein Objekt ist die kleinste in Bedeutung und Funktion unabhängige Einheit der Netzdarstellung. In der Regel bildet ein Objekt eine real existierende Netzkomponente ab. Ein solches Objekt heißt dynamisch, da es sich dynamisch dem Zustand der Komponente anpassen muß. Alle anderen Objekte werden statische Objekte genannt.*

Die graphischen Darstellungseinheiten, die Komponenten des realen Netzes abbilden, müssen sich ändern, wenn die abgebildeten Netzkomponenten neue Zustände angenommen haben. Dazu gehören nicht nur Form- oder Farbänderungen, sondern auch Werteänderungen oder Meldungsausgaben. In dieser Hinsicht sind sie auch als Variablen verschiedener Typen zu betrachten.

*Definition 5)* *Eine Variable ist ein dynamisches Objekt.*

Jede Variable hat eine zweiseitige Anwendung. Erstens zeigt sie die Informationen über den Zustand der Netzkomponente durch ihre aktuelle graphische Repräsentation an. Es ist die Displayfunktion der Variablen. Folgende graphische Mittel stehen zur Verfügung, dynamische Informationen darzustellen: Symbole, Nummern, Diagramme, Farben und verschiedene Blinkzustände.

*Definition 6)* *Eine Displaybeschreibung ist ein Ansammlung von Daten, die bestimmen, wie das dynamische Objekt (die Variable), zu dem die Beschreibung gehört, graphisch dargestellt werden muß, um die benötigten Informationen über die reale Schaltanlage anzuzeigen und ggf. eine Zustandsänderung zu signalisieren.*

Variablen können zweitens mittels Änderung ihrer Attribute durch direkte Dateneingaben die Komponente steuern. Es ist die Inputfunktion der Variablen. Nicht alle Variablentypen können Dateneingaben entgegennehmen.

*Definition 7)* *Eine Inputbeschreibung ist eine Ansammlung von Daten, die bestimmen, in welcher Form das dynamische Objekt (die Variable), zu dem die Beschreibung gehört, Eingabe in der Zentralsteuerung entgegennehmen kann.*

Es gibt Variablentypen, die sich ausschließlich auf die Displayfunktion beschränken, z.B. Time- bzw. Text-Variablen, die dafür verwendet werden, Zeiten bzw. Meldungen auszugeben. Value-Variablen werden definiert, um aktuell gemessene Werte einer Messtation anzuzeigen. Wenn eine Sammlung von historischen Werten für die Betrachtung notwendig ist, wird eine Logbook-Variable benutzt. Für Zustandsänderungen aller Art in den Komponenten kann man die Message-Variablen benutzen. Je

nach der graphischen Gestalt der Variable wird entweder eine neue Repräsentation für die Komponente oder ein neues Attribut in Textform aktiviert und angezeigt bzw. deaktiviert und vom Bildschirm gelöscht. Zur Unterstützung verschiedener Formen des Datenaustausches zwischen den Komponenten und ihren Repräsentationen können Mask-Variablen definiert werden. Daten können auch abhängig von ihren Änderungen nach der Zeit in Kurven dargestellt, dazu benötigt man die Curve-Variablen. Ausgewählte Werte aus dem Segment können mit Hilfe einer VID-Curve-Variable angezeigt werden. Der Unterschied einer VID-Kurve zu einer normalen Kurve ist, daß die VID-Kurve die Werte von den anderen Variablen bekommt, die die Werte besitzen (z.B. Values- oder Logbook-Variablen). Die graphische Anzeige dieser Werte bleiben im Bild auch noch, wenn die angezeigten Werte sich bereits in Echtzeit geändert haben.

Jedes Objekt der Worldmap hat eine graphische Repräsentationsform und ist im Bildschirm sichtbar. Dadurch wird dem Nutzer über die Existenz einer spezifischen Netzkomponente mitgeteilt. Alle graphischen Repräsentationen bestehen aus einigen vordefinierten Graphikprimitiven.

*Definition 8)* *Die Graphikprimitive des SINAUT Spectrum Vollgraphiksystems sind Linienzug, Polygonzug, Kreis, Vektortext und Pixeltext. Dabei werden Sonderfälle wie Gerade als ein horizontaler oder vertikaler Linienzug mit zwei Punkten, Rechteck als ein Polygonzug mit vier Punkten und vier rechten Winkeln, Kreisbogen als ein Teil von einem Kreis wie andere Primitive behandelt.*

Mit Hilfe dieser Graphikprimitive lassen sich eine Vielzahl von Graphikelementen und letztendlich ganze Netzbilder darstellen. Netzkomponenten können durch ein einziges Graphikprimitiv repräsentiert werden, wie z.B. eine Transmissionslinie durch einen Linienzug, eine Meldung durch einen Text. Es gibt komplexere Netzkomponenten, die nur durch eine Zusammensetzung von mehreren Graphikprimitiven symbolisieren lassen, wie z.B. Transformatoren, Leistungsschalter.

*Definition 9)* *Figuren sind besondere graphische Repräsentationsformen der Objekte in den Worldmaps. Sie sind aus mehreren Graphikprimitiven zusammengesetzt. Figuren unterscheiden sich untereinander in ihrem Einsatzbereich. Die Figuren, die in allen Worldmaps vorkommen können, werden Globalfiguren genannt. Die Figuren, die dagegen nur in der aktuellen Worldmap eine Bedeutung haben, bilden die Gruppe der Lokalfiguren der zugehörigen Worldmap.*

Für gleiche Netzkomponenten werden gleiche graphische Symbole benötigt. Man findet sie in allen Teilen des Netzes wieder, weshalb ihre zugehörigen graphischen Repräsentationen separat für Wiederverwendung gespeichert werden müssen. Man kann aber auch innerhalb einer Worldmap mehrere Graphikprimitive zu einer Lokalfigur gruppieren, wenn sie gemeinsam eine Komponente des Netzes darstellen oder auch wenn sie gemeinsam eine Einheit bilden. Die Figur hat eine Bedeutung, die nur in der aktuellen Worldmap nachvollziehbar ist. Netzfremde Objekte werden als einzelne Graphikprimitive unabhängig voneinander und von anderen Netzobjekten betrachtet und gespeichert, da ihr Zusammenspiel in bezug auf das Netz uninteressant ist.

### 2.2.2 Aktuelle Implementierung

Die Bilddaten des SINAUT Spectrum Graphiksystems werden zur Zeit schon in ORACLE verwaltet. Diese Realisierung ist einfach und unzusammenhängend. Sie besteht aus mehreren Tabellen, deren Struktur hauptsächlich aus einer Nummer und einer LONG RAW-Spalte besteht. Die Datensatznum-

mer *kid* wurde vom System für jeden Datensatz beim Generieren der Datenbank vergeben und sind für uns uninteressant. Die anderen Nummern sagen aus, um welche Worldmaps es sich handelt, bzw. zu welcher Worldmap eine Plane gehört. Auf die Probleme dieser Implementierung wird im nächsten Abschnitt noch konkreter eingegangen.

Die Datenbank wurde mit keinerlei Constraints bzw. Triggern vor inkonsistenten Datenmanipulationen geschützt, da es nicht möglich ist, auf die LONG RAW-Spalten auf Datenbankebene zu zugreifen und zu analysieren. Also besteht die Datenbank ausschließlich aus Tabellen. Sie sind im Anhang A aufgelistet, um eine genaue Übersicht über die jetzige Implementierung zu geben. Die C-Strukturen, die in den Applikationen verwendet werden, um Daten aus den LONG RAW-Spalten zu lesen, sagen aus, welche Informationen darin enthalten sind.

Zur Illustration wird der Aufbau der Relation der Worldmap-Topologien hier aufgeführt:

Tabelle 1 FXWMAPT

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | *+* | *record identifier* |
| *wmap* | *NUMBER* | *-* | *worldmap number* |
| *data* | *LONG RAW* | *-* | *data* |

wobei der *data*-Eintrag folgende C-Struktur beinhaltet:

```
  typedef struct {             /* relation fxWMapT                      */
    short       wmap;          /* 00 worldmap root segment id.      KEY */
    short       version;       /* 02 worldmap version                   */
    float       Crange;        /* 04 user coordinate range              */
    long        REFdots;       /* 08 dots of reference monitor          */
    float       REFdist;       /* 12 distance view monitor of ref. mon. */
    tVRPmode    vrpmode;       /* 16 vrp - mode of worldmap             */
    char        bspare[4];     /* 20 spare                              */
    short       toolbar;       /* 24 toolbar identifier (->TOOLBAR)     */
    short       fnklink;       /* 26 fnklink identifier (->FNKLINK)     */
    short       sspare[1];     /* 28 spare                              */
    short       plNumb;        /* 30 number of planes                   */
    short       Bcol;          /* 32 background colour                  */
    short       WMopt;         /* 34 worldmap options (see below)       */
    short       zoomL,zoomH;   /* 36 zoom factor range (WMopt bit 2=1)  */
    Fpoint      sr[2];         /* 40 surrounding rectangle (zoomable)   */
    long        free;          /* 56 first free addr.  (rel.to record)  */
    } fxWMapT;                 /* 60                                    */
```

## 2.3 Probleme dieser Implementierung

Mit der LONG RAW-Darstellung der Daten in dieser Implementierung kann man wohl das Optimale der Performanz der Applikationen erreichen, die auf die Datenbank zugreifen und Daten daraus lesen, da die von einem RDBMS zur Verfügung gestellten Funktionalitäten hier minimal in Einsatz kommen. Man muß im Gegenzug dazu eine Menge von Problemen und Nachteilen mit dieser Repräsentation der Daten hinnehmen.

### 2.3.1 Direkter Zugriff

Die LONG RAW-Darstellung verbirgt von vornherein die Sicht auf die Datenbank, da die Werte des Datentyps LONG RAW in ORACLE nicht richtig ausgewählt und angezeigt werden kann wie andere Datentypen wie VARCHAR, NUMBER. Der Benutzer kann nichts anders als Worldmap-, Plane- oder Segment-Nummern aus den Datensätzen der Relationen lesen. Er muß erst Applikationen starten (die Graphikeditoren), um sehen zu können, was sich in diesen LONG RAW-Feldern verbirgt, also welche Graphikobjekte ein Segment enthält bzw. welche Komponenten des realen Netzwerks im Segment aufgefaßt sind. Statistiken über den Aufbau einzelner Worldmaps, z.B. welche Variablentypen und wieviele zugehörige Objekte existieren, welche Graphikrepräsentationsformen und wieviele Objekte mit einer bestimmten Repräsentationsform es gibt, sind also im Datenbanksystem nicht durchführbar.

Die elementaren Befehle (SELECT, INSERT, UPDATE, DELETE) werden in der Datenbank nur teilweise, vor allem nur bei den Nicht-LONG RAW-Spalten, unterstützt. Die eigentliche Datenbank ist eine Datenbank der Worldmaps mit ihren Planes und Segmenten. Die Segmente enthalten wiederum statische Objekte und Variablen. Jedes Segment wird in einem einzigen Datensatz abgespeichert, also seine allgemeine Beschreibung zusammen mit allen seinen Objekten. Ein Segment ist dementsprechend als Ganzes in den Speicher einzulesen und auch als Ganzes in die Datenbank zurückzuschreiben. Auf einzelne Objekte kann nur über Applikationen zugegriffen werden.

Die ORACLE's SQL-Spracherweiterung PL/SQL kann die LONG RAW-Spalten, die in der Datenbank abgespeichert sind, nicht richtig bearbeiten. In PL/SQL kann man zwar den Wert einer LONG RAW-Spalte in eine LONG RAW-Wirtsvariable selektieren aber die Länge der Variable ist auf 32 KB beschränkt. Also die Spalten, deren Länge diesen Grenzwert überschreiten, was häufig vorkommt, da sie bis zu 2 GB zugelassen ist, verursachen Laufzeitfehler oder werden einfach abgeschnitten. Um den automatischen Codegenerierungsmechanismus im gesamten Datenbanksystem zu unterstützen, muß in PL/SQL möglich sein, einzelne Objekte zu manipulieren und deren Änderungen zu kontrollieren und zu protokollieren.

Von den Vorteilen der PL/SQL im Vergleich zu den Wirtssprachen, hier z.B. besserem Zugriff auf Datenobjekte, einfacher Datenmanipulation, kann deshalb noch kein Gebrauch gemacht werden. So können Applikationen einige Zeit beim Laden bzw. Schreiben der Daten in großen Mengen gewinnen, aber wiederum Zeit dafür verschwenden, um einzelne Objekte in Segment zu manipulieren, da wenn ein Objekt sich ändert, es auch heißt, daß das ganze Segment sich ändert.

### 2.3.2 Redundanz und Referenz

Die Datenbank enthält viel redundante Daten. Die LONG RAW-Spalten sind zwar schnell als eine Einheit in einem SELECT-Befehl in den Speicher einlesbar. Sie haben aber den Nachteil, daß die Applikationen erst mittels Zeiger und Schleifen die Daten daraus lesen können. Aus dem Grund wurde die Datenstruktur so gewählt, daß die Daten, die gemeinsam an einer Stelle benutzt werden, einfach in das gleiche LONG RAW-Feld gepackt und abgespeichert werden, unabhängig davon, ob sie irgendwo schon vorhanden sind. Dadurch möchte man Datenbankzugriffe sparen und Laufzeit in den Suchschleifen vermeiden. Dies verursacht wiederum mehr Konsistenzprobleme, da man immer eine Übersicht über die redundanten Daten in den Relationen behalten muß, wenn man die Objekte modifiziert bzw. neu erzeugt. Mehr Zeitaufwand wird in solchen Änderungen benötigt, da alle Stellen, wo sich die gleichen Daten befinden, zu aktualisieren sind.

Es existieren z.B. Tabellen: FXWMNAME, FXPLNAME, FXSGNAME, die die Namen der Worldmaps, Planes und Segmente enthalten. In der Beschreibung der Segmente (FXWSEGS) sind aber alle Namen für die zugehörigen Worldmap, Plane und Segment noch einmal vorhanden.

Die Message-Variablen haben in ihrer Displaybeschreibung ein eigenes Feld für die technologischen Adressen:

```
  struct {                    /*Typified Combination               */
    tB1             B1;       /*Block number B1                    */
    tB2             B2;       /*Block number B2                    */
    tB3             B3;       /*Block number B3                    */
    tElem           Elem;     /*(Standard)element number           */
  } U0;
```

In der Inputbeschreibung der Message-Variable sind folgende Einträge zu finden:

```
  struct {
    byte        dummy[3];      /* 00 hole for word grid              */
    tDispVar    vipVar;        /* 03 variable type: meVar            */
    byte        spare1[2];     /* 04                                 */
    short       IncJoGr;       /* 06 index of invar. NC job group    */
    tTa         meTa;          /* 08 technological address           */
  } me;                        /* 20                                 */
```

Im Feld *meTa* sind die Werte *B1*, *B2*, *B3* und *Elem* nochmals eingetragen. Dies passiert in der Regel bei den Variablenbeschreibungen, wenn man für die Displaybeschreibung und Inputbeschreibung des Objekts seine technologische Adresse braucht.

Die Referenzierungen in der Datenbank werden auf eine komplizierte Weise realisiert. In den Tabellen FXWMAPP, FXWMAPS werden die Zuordnungen von Segment zu Plane und von Plane zu Worldmap festgehalten. Die Worldmap-, Plane- und Segment-Nummern werden in eigenen Spalten gespeichert. Aber die *data*-Spalten der beiden Tabellen enthält nochmals diese Referenzierungen. Hier ist die Struktur der *data*-Spalte der FXWMAPS-Tabelle:

```
  typedef struct {            /* relation fxWMapS                   */
    short       wmap;          /*  00 worldmap number             KEY */
    short       plane;         /*  02 plane number                KEY */
    short       segm;          /*  04 segment number              KEY */
    short       did;           /*  06 segment identifier (->WSegD,WSegS)*/
    char        comm[80];      /*  08 comment                         */
    } fxWMapS;                 /*  88                                 */
```

Und in den Tabellen FXWMNAME, FXPLNAME und FXSGNAME, die eigentlich nur die Namen der Worldmaps, Planes bzw. Segmente enthalten, kann man dieselben Zuordnungen wiederfinden (siehe Anhang A ***Aktuelle Implementierung: Tabellen und LONG RAW-Strukturen,*** Seite 91).

Die Segmente müssen außerdem in der Datenbank eindeutig numeriert werden, da alle Segmente, getrennt in statischen und dynamischen Segmenten (FXWSEGS, FXWSEGD), unabhängig von Planes, zu denen sie gehören, in einer eigenen Relation abgelegt werden. Also es gibt zwei unterschiedliche Zuordnungen von Segmenten (einmal mittels *wmap, plan, segm*, einmal mittels *did*), was die hierarchische Struktur der Worldmaps bricht. Die eindeutige Korrespondenz dazwischen muß in der gesamten Datenbank dementsprechend immer aufrechtgehalten werden.

Die Display- und Inputbeschreibungen der Variablen werden auch per Segment gespeichert. Alle Variablen brauchen aber eine graphische Repräsentation im Bild. Deshalb gibt es noch eine Referenzierung von den dynamischen Graphikobjekten in der Relation FXWSEGD zu den Variablenbeschreibungen in DISPDES und INPUTVAR. Diese Referenzierung kann man erst herstellen, wenn man durch alle Elemente in drei Tabellen durchgelaufen ist.

Die graphischen Repräsentationen der Variablen eines Segments werden normalerweise zusammen in einem LONG RAW-Feld in der Relation FXWSEGD abgespeichert. Da gibt es aber eine Ausnahme, nämlich alle Lokalfiguren, die ausschließlich graphische Repräsentationen für dynamische Objekte sind, werden in dem LONG RAW-Feld des zugehörigen Datensatzes in der Relation FXWSEGS gespeichert, wo eigentlich nur die Graphikbeschreibungen für statische Objekte zusammengefaßt sind. Die Referenzierung erfolgt mit Hilfe einer Offsetangabe, die in der Objektbeschreibung in der Relation FXWSEGD einkodiert wird. Es verursacht zwar wenig Aufwand, wenn die Worldmap gezeichnet werden soll, weil alle zugehörigen Datensätze (also alle benötigten LONG RAW-Felder einschließlich FXWSEGS, FXWSEGD) schon vorher geladen werden. Um an die Daten zu einer Lokalfigur zu kommen, braucht das Programm nur eine Adreßumrechnung mit Hilfe dieser Offsetangabe. Die Art und Weise, Lokalfiguren so abzulegen, entspricht aber nicht der logischen Struktur einer Datenbank, in der die zusammengehörenden Daten auch zusammen (in einer Relation) abgelegt werden.

### 2.3.3 Konsistenz

Die LONG RAW-Darstellung verhindert jeden Versuch, Konsistenzkontrolle auf der Datenbankebene zu implementieren, da die Bearbeitung der LONG RAW-Felder nur in einer höheren Programmiersprache möglich ist (C/C++, Fortran, COBOL). Die Datenbank reduziert sich auf einige Relationen und existiert nur noch als Ablageplatz für die Daten der Applikationen, und nicht als eine selbständige datenverwaltende Einheit wie im eigentlichen Sinne. Konsistenzchecks müssen auf einer höheren Ebene, nämlich auf der Applikationsebene, implementiert werden. Die Applikationen, die auf die Daten zugreifen bzw. die Daten erzeugen, müssen also immer prüfen, ob die Daten sich in einem konsistenten Zustand befinden, bevor sie die Daten darstellen oder in die Relationen schreiben. Es bringt nicht nur viel Zeitaufwand sondern auch viel Programmieraufwand mit sich, da die Konsistenzkontrolle sich nicht an einer Stelle abspielt, sondern verteilt, in jeder Applikation. Die Änderung irgendeiner Beziehung zwischen den Entitäten oder die Hinzunahme einer neuen Entität wird veranlassen, daß alle Schnittstellen der Applikationen zur Datenbank umprogrammiert werden müssen. Außerdem kann die Datenbank bei solcher hohen Datenredundanz und ohne Integritätsschutz von Constraints und Trigger leicht in einen inkonsistenten Zustand geraten. Selbst wenn es der Fall ist, können die inkonsistenten Daten unentdeckt in der Datenbank bleiben und in Anwendungen später zu unerklärlichen Fehlern führen.

Für die Objekte gelten bestimmte Bedingungen, die ihnen ein gültiges Aussehen geben. Darunter sind solche Konsistenzprobleme zu verstehen, die sich innerhalb der graphischen Form eines Objektes abspielen. Es gibt bestimmte Linientexturen, mit denen sich ein Polygon, Polyline, Rectangle, Circle bzw. ein Arc zeichnen läßt. Es gibt bestimmte Verfahren, mit denen sich ein Polygon, ein Rectangle oder ein Circle füllen läßt. Nicht alle graphischen Repräsentationsformen können aus mehreren Graphikprimitiven bestehen. Globalfiguren und Lokalfiguren können das. Eine Variable kann nur bestimmte Formen der Graphikrepräsentation annehmen. Diese Konsistenzanforderungen spielen aber nur eine untergeordnete Rolle, da die Probleme, die mit ihnen im Zusammenhang stehen, in der Regel leicht korrigierbar sind.

Die Objekte unterliegen außerdem auch anderen Beziehungen, die sie mit anderen Elementen der Netzdaten verbinden. Solche Beziehungen sind ganz wichtig, um ein dynamisches Objekt als Abbild einer Komponente im realen Netzwerk darstellen zu können. Diese Beziehungen korrekt zu halten hat die höchste Priorität, die an die Datenbank gestellt wird und spricht für sich deshalb als die wichtigste Konsistenzanforderung.

Ein Teil der Beschreibung einer Value-Variable hat z.B. folgende Struktur:

```
typedef struct tVaInfDes {
  tTa              TA;           /*technological address              */
  tNimSet          NimSatz;      /*record No. within network image    */
  NormElem         NoElType;     /*norm element type: measVal/countVal*/
  tADSubTyp        ADSubTyp;     /*for applicat.data subtype else dummy*/
  short            CL_SHORT_FILLER;
} tVaInfDes;
```

Eine Variable (hier eine Value-Variable) ist immer mit einer technologischen Adresse verbunden. Eine technologische Adresse besteht aus fünf Feldern: *B1*-, *B2*-, *B3*-, *Elem*-, und *Info*-Namen. Durch diese Adresse wird eine Komponente referenziert, das im realen Netzwerk vorhanden ist, und das Online-System kann immer dafür sorgen, daß die Variable die Werte von der eindeutig bestimmten Netzkomponente bekommt. Die Variable zeigt dann im Bild an, welche Werte die dargestellte Komponente (eine Messtation) gerade angenommen hat. Im folgenden Beispiel werden die Elemente von einem Bereich des Netzwerkes durch technologische Adressen benannt und identifiziert:

Abbildung 4 Technologische Adressen

```
Siemens System SPECTRUM              DMS-Listing Program DML       Page     1

L i s t i n g   o f   E l e m e n t s

---B1---   ---B2---   ---B3---   Element   ElemType  NoElType    NimSet

Berlin     110        Leipzig    B1 Spec   BlocTags  Univers         20 Element text:--------------------

                                 Topo 1    TopElem   TopElem        134 Element text:--------------------
                                 Topo 2    TopElem   TopElem        135 Element text:--------------------
                                 Topo 3    TopElem   TopElem        136 Element text:--------------------
                                 I         mv I a    MeasVal        834 Element text:--------------------
                                 P         mv P      MeasVal        835 Element text:--------------------
                                 Q         mv Q      MeasVal        836 Element text:--------------------
                                 Iso Bb 1  Iso       Switch          40 Element text:--------------------
                                 Iso Bb 2  Iso       Switch          44 Element text:--------------------
                                 CB        CB        Switch          41 Element text:--------------------
                                 Iso Line  Iso       Switch          42 Element text:--------------------
                                 Iso Eth   Iso       Switch          43 Element text:--------------------
```

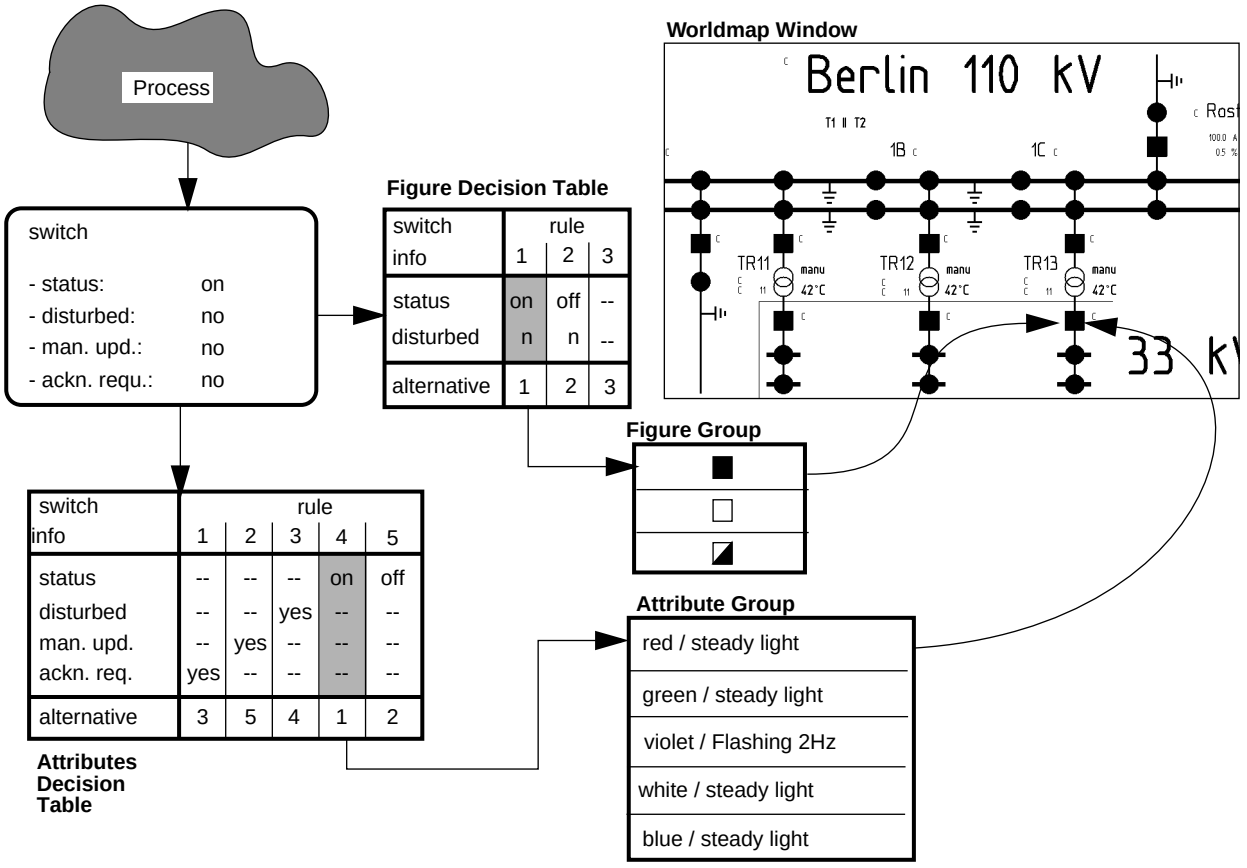
Eine Message-Variable besitzt u.a. folgende Attribute:

```
vID         id;            /*variable identifier                      */
short       aAlt;          /*image alternative of attribute           */
short       aGroup;        /*image group of attribute                 */
short       deTaNA;        /*Number of decision table for attribute gr*/
short       deTaNF;        /*Number of decision table for figure group*/
```

Eine Netzkomponente kann verschiedene Zustände annehmen. In verschiedenen Zuständen muß ihre graphische Form auch verschieden dargestellt werden. Eine Figurgruppe ist ein externes Datenobjekt, aus dem eine graphische Repräsentation für eine Komponente ihrem Zustand entsprechend gewählt

werden kann. Um eine Komponente in einem bestimmten Zustand darstellen zu können, braucht man aber außerdem noch andere Attribute wie Farben, Blinkraten. Diese Attribute, die spezifisch für einen bestimmten Typ von Komponenten sind, machen eine Attributgruppe aus. Je mehr Attribute eine Komponente hat, desto mehr Varianten werden zum Symbolisieren dieser Komponente benötigt, weil es aus ihrer Darstellung auch hervorgehen muß, wie die relevanten Attribute (Informationen) besetzt sind. Eine endgültige Darstellung ergibt sich erst aus der Betrachtung aller Attribute. Deshalb müssen die Wahl der Figuralternativen bzw. Attributalternativen über Entscheidungstabellen (Decision Tables) erfolgen. Der Vorgang wird im folgenden Bild illustriert:

Abbildung 5 Abhängigkeit von externen Datenobjekten




Diese Referenzierungen spiegeln sich derzeit nur in den Anwendungen (Graphikeditoren) wider. Beim Einfügen einer Variablen in Segment, werden zugehörige Checks angestoßen, um festzustellen, daß die Variable eine gültige technologische Adresse hat und die Entscheidungstabellen, Figur- bzw. Attributgruppen auch wirklich vorhanden sind.

Um die Konsistenzprobleme direkt zu lösen, müssen die LONG RAW-Felder aufgebrochen werden. Innerhalb der Datenbank kann man einfach Check- bzw. Foreign Key-Constraints oder Trigger implementieren und die Bewahrung der Datenintegrität der Kontrollinstanz des Datenbanksystems überlassen.

### 2.3.4 Sperrgranularität

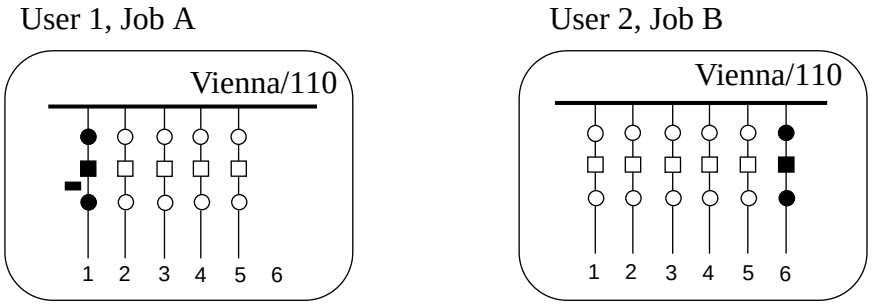
Die Herstellung der Worldmaps ist eine sehr komplexe Aufgabe. Hinzu kommt, daß die Struktur des elektrischen Netzes ständigen Änderungen unterliegt. Eine selbstverständliche Anforderung an das System ist, daß die Bearbeitungsprozesse an Worldmaps in einem Mehrbenutzerbetrieb parallel unterstützt werden müssen.

Die LONG RAW-Darstellung der Daten entspricht nicht dieser Anforderung. Ein Segment wird als Ganzes in den Speicher geladen und in der Regel erst beim Abschluß der Änderungen zurückgeschrieben. Um die Datenbank nicht in einen inkonsistenten Zustand zu bringen, wird von vornherein vorausgesetzt, daß zwei Benutzer nicht das gleiche Segment (normalerweise ein Stationbild, das in der LONG RAW-Spalte des Segment-Datensatzes abgespeichert ist) editiert. Das kleinste von ORACLE implementierte Sperrgranulat sind Datensätze. Es führt dazu, daß in den Jobs festgestellt werden muß, daß das Segment nicht schon von einem anderen Benutzer in einem anderen Job editiert wird. Das ganze Segment wird dann gegenüber anderen Benutzern gesperrt.

Datenänderungen werden in der Regel vorbereitet und erst dann aktiviert, wenn der entsprechende Umbau in der Anlage erfolgt ist. Dies setzt voraus, daß mehrere Änderungen innerhalb einer Worldmap sowie innerhalb einer Station parallel vorbereitet werden können. Die Änderungen werden als *Delta*'s abgelegt, die dann vom System auf den jeweiligen aktuellen Datenbestand angewendet werden können. Die Reihenfolge der Änderungen in der Anlage ist beliebig, da diese Änderungen abhängig vom Aufwand und der Dauer der Arbeit unterschiedlich abgeschlossen werden. Durch die LONG RAW-Darstellung der Daten kann die Reihenfolge der Aktivierungen nicht beliebig sein, da ein Objekt immer an einen Job gebunden ist und eine Aktivierung, die sich auf eine Änderung im Bild bezieht, erst möglich wird, wenn diese Änderung auch vollzogen ist. Während der Änderung eines Segments ist es für andere Benutzer nicht erlaubt, weitere Änderung an Objekten dieses Segments durchzuführen und zu aktivieren. Um parallele Aktivierungen zulassen zu können, muß es den Benutzern erlaubt sein, parallel an einem Segment editieren zu können. Dies wird nur erreicht, wenn der Sperrmechanismus des Segments als Ganzen gegenüber anderen Benutzern aufgelöst wird. Dadurch wird die Notwendigkeit ersichtlich, die Segmente in einzelnen zugehörigen Objekten anstatt als ein einziges Großobjekt zu speichern. Die Änderungen bzw. Aktivierungen orientieren sich an Objekten und können nun beliebig durchgeführt werden, nachdem das einzig betroffene Objekt gesperrt wird.

Der Vorteil der Darstellung in einzelnen Objekten wird sein, daß ein neues und einheitliches Verfahren für die Verwaltung aller Daten - Netzdaten und Bilddaten - realisiert werden kann, das die Verwaltung der Änderungen in Jobs und das Aktivieren von einzelnen Änderungen unterstützt. Der Vorgang der Änderung und Aktivierung wird im folgenden Beispiel illustriert:

Abbildung 6 Parallele Änderungen im Stationbild *Vienna*

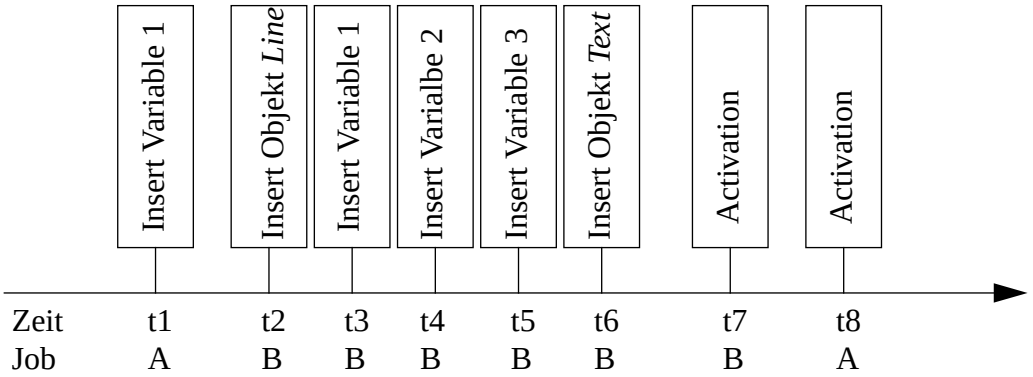



Das Beispiel zeigt folgende Situation:

- Benutzer 1 ändert in Job A die Darstellung des Abzweigs 1.
- Benutzer 2 fügt in Job B die Darstellung des Abzweigs 6 hinzu.

Beide Änderungen werden vorbereitet und erst aktiviert, wenn die Änderungen im Netz abgeschlossen sind. Die Reihenfolge der Aktivierung der beiden Änderungen kann beliebig sein. Durch die Darstellung der Stationbilder in einzelnen kleinen statischen und dynamischen Objekten kann der Prozeß vereinfacht wie folgt ablaufen:

Abbildung 7 Ablauf der Modifizierungen und Aktivierungen




In der jetzigen Implementierung, wo eine Unterstation (hier *Vienna*) komplett in einem LONG RAW-Feld abgelegt wird, kann nur der Benutzer editieren und aktivieren, der zuerst mit seiner Arbeit begonnen hat. Der andere muß so lange warten, bis das Segment mit dem Stationbild *Vienna* freigegeben wird. Erst dann kann er seine Änderungen machen und aktivieren. Das ist sicherlich nicht sinnvoll, denn wenn ein Umbau in der Anlage länger dauert, wird die Aktivierung auf die entsprechende Änderung im Netzbild auch verzögert.

## 2.4 Präzisierung der Zielstellung

### 2.4.1 Ziel

Nach der Betrachtung der Probleme und Nachteile, die die jetzige Implementation der Datenbank der Bilddaten hervorgerufen hat, ist nun ein Weg zu finden, um all diese Probleme lösen zu können. Ziel

der Diplomarbeit ist, eine neue Datenbank für die Bilddaten zu entwerfen und zu implementieren, die folgende Anforderungen erfüllen muß:

- Durch neuen Entwurf die LONG RAW-Darstellung der Daten aufzubrechen und einem bestimmten Grad der Granularität entsprechend die voneinander unabhängige Objekte einzeln abzuspeichern.
- Durch entsprechende Constraints und Trigger die Konsistenz der Bilddaten im allgemeinen und die Konsistenz der Bilddaten in Beziehung zu den Netzdaten im einzelnen zu halten.
- Durch zugehörige Komponenten wie Packages Routinen bereitzustellen, um Datenbankzugriffe von anderen Applikationen bzw. innerhalb der Datenbankumgebung zu unterstützen und zu vereinfachen.
- Durch entsprechende Optimierungen wie z.B. Denormalisierungen des Datenschemas, Parametrierung des Datenbankservers bzw. durch entsprechende Datenstrukturen und Algorithmen für die Funktionen eine optimale Performanz der Datenbankzugriffe zu gewährleisten, um Wartezeiten bei den zeitaufwendigen Aktionen wie z.B. Laden großer Worldmaps geringzuhalten.

### 2.4.2 Teilschritte zur Lösung

Die Aufgabenstellung kann u.a. in folgenden Teilaufgaben betrachtet werden:

- Modellierung

  In dieser Phase soll eine genauere Analyse der verschiedenen Entitäten innerhalb der Worldmap-Beschreibung vorgenommen werden. Aus der alten Datendarstellung geht schon hervor, daß zumindest drei Entitäten existieren, nämlich Worldmap, Plane und Segment. Deshalb muß man sich hier besonders auf die Komponenten eines Segments konzentrieren. Die Objekte werden unter verschiedenen Aspekten betrachtet. Vom graphischen Gesichtspunkt her müssen alle Objekte irgendeine Repräsentation im Bild haben. Vom logischen Gesichtspunkt her müssen für diejenigen Objekte, die eine reale Schaltkomponente abbilden, noch weitere Vorschriften existieren, wie sie sich graphisch dem Zustand der realen Netzkomponenten entsprechend ändern. Dadurch kann man Grenzen zwischen den Objekten ziehen bzw. verschiedene Objekttypen, die in der alten Datendarstellung zwar verschieden betrachtet werden, doch wegen ihrer ähnlichen Struktur zusammenfassen.

  Um die Beziehungen zwischen den Entitäten zu konkretisieren, soll hier eine Beschreibung der Miniwelt aufgestellt werden, in der nur die Entitäten, die für die unsere Datenbank relevant sind, und deren Beziehungen zueinander erläutert werden.

  Aus der Analyse der Entitäten soll eine konkrete Beschreibung für jeden Entitätstyp hervorgehen. Die Attribute, die für eine Entität relevant sind, werden von der alten Datendarstellung herausgefiltert, um redundante Daten in der neuen Datenbank zu vermeiden.

  Das entsprechende Entity-Relationship-Diagramm soll dann aufgrund dieser Analyse aufgestellt werden, um die Beziehung zwischen den Entitäten besser zu illustrieren und dem Entwurf der Datenbank noch einen logischen Halt zu geben.

  Dem Entwurfsprozeß liegt die Zerlegung der Worldmap-Daten in granulare Einheiten (Entitäten) zugrunde. Normalerweise ist es der Fall, daß das entworfene Datenschema nicht gleich realisierbar ist. Optimierungsschritte sind zuerst für ein zu sehr zerkleinertes Datenschema notwendig,

bevor es in eine (wahrscheinlich) brauchbare Implementierung umgesetzt werden kann. Dabei sind verschiedene Strategien vorzuschlagen, um weitere Schemata zu gewinnen, wenn ein Schema den Performanzanforderungen später beim Testen nicht genügt.

- Implementierung

  In dieser Phase soll die komplette Datenbank mit ihren physikalischen Bestandteilen implementiert werden. Die Objekte mit ihren Attributen werden entsprechend in den ORACLE-Relationen abgespeichert. Die Beziehungen der Objekte zueinander bzw. die Datentypen der Attribute, die Wertebereiche der Attribute spiegeln sich in der Implementierung der neuen Relationen wieder. Zugehörige Constraints, Trigger müssen in der Datenbank eingebaut werden, um Anforderungen der Konsistenz in Bilddaten erfüllen zu können.

  Um eine bessere Performanz zu erreichen, müssen die Vorteile der Zugriffe auf die Datenbank von PL/SQL im Vergleich zu Zugriffen von Schnittstellen zu höheren Programmiersprachen ausgenutzt werden. Deshalb sollen die häufigsten Datenbankaktionen in Packages als Stored Procedures implementiert werden, die dann innerhalb ORACLE-RDBMS gespeichert und dynamisch aufgerufen werden können. Zugriffe von Applikationen sollen jedoch auch unterstützt werden, deshalb müssen verschiedene Schnittstellenfunktionen in ProC/C++ bereitliegen, um den Applikationen die Zugriffe auf die Datenbank zu vereinfachen.

  Für den Fall, daß ein Performanzvergleich zwischen mehreren Datenschemata notwendig ist, um feststellen zu können, was die Optimierungen bringen bzw. welches Schema besser ist, müssen auch beide implementiert werden.

- Generieren der Datenbank

  In dieser Phase soll die bereit implementierte Datenbank mit dem Datenbestand der existierenden Worldmaps gefüllt werden. Da die LONG RAW-Spalten nur in höheren Programmierebenen zu bearbeiten sind, müssen entsprechende Konvertierungsroutinen in ProC/C++ geschrieben werden, um aus den LONG RAW-Spalten die zugehörige Daten für einzelne Objekte zu lesen und in ORACLE zu speichern.

  Die Konvertierungsprogramme müssen die nicht mehr relevanten Daten, die sich aus irgendeinem Grund zwar noch im alten Datenbestand befinden, ausschliessen, um die neue Datenbank nicht mit unnötigem Speicherplatz zu belasten.

  Um den Import vom vorhandenen Datenbestand in verschiedene Datenschemata zu unterstützen, müssen die Konvertierroutinen auch spezifisch für einzelne Schemata entwickelt werden.

- Performanzanalyse

  In dieser Phase sollen verschiedene Algorithmen gefunden und implementiert werden, um Aussage über die Performanz der neuen Datenbank in verschiedenen Fällen machen zu können. Eine sehr zeitaufwendige Aufgabe an die Datenbank ist eine komplette Worldmap zu laden. In der alten Implementierung wäre dies in ein paar Datenbankzugriffen getan: je aus den Relationen müssen ein oder mehrere zugehörige LONG RAW-Felder, der Anzahl der Planes, der Segmente entsprechend, in den Speicher gelesen werden. Die Worldmapkomponenten werden dann im Speicher mittels Zeiger durchsucht und dann dargestellt. In der neuen Datenbank, wenn wir die Segmente in kleinen Objekten zerlegt abgespeichert haben, benötigt das System mehr Zeit dazu, die Objekte von der Datenbank zu lesen. Ausschlaggebend für eine akzeptierbare Ladezeit wird

sein, daß der Benutzer in einer vernünftige Wartezeit seine Worldmap auf dem Bildschirm bekommt. Auf die Ladezeit der Worldmaps soll in erster Linie eingegangen werden.

In Frage kommen auch umfangreiche Änderungen einer Worldmap. Dazu gehören die Aktionen UPDATE, INSERT, und DELETE, die auf einzelne Objekte angewendet werden, weil sie häufig in der Realität vorkommen und in der Darstellung der Worldmaps in kleinen Objekten sicherlich Wartezeit verursachen. Es soll Aussage darüber gemacht werden, wie das System im schlimmsten Fall, z.B. Löschen bzw. Einfügen einer kompletten Worldmap, reagiert.

Zum Schluß muß die Entscheidung darüber getroffen werden, ob die verbrauchte Zeit akzeptierbar ist oder nicht. Wenn nicht, dann müssen Gründe hierfür ausfindig gemacht werden und entsprechende Optimierungsschritte vorgenommen werden.

Die Performanzoptimierung ist an mehreren Stellen zu realisieren. Die Performanz einer Applikation, die auf die Daten einer Datenbank zugreift und sie manipuliert, hängt davon ab, wie schnell der Datenbankserver ist, wie die Daten in der Datenbank abgelegt werden und wie die Zugriffe implementiert werden. Die erste Aufgabe ist deshalb der Datenbankserver zu optimieren. Dazu gehört in erster Linie eine angemessene Parametrierung seiner Bestandteile (Speicher, Dateien, ...). Die zweite Aufgabe ist das Datenschema zu optimieren. Bereits in der Modellierung müssen schon verschiedene Möglichkeiten bzw. Strategien genannt werden, um das Datenschema für bessere Performanz zu ändern, das dann in die Datenbank umgesetzt wird. Die dritte Aufgabe ist, passende Datenstrukturen und Zugriffsalgorithmen zu erarbeiten, die auf das implementierte Datenschema gerichtet sein muß und dabei die beste Performanz liefert.

# 3 Modellierung

(Literatur zu diesem Abschnitt: [Barker90], [Batini92], [Elmasri94], [Vossen94])

## 3.1 Methode des Entwurfs

Ziel des Datenbankentwurfsprozesses ist der Entwurf der logischen und physikalischen Struktur einer Datenbank, “daß die Informationsbedürfnisse der Benutzer in einer Organisation für bestimmte Anwendungen adäquat befriedigt werden können” ([Vossen94], Seite 45).

Der Vorgang ist in verschiedenen Schritten zu realisieren. Erstens ist eine gründliche Studie der sog. Informations- bzw. Bearbeitungsanforderungen notwendig. Als Resultat weiß man, welcher Teil der realen Welt zu verwalten ist und aus welchen Elementen er besteht.

Der zweite Schritt ist, auf der obigen Studie basierend ein konzeptionelles Datenschema, also eine “konzeptionelle Globalsicht” zu erstellen. Die Datenobjekte und Beziehungen werden in dieser Phase modelliert und auf Relevanz geprüft. Dafür gibt es zwei Standardverfahren: man kann das Entity-Relationship-Diagramm der Datenobjekte aufstellen, also mit konkreten Datenobjekten beginnen und ihre Beziehungen zueinander bestimmen (*top-down*), oder die Datenobjekte vom Gesamtbestand der Informationen durch Normalisierungen (*bottom-up*) herausarbeiten. Das erste Verfahren ist schnell, fordert im Gegenzug aber Unternehmenskenntnisse bzw. lange Studienzeit. Das zweite Verfahren ist zwar langsam, eignet sich aber sehr gut für die Prüfung der Relevanz der Daten bzw. der Redundanz in den Daten. Deshalb empfiehlt sich hier eine Kombination der beiden Verfahren: *top-down* modellieren und *bottom-up* prüfen.

Der dritte Schritt ist, das konzeptionelles Datenschema mit Hilfe von Transformationsregeln in ein Datenbankschema umzusetzen, das sich dem Zielsystem entspricht. Z.B. soll für ein relationales Datenbanksystem dadurch ein relationales Datenmodell (das sog. externes Datenschema) entstehen.

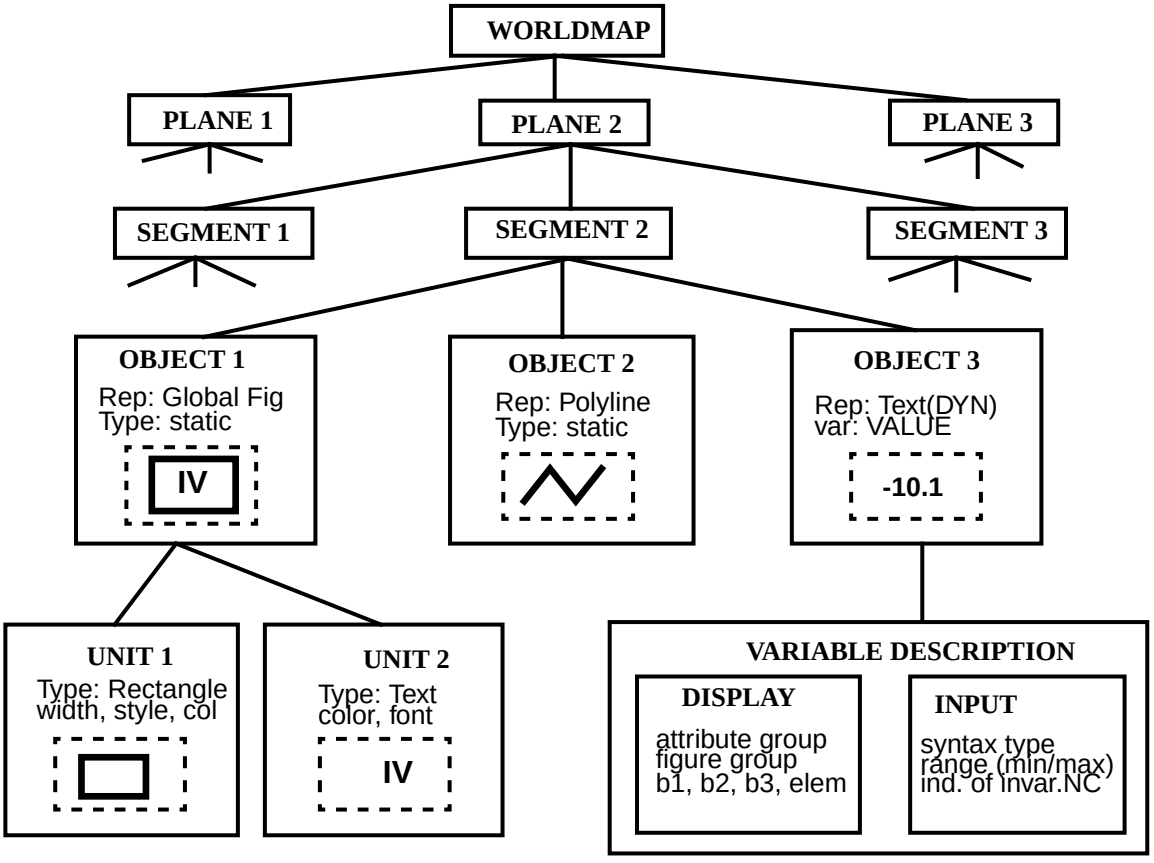
Der vierte Schritt ist, das interne Datenschema im Zielsystem zu realisieren. Mit DDL-Befehlen (Data Definition Language) organisiert man den physikalischen Speicher für die Aufnahme von Daten.

Die Nutzbarkeit eines Datenbankentwurfs besteht nun darin, wie die physikalische Struktur auf die Performanz der Anwendungen wirkt. Man kann es entweder schon im 2. Schritt voraussehen, besonders wenn das resultierende Datenschema zu kompliziert ist, oder erst beim Durchlaufstests von Anwendungen nach dem 4. Schritt beurteilen. Die Entwurfsschritte müssen ggf. wiederholt werden.

## 3.2 Miniwelt

Aus der Analyse der Worldmap-Daten geht folgende hierarchische Struktur der Worldmap-Elemente hervor:

Abbildung 8 Hierarchische Struktur einer Worldmap

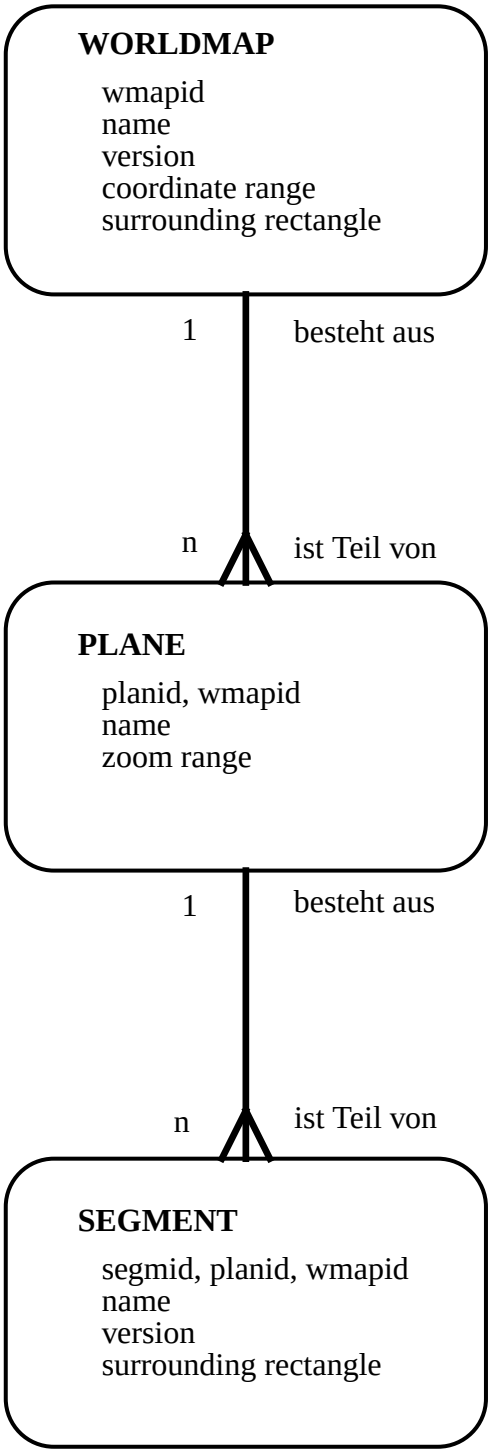



Im folgenden werden nun die interessanten Datenobjekte und ihre Beziehungen zueinander in Kurzform zusammengeführt und erläutert. Auf die Aufzählung der Attribute der einzelnen Objekte und Erklärung ihrer Bedeutungen wird absichtlich verzichtet.

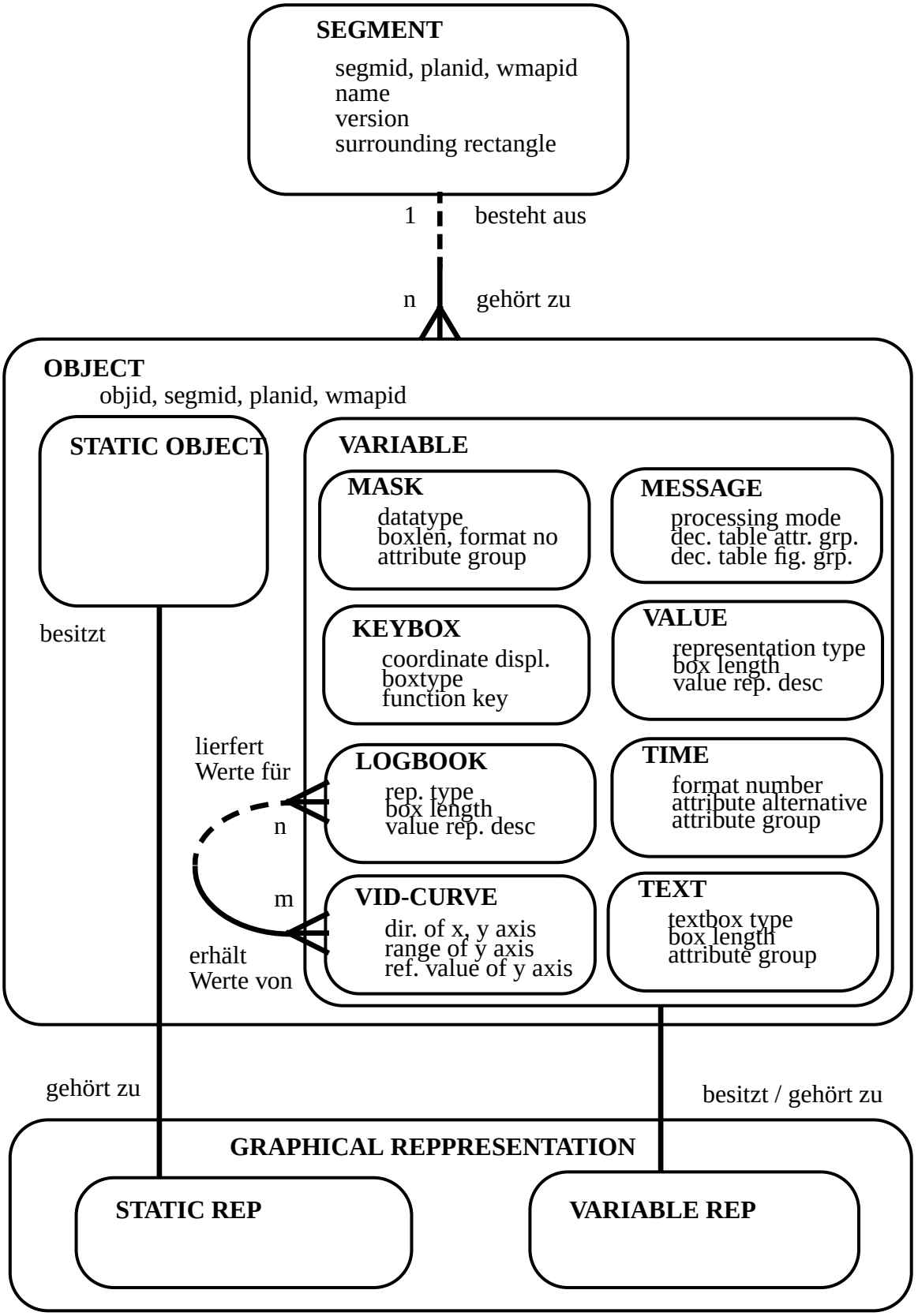
- Alle Bilder des SINAUT Spectrum Vollgraphiksystems sind Worldmaps.

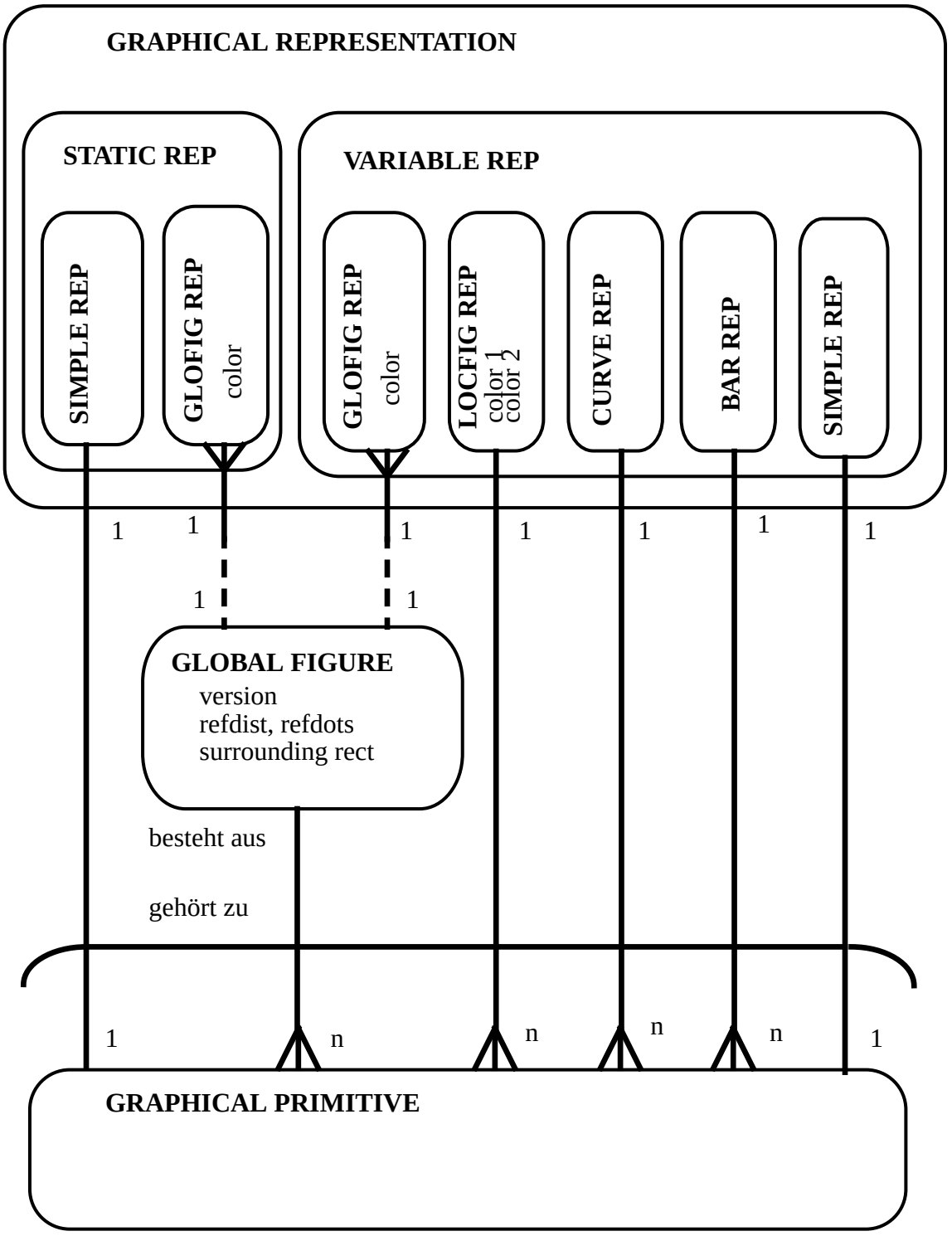
- Jede Worldmap besteht aus einer oder mehreren zweidimensionalen Abbildungen der realen Welt (Planes).
- Jede Plane besteht aus einem oder mehreren Segmenten. Segmente bezeichnen geographische Bereiche innerhalb einer Plane und sind voneinander unabhängig.
- Jedes Segment umschließt genau eine definierte Menge von statischen Objekten und dynamischen Objekten.
- Die dynamischen Objekte dienen der Visualisierung des Anlagenzustands. Jedes dynamisches Objekt hat eine Beschreibung, wie die Form des Objekts sich von Zustand zu Zustand ändert. Sie ist die Displaybeschreibung. Objekte, die der Steuerung der Netzkomponenten dienen, haben noch eine Beschreibung, wie und welche Parameter an das Objekt zu übergeben sind. Sie ist die Inputbeschreibung. Die Display- und Inputbeschreibung bilden gemeinsam die Variablenbeschreibung eines dynamischen Objekts. In dieser Hinsicht werden die dynamischen Objekte als Variablen in verschiedene Typen klassifiziert. Es gibt folgende Typen: Mask, Message, Keybox, Logbook, Text, Time, Curve, VID-Curve und Value.
- Jedes Objekt (ob dynamisches oder statisches) hat eine graphische Repräsentationsform. Ein dynamisches Objekt kann die graphische Form einer Globalfigur, einer Lokalfigur, einer Kurve eines Balkens oder eines einfachen Graphikelements annehmen, ein statisches Objekt dagegen nur die Form einer Globalfigur oder eines einfachen Graphikelements.
- Jede graphische Repräsentationsform besteht aus einem oder mehreren Graphikprimitiven.
- Die Global- und Lokalfiguren haben die gleiche graphische Struktur. Sie bestehen aus verschiedenen Graphikprimitiven. Der Unterschied zwischen den Globalen Figuren und Lokalen Figuren besteht darin, daß eine Globale Figur über die Grenze der Worldmaps hinaus eine Bedeutung hat, im Vergleich zu einer Lokalen Figur, die nur im aktuellen Worldmap eine Bedeutung besitzt und somit nicht wiederverwendbar ist.
- Es werden zwischen Linienzug, Gerade, Polygonzug, Rechteck, Vektortext, Pixeltext, Kreis und Kreisbogen als Typen der Graphikprimitive unterschieden, wobei die Sonderfälle wie Gerade, Rechteck und Kreisbogen selbständig wie anderen Primitiv-Typen behandelt werden.

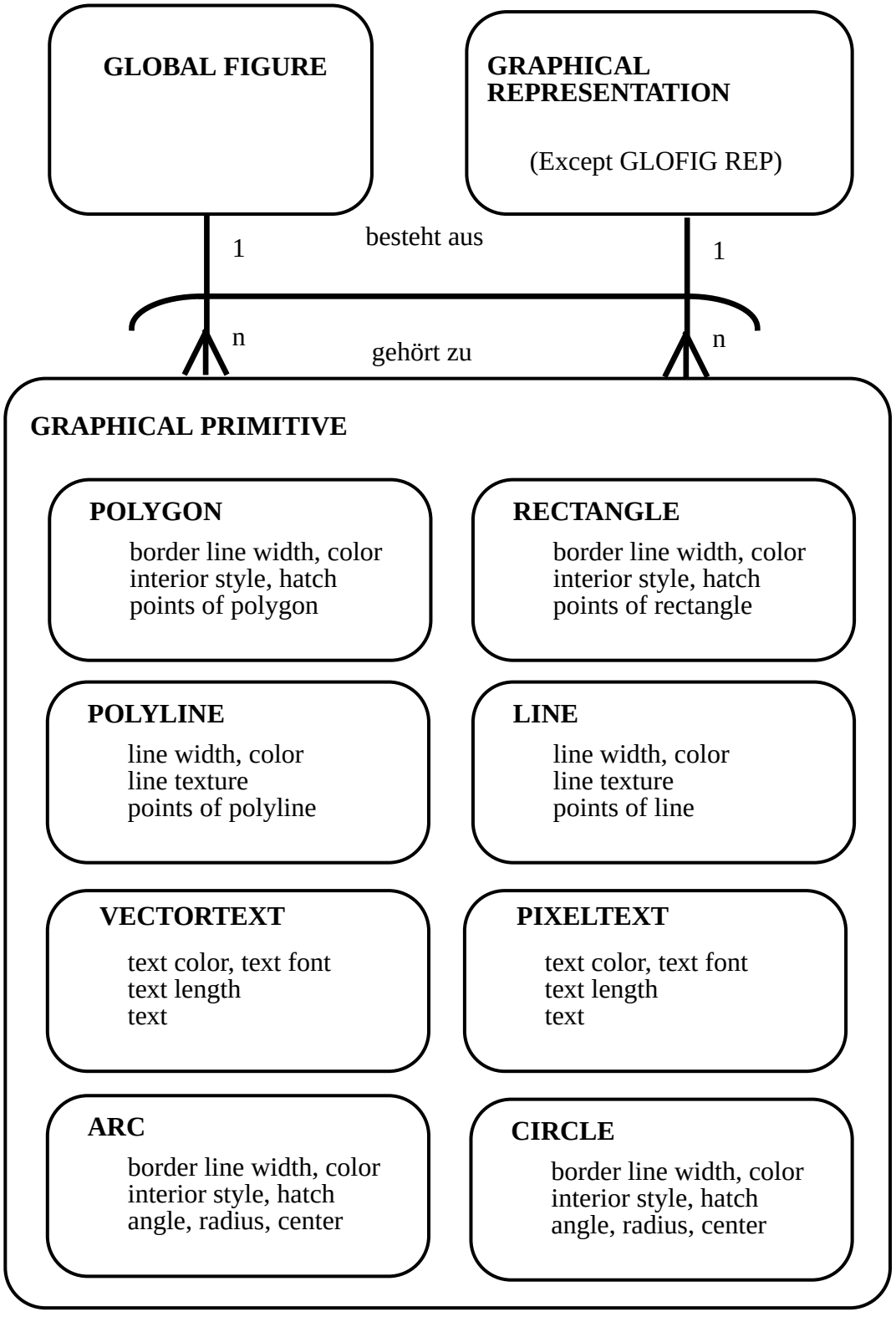
## 3.3 Entity-Relationship-Diagramm[1]




[1] Für die Darstellung des Entity-Relationship-Diagramm wird die Custom Development Method-Konvention benutzt, die in [Barker90] beschrieben ist. Gründe dafür sind die verständliche Einfachheit und der wenige Platzverbrauch.


SEGMENT
segmid, planid, wmapid
name
version
surrounding rectangle
1
besteht aus
n
gehört zu
OBJECT
objid, segmid, planid, wmapid
STATIC OBJECT
VARIABLE
MASK
datatype
boxlen, format no
attribute group
MESSAGE
processing mode
dec. table attr. grp.
dec. table fig. grp.
KEYBOX
coordinate displ.
boxtype
function key
VALUE
representation type
box length
value rep. desc
besitzt
lierfert
Werte für
n
LOGBOOK
rep. type
box length
value rep. desc
TIME
format number
attribute alternative
attribute group
m
VID-CURVE
dir. of x, y axis
range of y axis
ref. value of y axis
TEXT
textbox type
box length
attribute group
erhält
Werte von
gehört zu
besitzt / gehört zu
GRAPHICAL REPPRESENTATION
STATIC REP
VARIABLE REP


GRAPHICAL REPRESENTATION
STATIC REP
VARIABLE REP
SIMPLE REP
GLOFIG REP
color
GLOFIG REP
color
LOCFIG REP
color 1
color 2
CURVE REP
BAR REP
SIMPLE REP
1
1
1
1
1
1
1
1
1
1
GLOBAL FIGURE
version
refdist, refdots
surrounding rect
besteht aus
gehört zu
1
n
n
n
n
1
GRAPHICAL PRIMITIVE


GLOBAL FIGURE
GRAPHICAL REPRESENTATION
(Except GLOFIG REP)
1
besteht aus
1
n
gehört zu
n
GRAPHICAL PRIMITIVE
POLYGON
border line width, color
interior style, hatch
points of polygon
RECTANGLE
border line width, color
interior style, hatch
points of rectangle
POLYLINE
line width, color
line texture
points of polyline
LINE
line width, color
line texture
points of line
VECTORTEXT
text color, text font
text length
text
PIXELTEXT
text color, text font
text length
text
ARC
border line width, color
interior style, hatch
angle, radius, center
CIRCLE
border line width, color
interior style, hatch
angle, radius, center

## 3.4 Entitätsumsetzung

- Jede WORLDMAP besteht aus einer oder mehreren PLANEs.

  Jede PLANE muß zu einer einzigen WORLDMAP gehören.

  – WORLDMAP(**wmapid**, ...)
  – PLANE(**planid**, **wmapid**, ...)

- Jede PLANE besteht wiederum aus einem oder mehreren SEGMENTs.

  Jedes SEGMENT muß zu einer einzigen PLANE gehören.

  – SEGMENT(**segmid**, **planid**, **wmapid**, ...)

- Jedes SEGMENT enthält mehrere OBJECTs.

  Jedes OBJECT muß zu einem einzigen SEGMENT gehören.

  – OBJECT(**objid**, **segmid**, **planid**, **wmapid**, ...)

- Jedes OBJECT hat einen eindeutigen Typ. Es ist entweder ein STATIC OBJECT oder ein VARIABLE OBJECT. Die Exklusivität in der Entität OBJECT kann generisch modelliert werden, indem eine Referenzspalte zusammen mit einer zusätzlichen Spalte zur Kennzeichnung des Subtyps verwendet wird.

  – OBJECT(objid, segmid, planid, wmapid, ..., **objsubtype**, **objsubid**)

    wobei die *objsubtype*-Spalte zwei Werte haben kann, nämlich die Subtypen ***static*** oder ***variable***.

  – STATOBJECT(**statid**, ...)
  – VAROBJECT(**varid**, ...)

- Jedes DYNAMIC OBJECT hat einen eindeutigen Typ. Die möglichen Typen für ein DYNAMIC OBJECT sind CURVE, VID-CURVE, MASK, MESSAGE, KEYBOX, LOGBOOK, VALUE, TIME und TEXT. Wie oben kann man mit Hilfe einer Referenzspalte und einer Spalte, die den Typ kennzeichnet, die Entität DYNAMIC OBJECT in verschiedenen Relationen modellieren:

  – VAROBJECT(varid, **varsubtype**, **varsubid**)

    wobei die *varsubtype*-Spalte folgende Werte haben kann: ***curve, vcurve, mask, message, keybox, logbook, value, time*** und ***text***.

  – VARCU(**curvid**, ...)
  – VARVC(**curvid**, ...)
  – VARMA(**maskid**, ...)
  – VARME(**messid**, ...)
  – VARKE(**keyid**, ...)
  – VARLO(**logid**, ...)
  – VARVA(**valid**, ...)
  – VARTI(**timeid**, ...)
  – VARTE(**textid**, ...)

- Jedes VARIABLE OBJECT vom Typ *varvcurv* erhält Werte von mindestens zwei anderen VARIABLE OBJECTs vom Typ *logbook*.

  Jedes VARIABLE OBJECT vom Typ *logbook* kann Werte für verschiedene VARIABLE OBJECTs vom Typ *vcurve* liefern

  Das ist eine m:n-Beziehung. Um diese Beziehung zu modellieren, erstellt man eine neue Entität als Detail-Entität der zwei beteiligten Entitäten. Die m:n-Beziehung wird entsprechend in zwei m:1-Beziehungen aufgebrochen:

  – VARLO(logid, ...)
  – VARVC(curvid, ...)
  – VARVCLO(**curvid**, **logid**)

- Jedes STATIC OBJECT hat eine einzige STATIC GRAPHICAL REPRESENTATION

  Jede STATIC GRAPHICAL REPRESENTATION gehört zu einem einzigen STATIC OBJECT.

  – STATREP(**repid**, ...)
  – STATOBJECT(statid, ..., **statrepid**)

- Jedes VARIABLE OBJECT hat eine einzige VARIABLE GRAPHICAL REPRESENTATION

  Jede VARIABLE GRAPHICAL REPRESENTATION gehört zu einem einzigen VARIABLE OBJECT.

  – VARREP(**repid**, ...)
  – VAROBJECT(varid, ..., **varrepid**)

- Jede STATIC GRAPHICAL REPRESENTATION ist entweder eine GLOBAL FIGURE oder eine SIMPLE REPRESENTATION.

  – STATREP(repid, **statrepsubtype**, **statrepsubid**)

    wobei die *statrepsubtype*-Spalte zwei folgende Werte haben kann: ***glofig*** und ***simple***.

  – SREPGLOFIG(**figid**, ...)
  – SREPSIMPLE(**simpid**, ...)

- Jede VARIABLE GRAPHICAL REPRESENTATION hat auch einen eindeutigen Typ. Die möglichen Typen dafür sind GLOBAL FIGURE, LOCAL FIGURE, BAR, CURVE und SIMPLE REPRESENTATION.

  – VARREP(repid, **varrepsubtype**, **varrepsubid**)

    wobei die *varrepsubtype*-Spalte folgende Werte haben kann: ***glofig, locfig, bar, curve*** und ***simple***.

  – VREPGLOFIG(**figid**, ...)
  – VREPLOCFIG(**figid**, ...)
  – VREPBAR(**barid**, ...)
  – VREPCURVE(**curvid**, ...)
  – VREPSIMPLE(**simpid**, ...)

- Jede GRAPHICAL REPRESENTATION vom Typ *glofig* nimmt die Form einer GLOBAL FIGURE an.

  Jede GLOBAL FIGURE kann die Darstellung für mehrere GRAPHICAL REPRESENTATION vom Typ *glofig* liefern.

  – GLOFIG(**glofigid**, version, refdots, refdist, ...)
  – SREPGLOFIG(figid, color, **glofigid**)
  – VREPGLOFIG(figid, color, **glofigid**)

- Jede GRAPHICAL REPRESENTATION (nicht vom Typ *glofig*) besteht aus 1 oder mehreren GRAPHICAL PRIMITIVEs.

  Jedes GRAPHICAL PRIMITIVE muß entweder zu einer GRAPHICAL REPRESENTATION (nicht vom Typ *glofig*) oder zu einer GLOBAL FIGURE gehören.

  Die exclusive Beziehung zwischen GRAPHICAL PRIMITIVE und GRAPHICAL REPRESENTATION bzw. den REPRESENTATION-Subtypen und GLOBAL FIGURE kann wie folgt modelliert werden:

  – PRIMITIVE(**primid**, ...)
  – GLOFIG(glofigid, version, ..., **primid**)
  – SREPSIMPLE(simpid, ..., **primid**)
  – VREPLOCFIG(figid, color1, color2, ..., **primid**)
  – VREPBAR(barid, ..., **primid**)
  – VREPCURVE(curvid, ..., **primid**)
  – VREPSIMPLE(simpid, ..., **primid**)

- Jedes GRAPHICAL PRIMITIVE hat einen eindeutigen Typ. Die möglichen Typen für ein GRAPHICAL PRIMITIVE sind POLYLINE, LINE, POLYGON, RECTANGLE, VECTORTEXT, PIXELTEXT, ARC oder CIRCLE.

  – PRIMITIVE(primid, **primsubtype**, **primsubid**)

    wobei die *primsubtype*-Spalte acht folgende Werte haben kann, nämlich die Subtypen ***polyline, line, polygon, rectangle, vectortext, pixeltext, arc*** oder ***circle***.

  – PLINE(**plinid**, ...)
  – LINE(**lineid**, ...)
  – PGON(**pgonid**, ...)
  – RECT(**rectid**, ...)
  – VTEXT(**textid**, ...)
  – PTEXT(**textid**, ...)
  – ARC(**arcid**, ...)
  – CIRCLE(**circid**, ...)

## 3.5 Schemavalidierung

### 3.5.1 Analyse

Zur Übersicht sind alle Relationen hier nochmals aufgeführt, die durch die Modellierung im letzten Abschnitt hervorgegangen sind:

Tabelle 2 Umgesetztes Datenschema

| Relation | Primary Key | Foreign Key | Data |
|---|---|---|---|
| *WORLDMAP* | *wmapid* | | *name, crange, refdots, refdist, vrp, ...* |
| *PLANE* | *planid, wmapid* | *wmapid* | *name, zoom range, ...* |
| *SEGMENT* | *segmid, planid, wmapid* | *planid, wmapid* | *name, version, comment, ...* |
| *OBJECT* | *objid, segmid, planid, wmapid* | *segmid, planid, wmapid,* | *objsubtype, objsubid, surrounding rect, location, ...* |
| *VAROBJECT* | *varid* | *varrepid* | *varsubtype, varsubid* |
| *VARMA* | *maskid* | | *datatype, attrgr, attral, info, ...* |
| *VARME* | *messid* | | *promode, dectab, attrgr, attral, ta, ...* |
| *VARKE* | *keyid* | | *keyboxtype, keytype, datatype, ...* |
| *VARLO* | *logid* | | *reptype, boxlen, repctrl, ta, ...* |
| *VARVA* | *valid* | | *reptype, boxlen, repctrl, ta, ...* |
| *VARTE* | *textid* | | *textboxtype, boxlen, attrgr, attral, ...* |
| *VARTI* | *timeid* | | *formatno, attrgr, attral, unit, ...* |
| *VARCU* | *curvid* | | *valsrc, zeropart, ta, axis, style, ...* |
| *VARVC* | *curvid* | | *axis dir, style, y-axis ranges, ...* |
| *VARVCLO* | *curvid, logid* | *logid* | |
| *VARREP* | *repid* | | *varrepsubtype, varrepsubid* |
| *GLOFIG* | *glofigid* | *primid* | *version, refdots, refdist, surr rect, ...* |
| *VREPGLOFIG* | *figid* | *glofigid* | *color, ...* |
| *VREPLOCFIG* | *figid* | *primid* | *color1, color2, ...* |
| *VREPBAR* | *barid* | *primid* | |
| *VREPCURVE* | *curvid* | *primid* | |
| *VREPSIMPLE* | *simpid* | *primid* | |
| *STATOBJECT* | *statid* | *statrepid* | |
| *STATREP* | *repid* | | *statrepsubtype, statrepsubid* |
| *SREPGLOFIG* | *figid* | *glofigid* | *color, ...* |
| *SREPSIMPLE* | *simpid* | *primid* | |
| *PRIMITIVE* | *primid* | | *primsubtype, primsubid* |
| *PLINE* | *plinid* | | *color, width, texture, points, ...* |
| *LINE* | *lineid* | | *color, width, texture, points, ...* |
| *PGON* | *pgonid* | | *color, border, style, fill, points, ...* |
| *RECT* | *rectid* | | *color, border, style, fill, points, ...* |

| Relation | Primary Key | Foreign Key | Data |
|---|---|---|---|
| *VTEXT* | *textid* | | *color, font, height, upvect, text, ...* |
| *PTEXT* | *textid* | | *color, font, height, text, ...* |
| *ARC* | *arcid* | | *color, width, texture, center, rad, ...* |
| *CIRCLE* | *circid* | | *color, border, fill, center, radius, ...* |

Um das Schema noch zu verfeinern und bzw. zu prüfen, ob es eventuell noch Redundanz in Daten vorkommt, ist nun zu versuchen, es zu normalisieren. ([Elmasri94], Normal Forms Based on Primary Keys, Seite 407ff., [Vossen94], Normalisierung und algorithmischer Schema-Entwurf, Seite 249ff.)

Betrachtet man die Relation GLOFIG der Globalfiguren, erkennt man, daß die Attribute *version*, *refdots*, *refdist*, ... eine Wiederholgruppe bilden, wenn diejenige GLOFIG aus mehreren PRIMITIVEs bestehen. In den Relationen der Graphikrepräsentationsformen (SREP- bzw. VREP-Relationen) existieren auch noch Wiederholgruppen, nämlich wenn eine Graphikrepräsentation aus mehreren Graphikprimitiven bestehen. Beispiel dafür sind die Attribute *color1*, *color2* in VREPLOCFIG. Die Wiederholgruppen werden in folgende zusätzliche Relationen bewegt.

| Relation | Primary Key | Foreign Key | Data |
|---|---|---|---|
| *GLOFIGDES* | *glofigid* | | *version, refdots, refdist, rect, ...* |
| *GLOFIG* | *glofigid, primid* | *glofigid, primid* | |
| *LOCFIG* | *figid* | | *color1, color2* |
| *VREPLOCFIG* | *figid, primid* | *figid, primid* | |

In den Relationen existieren keine Wiederholgruppen mehr. Die Werte, die die einzelnen Attribute annehmen können, sind einfach und nicht mehr teilbar. Also befindet sich das Datenschema in der 1. Normalform.

Aus den Relationen geht noch hervor, daß jedes Nicht-Primärattribut voll funktional von jedem Schüsselkandidat der zugehörigen Relation abhängt. Das Datenschema befindet sich somit in der 2. Normalform.

Es existieren keine transitiven Abhängigkeiten der Nicht-Primärattribute von den Schlüsselkandidaten in den Relationen. Die Schlüssel der Relationen besteht meistens aus einem Kandidat. In den Relationen mit mehrwertigen Schlüsseln wie z.B. SEGMENT, OBJECT hängen alle Nicht-Primärattribute nur vom Schüssel der Relation ab. Das Datenschema befindet sich auch in der 3. Normalform.

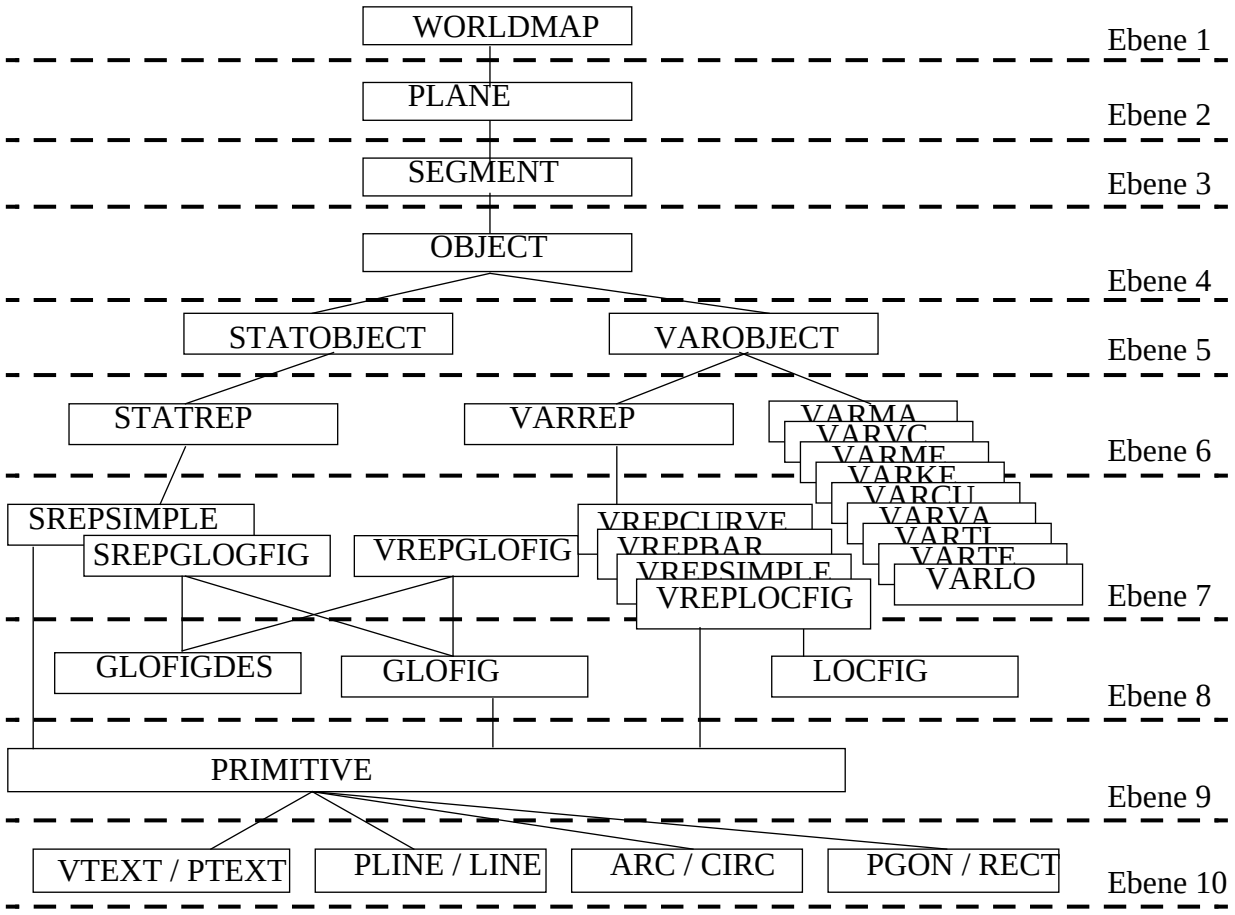
Dieses Datenschema ist aber noch nicht in der Boyce Codd-Normalform, da Relationen mit mehrwertigen Schlüsseln wie z.B. OBJECT noch Abhängigkeiten zwischen Schlüsselkandidaten aufweisen. Die Objekt-Nummern sind nur eindeutig in einem Segment, die Segment-Nummern wiederum in einer Plane und die Plane-Nummern nur eindeutig in einer Worldmap. Die Überführung des Datenschemas in die Boyce Codd-Normalform kann man dadurch erreichen, die Nummern der Objekte, Segmente, Planes als einwertige Primärschlüssel für die entsprechenden Relationen zu definieren:

- WORLDMAP(wmapid, ...)
- PLANE(planid, wmapid, ...)
- SEGMENT(segmid, planid, ...)
- OBJECT(objid, segmid, ...)

Es ist jedoch nicht angebracht, da diese Zuordnungen nur nach mehreren Lesezugriffen herzustellen sind.

Ein Datenschema ist nun für die gegebene Miniwelt entwickelt worden. Die Daten werden sortiert und in einzelnen Entitäten gruppiert. Es enthält idealerweise keine Datenredundanz. Die Beziehungen der Entitäten untereinander sind eindeutig durch Primärschlüssel bzw. durch Fremdschlüssel bestimmt. Es ist jedoch unverwendbar in der Realität. Der einzige Grund hierfür ist, daß es einen zu hohen referentiellen Grad aufweist. Wir haben das Granularitätsprinzip radikal verfolgt und die Worldmaps bis auf kleinste Entitäten zerlegt.

Abbildung 9 Zugriffspfad in der Wordmap-Hierarchie




Die Kosten, um ein Worldmap zu laden, werden in folgende Schritte aufgeteilt:

- Daten zur aktuellen Worldmap (WORLDMAP) : +1 Zugriff / Worldmap
- Daten zu Planes (PLANE) : +1 Zugriff / Plane
- Daten zu Segmenten (SEGMENT) : +1 Zugriff / Segment
- Daten zu Objekten (OBJECT) : +1 Zugriff / Objekt
- Objekt ist *static* oder *variable* (STATOBJECT/VAROBJECT) : +1 Zugriff / Objekt
- Typ/Nummer für Graphikrepräsentation (STATREP/VARREP) : +1 Zugriff / Objekt

- Global-/Lokalfigur als Graphikrep. (GLOFIGDES/LOCFIG) : +1 Zugriff / Figur
- Nummer für Graphikprimitiv (SREP-/VREP-/GLOFIG) : +1 Zugriff / Primitiv
- Typ/Nummer für Graphikprimitiv (PRIMITIVE) : +1 Zugriff / Primitiv
- Daten zu Graphikprimitiv (PLINE/PGON/TEXT/ARC) : +1 Zugriff / Primitiv
- Daten zu Variablenbeschreibung (VARMA/VARME/...) : +1 Zugriff / Variable

Um eine Worldmap auf dem Bildschirm darzustellen, müssen alle Graphikprimitive geladen werden und danach gezeichnet werden. In unserem Schema ist es erst nach 9 -10 Ladestufen bzw. Datenbankzugriffen für einzelne Graphikprimitive möglich, was die Ladezeit der gesamten Worldmap in die Länge zieht. Der Lesevorgang mittels Table-Joins ist in diesem Schema nicht überall verwendbar, da die Subtypen einer Entität mittels einer Numerierungsspalte (Identifier) und einer Typ kennzeichnenden Spalte in der Entität referenziert werden.

Ein zu hoher Granularitätsgrad weist noch einen anderen Nachteil auf. Die Konsistenz der Datenbank und insbesondere die Konsistenz zwischen dem Supertyp und seinen Subtypen müssen mit Hilfe von Trigger realisiert werden, da Fremdschlüssel für diese Beziehungen nicht geeignet sind. Erstens ist die Ausführung von Trigger deutlich langsamer als eine Prüfung von tabelleninternen Constraints. Zweitens würde eine datenmanipulierende Operation (z.B. DELETE bzw. INSERT) ein kaskadierendes Aufrufen von Triggern in der Datenbank auslösen, was wiederum viel Zeit kostet, wenn es tiefer gehen muß.

### 3.5.2 Optimierung

Ziel dieses Abschnitts ist nun ein anderes Schema zu gestalten, das auf dem ursprünglichen Schema basiert und unnötige Referenzierungen vermeidet, um einen geringeren Granulariätsgrad zu erreichen. Dies geschieht hauptsächlich durch Kopieren von Attributen der referenzierten Entitäten in die referenzierende Entität. Als Folge werden zwar die Superentitäten komplexer, aber als Endergebnis erhält man ein Schema mit weniger Entitäten, mit weniger Referenzierungen, was auch weniger Datenbankzugriffe bedeutet. Es wird von Entitäten unterster Stufe herauf untersucht.

Die Entitäten, die keine anderen Entitäten referenzieren, können wir außer Acht lassen. Dies betrifft die Graphikprimitiv-Entitäten PLINE, LINE, PGON, RECT, VTEXT, PTEXT, ARC und CIRC sowie die Variablenbeschreibungs-Entitäten VARMA, VARME, VARKE, VARLO, ...

Die Entität PRIMITIVE dient zur Identifizierung von einzelnen Graphikprimitiven, die in verschieden Graphikrepräsentationsformen vorkommen, und hat keine zusätzlichen Attribute. Wenn wir diese Entität auflösen, so kommen die Attribute der Typbezeichnung und der Numerierung in die Relationen der Graphikrepräsentationsformen:

- GLOFIG(glofigid, **primsubtype**, **primsubid**)
- SREPSIMPLE(simid, **primsubtype**, **primsubid**)
- VREPLOCFIG(figid, **primsubtype**, **primsubid**)
- VREPBAR(barid, **primsubtype**, **primsubid**)
- VREPCURVE(curvid, **primsubtype**, **primsubid**)
- VREPSIMPLE(simpid, **primsubtype**, **primsubid**)

Die verschiedenen Formen von Graphikrepräsentation für STATOBJECT und VAROBJECT werden vereinigt. Es wird nicht mehr unterschieden, welche Repräsentationsformen ein statisches Objekt und welche ein dynamisches Objekt haben kann. Nun stehen folgende Formen zur Darstellung der Objekte im allgemeinen:

- REPGLOFIG(figid, glofigid, color)
- REPLOCFIG(figid, primsubtype, primsubid)
- REPBAR(barid, primsubtype, primsubid)
- REPCURVE(curvid, primsubtype, primsubid)
- REPSIMPLE(simpid, primsubtype, primsubid)

Betrachten wir die Superentitäten von den obigen Entitäten: STATREP und VARREP. Sie enthalten Information zur Identifizierung von verschiedenen Graphikrepräsentationen, die einem STATOBJECT oder einem VAROBJECT zugehören. Lösen wir die zwei Relationen STATREP und VARREP auf, so erhalten wir:

- STATOBJECT(statid, **repsubtype**, **repsubid**)
- VAROBJECT(varid, varsubtype, varsubid, **repsubtype**, **repsubid**)

VAROBJECT und STATOBJECT enthalten Informantion zur Identifizierung der Variablenbeschreibung und Graphikrepräsentation eines OBJECTs. Diese Information kann in die Relation OBJECT aufgenommen werden. Die Spalten zur Typbezeichnung eines OBJECTs selbst (*static* oder *variable*) und zu dessen Identifizierung in VAROBJECT bzw. in STATOBJECT entfallen. Es wird nicht mehr zwischen Formen der Graphikrepräsentation für STATOBJECT und für VAROBJECT unterschieden.

- OBJECT(objid, segmid, planid, wmapid, ..., **repsubtype**, **repsubid**, **varsubtype**, **varsubid**)

  wobei die *repsubtype*-Spalte die Werte: ***glofig, locfig, bar, curve*** und ***simple*** und die *varsubtype*-Spalte die Werte: ***curve, vcurve, mask, message, keybox, logbook, value, time*** und ***text*** haben können.

Es ist nun deutlich hervorgegangen, daß die Objekte die untersten bedeutungstragenden Instanzen innerhalb der Worldmap-Hierarchie sind. Zu einem Objekt gehören eine obligatorische Graphikrepräsentation und eine Variablenbeschreibung, wenn das Objekt ein dynamisches ist.

Hinter den SIMPLE-Graphikrepräsentationen stehen verschiedene Formen der Graphikprimitive, wie Polygon, Polyline, Text und Arc. Um die Graphikprimitive, die diese Repräsentationsformen darstellen, laden zu können, muß man erst von der Relation REPSIMPLE ihre Referenzierung (Typ und Nummer) herauslesen. Betrachtet man die Typen der Graphikprimitive gleichwertig wie andere Repräsentationsformen der Objekte, kann man den Ladevorgang um einen Schritt vereinfachen, nämlich durch die folgende Änderung:

- OBJECT(objid, segmid, planid, wmapid, ..., **repsubtype**, **repsubid**, varsubtype, varsubid)

  wobei die *repsubtype*-Spalte die Werte: *glofig*, *locfig*, *bar*, *curve*, ***polyline, line, polygon, rectangle, vektortext, pixeltext, arc*** und ***circ*** haben kann.

Die Spalten *repsubtype* und *repsubid* referenzieren dann direkt die Graphikprimitive, die zur Darstellung des Objekts benötigt werden, wenn der Typ der Repräsentation nicht *glofig*, *locfig*, *bar* oder *curve* ist. Dadurch entfällt die Relation REPSIMPLE.

Die Relationen VREPGLOFIG und LOCFIG enthalten einige zusätzliche Informationen über die Global- und Lokalfiguren. Interessant sind jetzt aber nur die Attribute *color* für Globalfiguren bzw. *color1*, *color2* für Lokalfiguren, welche einmalig sind für jedes Objekt mit der Graphikrepräsentation einer Globalfigur bzw. einer Lokalfigur. Aus diesem Grund kann man nun diese Attribute in die Relation OBJECT einführen, um unnötige Datenbankzugriffe zu vermeiden.

– OBJECT(objid, segmid, planid, wmapid, ..., **color1**, **color2**)

Dadurch entfallen die Relationen VREPGLOFIG und LOCFIG. Die Spalten *repsubtype* und *repsubid* referenzieren direkt die Figuren.

Die Struktur der BAR- bzw. CURVE-Graphikrepräsentationen sind im Unterschied zu den Global- und Lokalfiguren immer gleich, auch wenn sie aus mehreren Graphikprimitiven bestehen. Eine BAR-Repräsentation besteht immer aus zwei Rechtecken. Eine CURVE-Repräsentation besteht aus einem Rechteck (bzw. zwei Achsenlinien) und einem Linienzug, die aber erst im Online-System gezeichnet wird. Aus diesem Grund werden einfach die Graphikprimitive, die zu einer BAR- bzw. einer CURVE-Repräsentation gehören, aufgelöst und in einer anderen Form in den Relationen BAR und CURVE abgelegt.

Aufgrund der Ähnlichkeiten zwischen den Graphikprimitiven Polyline und Line, Polygon und Rectangle, Vektortext und Pixeltext, Arc und Circle werden sie jeweils in einer Relation zusammengefaßt.

Ein implementierbares Schema geht als Ergebnis der obigen Analyse und Änderungen hervor:

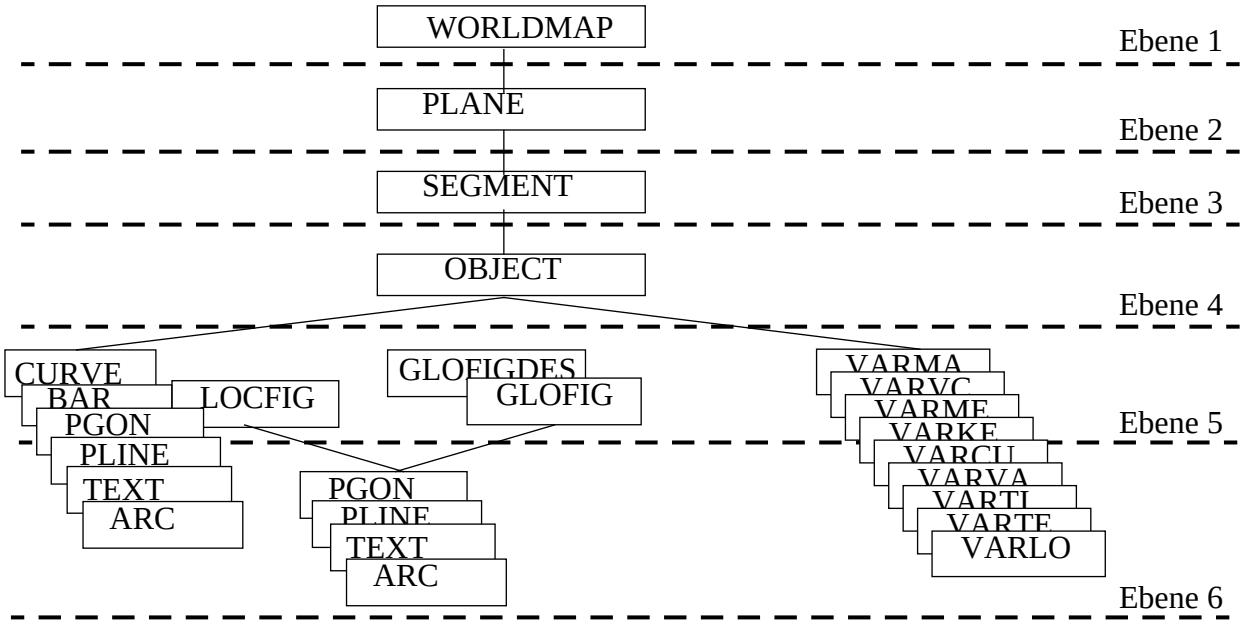
Tabelle 3 Optimiertes Datenschema

| Relation | Primary Key | Foreign Key | Data |
|---|---|---|---|
| *WORLDMAP* | *wmapid* | | *name, crange, refdots, refdist, vrp, ...* |
| *PLANE* | *planid, wmapid* | *wmapid* | *name, zoom range, ...* |
| *SEGMENT* | *segmid, planid, wmapid* | *planid, wmapid* | *name, version, comment, ...* |
| *OBJECT* | *objid, segmid, planid, wmapid* | *segmid, planid, wmapid,* | *reptype, repid, vartype, varid, color, color2, location, surr rect, ...* |
| *VARMA* | *maskid* | | *datatype, attrgr, attral, info, ...* |
| *VARME* | *messid* | | *promode, dectab, attrgr, attral, ta, ...* |
| *VARKE* | *keyid* | | *keyboxtype, keytype, datatype, ...* |
| *VARLO* | *logid* | | *reptype, boxlen, repctrl, ta, ...* |
| *VARVA* | *valid* | | *reptype, boxlen, repctrl, ta, ...* |
| *VARTE* | *textid* | | *textboxtype, boxlen, attrgr, attral, ...* |
| *VARTI* | *timeid* | | *formatno, attrgr, attral, unit, ...* |
| *VARCU* | *curvid* | | *valsrc, zeropart, ta, axis, style, ...* |
| *VARVC* | *curvid* | | *axis dir, style, y-axis ranges, ...* |
| *VARVCLO* | *curvid, logid* | *logid* | |
| *GLOFIGDES* | *figid* | | *version, refdots, refdist, surr rect, ...* |
| *GLOFIG* | | *figid* | *primtype, primid* |
| *LOCFIG* | *figid* | | *primtype, primid* |
| *BAR* | *barid* | | *fore/back color, back/slider rect, ...* |

| Relation | Primary Key | Foreign Key | Data |
|---|---|---|---|
| *CURVE* | *curvid* | | *lowerleft corner, witdth, height, ...* |
| *PLINE* | *plinid* | | *color, width, texture, points, ...* |
| *PGON* | *pgonid* | | *color, border, style, fill, points, ...* |
| *VTEXT* | *textid* | | *color, font, height, upvect, text, ...* |
| *ARC* | *arcid* | | *color, width, texture, center, rad, ...* |

Der Zugriff auf die Worldmap-Entitäten kann wie folgt modelliert werden:

Abbildung 10 Zugriffspfad nach der 1. Optimierung




Dieses Datenschema wird in den nächsten Abschnitten als das erste Schema angesprochen. (Für vollständige Aufzählung der Entitäten und deren Attribute siehe Anhang B.1 ***Das erste Datenschema,*** Seite 96)

### 3.5.3 Weitere Möglichkeiten

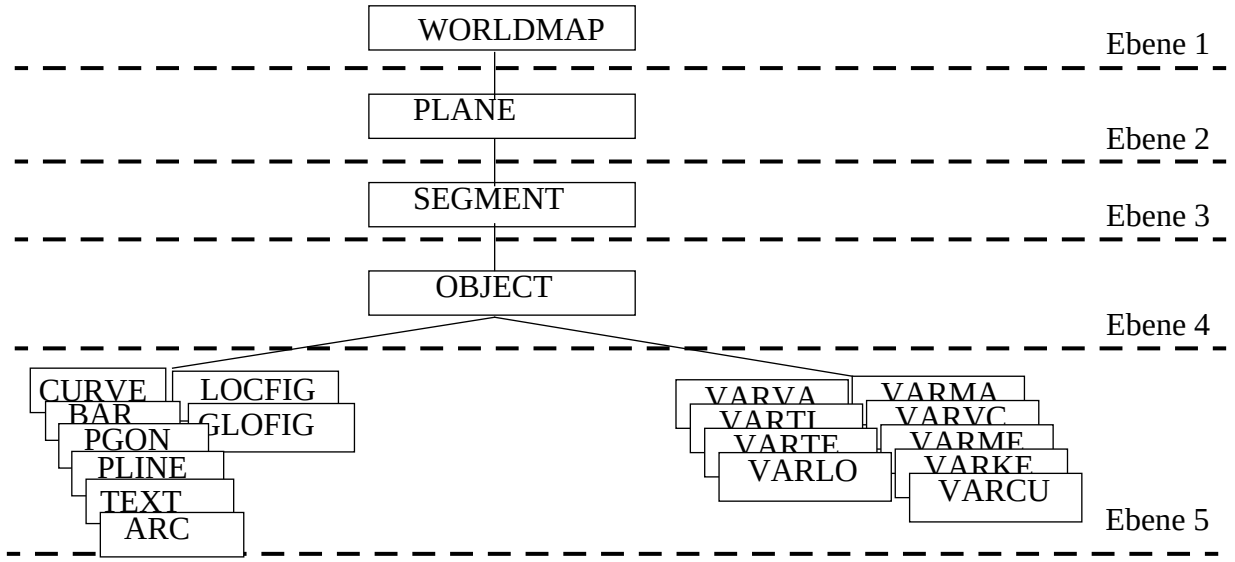
Am Zugriffspfad kann man sehr gut verschiedene Optimierungsstrategien illustrieren. Ein Schema ist besser für Performanz, wenn die Operationen, die auf dessen Daten zugreifen, nicht so tief in die Hierarchie der Entitäten gehen müssen.

Das optimierte Schema zeichnet sich durch eine Verringerung der Referenzebenen aus (jetzt nur noch 6 im Vergleich zu 10 in Abbildung 9). Da die Objekte die kleinsten bedeutungstragenden Einheiten in der Worldmap-Hierarchie sind und ein Sperrmechanismus auf einzelne Objekte später unterstützt werden muß, dürfen die Optimierungsverfahren die Grenze der Ebene 4 nicht überschreiten.

Da die Variablenbeschreibungen eine gewisse Komplexität bezüglich der enthaltenen Informationen aufweisen, ist es nicht ratsam, sie in die Beschreibung von einzelnen Objekten aufzunehmen. Wir setzen dementsprechend die Ebene 5 als die letzte Grenze für Optimierungen fest.

Die Referenzebene 6 ist nun die einzige Ebene, die noch vermieden werden kann. In dieser Ebene werden die Graphikprimitive der Global- bzw. Lokalfiguren aus der Datenbank gelesen. Um sie schon früher laden zu können, muß man die Graphikprimitive, die einer Global- oder Lokalfigur angehören, in die Beschreibung derjenigen Figur mit aufnehmen. Da der Aufbau einzelner Figuren beliebig ist, müssen die Graphikprimitive einer Figur nacheinander in ein LONG RAW-Feld gepackt und als eine Einheit zusammen mit ihrer Beschreibung gespeichert werden. Der Zugriffspfad vereinfacht sich zu:

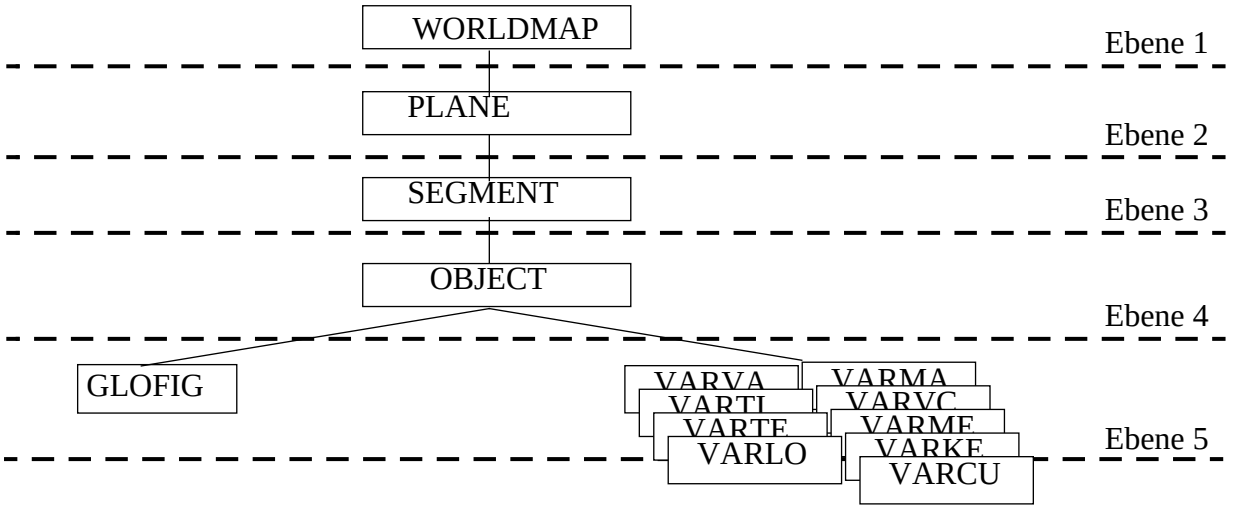
Abbildung 11 Zugriffspfad nach der 2. Optimierung




Dieses Datenschema wird in den nächsten Abschnitten als das zweite Schema angesprochen. (Für konkrete Änderungen im Datenschema siehe Anhang B.2 ***Das zweite Datenschema,*** Seite 105)

Eine weitere Optimierungsmöglichkeit wäre, die obige Methode noch weiter zu verfolgen und jede Graphikbeschreibung für ein Objekt in einem LONG RAW-Feld zu fassen und zusammen mit der Objektbeschreibung zu speichern, wenn das Objekt keine Globalfigur als seine Graphikrepräsentation hat. Dadurch verringert man nicht die Anzahl der Referenzebenen im Schema, sondern die Anzahl der Zugriffe, die beim Laden einer Worldmap benötigt werden. Auf der Ebene 5 bleiben nur noch die Zugriffe auf die Tabellen der Variablenbeschreibungen oder auf die Tabelle der Globalfiguren:

Abbildung 12 Zugriffspfad nach der 3. Optimierung




Dieses Datenschema wird in den nächsten Abschnitten als das dritte Datenschema angesprochen. (Für konkrete Änderungen im Datenschema siehe Anhang B.3 ***Das dritte Datenschema,*** Seite 106)

# 4 Performanzuntersuchungen

## 4.1 Datenbankkonzept von ORACLE

Ein Datenbankserver ist der Schlüssel zur Lösung verschiedener Aufgaben im Bereich der Informationsverwaltung. Er muß in der Lage sein, große Ansammlung von Daten zuverlässig in einer Mehrnutzerumgebung zu verwalten. Er muß über Mechanismen verfügen, um Zugriffe auf die Datenbank zu kontrollieren und den Datenbestand im Fehlerfall retten zu können. Dabei muß eine hohe Performanz erreicht werden. Der ORACLE-Server ist ein relationales Datenbankmanagement-System. Im nächsten Abschnitt wird untersucht, wie und in wie weit die Architektur eines ORACLE-Datenbanksystems auf die Performanz der Transaktionen ausgerichtet ist. Als Resultat dieser Analyse sollen die Kenntnisse darüber sein, welche Möglichkeiten sich für die Optimierung der Performanz der Applikationen anbieten.

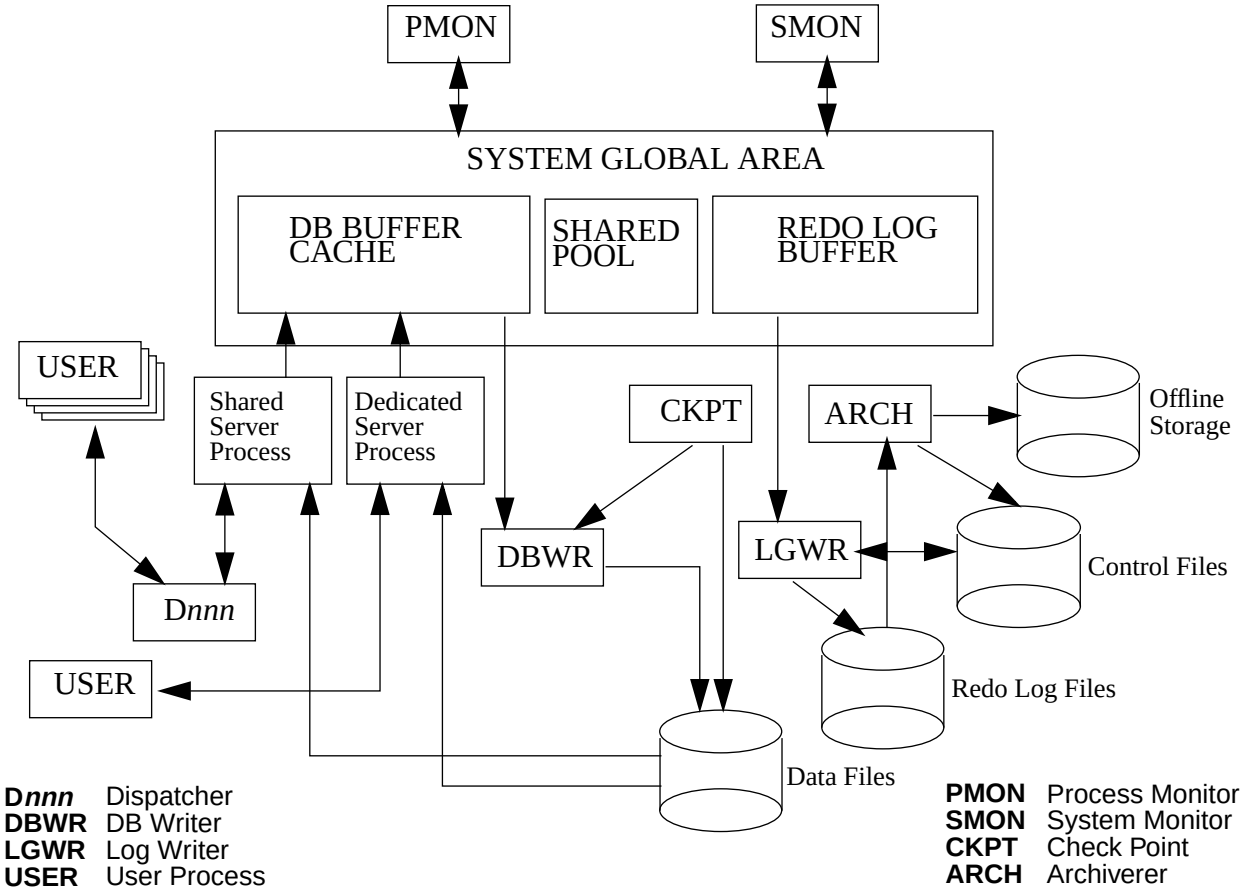
### 4.1.1 Datenbankarchitektur

(Literatur zu diesem Abschnitt: [OraSerCon96], [OraDisSys96])

Ein arbeitsbereites ORACLE-Datenbanksystem besteht aus zwei Teilen. Der eine ist die passive Datenbank, wo sich die eigentlichen Daten befinden. Der andere ist die sog. ORACLE-Instanz, der aktive Teil des Datenbanksystems, der die Aktionen ausführen, die mit den gespeicherten Daten im Zusammenhang stehen. Die ORACLE-Instanz umfaßt die allokierten Puffer und alle Prozesse, sowohl die Serverprozesse, die von ORACLE erzeugt werden, um Anfragen von Benutzerprozessen zu bearbeiten, als auch die Hintergrundsprozesse, die die Datenbank auf unterschiedliche Art und Weise pflegen. Die Benutzerprozesse, die normalerweise auf einer entfernten Maschine gestartet werden, müssen mit dem Server über SQL*Net kommunizieren.

Durch die Trennung der ORACLE-Instanz als verwaltenden Einheit von der Datenbank als verwalteten Einheit bietet ORACLE eine Vielfalt von Systemkonfigurationen an. Eine typische Systemkonfiguration ist die sog. *Single Instance*-Konfiguration. Hier verwaltet eine einzige ORACLE-Instanz die ganze Datenbank. In der Regel wird eine gut ausgestattete Maschine ausgewählt für den Datenbankserver. Man kann mit der Single Instance-Konfiguration gute Performanz erreichen, weil die gesamte Rechenkapazität der Anlage für den Datenzugriff und nicht für Koordinationsaktivitäten zwischen verschiedenen Knoten genutzt wird. Wenn mehrere ORACLE-Instanzen dieselbe Datenbank verwalten, nennt man eine *Multi Instance*-Konfiguration. In dieser Konfiguration wird eine verteilte Sperreverwaltung (distributed lock manager, DLM) benötigt, um Prozesse einzelner Instanzen, die um Daten konkurrieren, miteinander zu synchronisieren. Mit dieser Konfiguration kann der Aufwand dafür reduziert werden, Daten zwischen Instanzen auf verschiedenen Knoten zu transferieren. Mehrere Server und Datenbanken können miteinander verbunden werden, um ein verteiltes Datenbanksystem zu bilden. Jede Datenbank wird von einer Instanz verwaltet. Der Zugriff auf den Datenbestand wird anderen Instanzen mittels der Instanz-Instanz-Kommunikation gewährleistet.

Abbildung 13 Prozesse und Speicher einer ORACLE-Instanz




Beim Hochlaufen allokiert ORACLE einen globalen Shared Memory-Bereich (System Global Area, SGA) für die Daten und Kontrollinformationen. Jede ORACLE-Instanz besitzt einen eigenen SGA, der erst deallokiert wird, wenn die Instanz untergefahren wird. Die ORACLE-Prozesse teilen sich die Daten im SGA. Für eine bessere Performanz soll der gesamte Bereich von SGA so groß wie möglich,

um viel Daten aufnehmen zu können. Dadurch werden Zugriffe auf Hintergrundsspeicher vermieden. Für ein System mit wenig Hauptspeicher muß man die Größe für SGA genau kalkulieren, da ein zu großer SGA dem Betriebssystem viel Hauptspeicher wegnehmen wird, was Swapping und Paging im Gesamtsystem verursachen kann. Die Größe für einzelne Bereiche des SGAs kann man mit Initiationsparametern in der Startdatei INIT.ORA festlegen (siehe 4.1.2 ***Optimierungsmöglichkeiten des Servers,*** Seite 42).

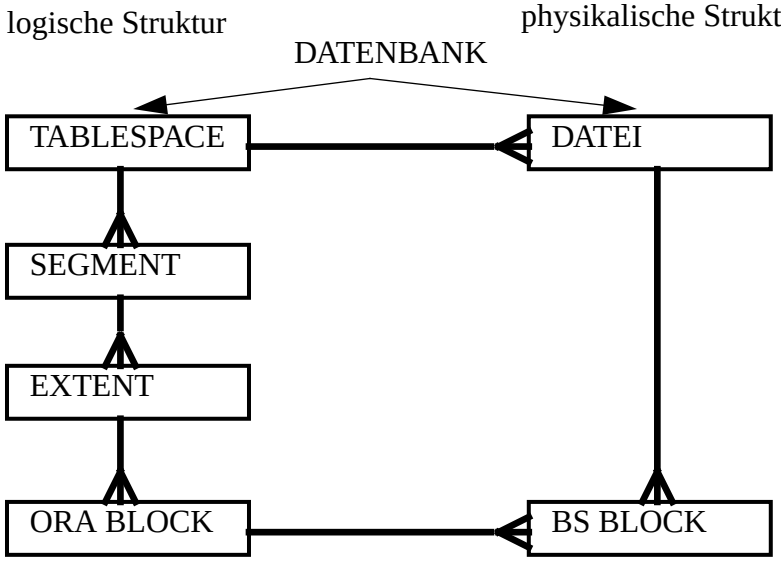
Für jede ORACLE-Instanz existieren bestimmte Hintergrundsprozesse, die die Datenbank auf unterschiedlichste Weisen pflegen. Der DBWR-Prozeß schreibt modifizierte Blöcke im Database Cache-Puffer in die Datendateien und ist allein verantwortlich für die Verwaltung des Puffers. Wenn Platz im Puffer für einen neuen Datenblock benötigt wird, entscheidet der DBWR-Prozeß nach dem LRU-Algorithmus (Least Recently Used) und lädt einen am längsten nicht gebrauchten Block aus. Der LGWR-Prozeß schreibt Redo-Informationen in die Redo Log-Dateien und ist zuständig für die Verwaltung des Redo Log-Puffers. Die Redo Log-Dateien enthalten dann alle Änderungen, die mit dem Datenbestand gemacht werden, um eine Restaurierung zu ermöglichen. Der CKPT-Prozeß macht die sog. Zeitstempel auf Headers der Datendateien zu den Checkpoints, weckt den DBWR-Prozeß und übergibt ihm die Liste der zu schreibenden Blöcke vom Database Cache-Puffer. Der LWGR-Prozeß kann diesen Job auch erledigen, wenn der CKPT-Prozeß nicht aktiviert wird. Der SMON-Prozeß restauriert in erster Linie die Datenbank, wenn die Instanz hochgefahren wird. Außerdem kümmert er sich noch darum, nicht mehr benutzte temporäre Segmente freizugeben. Der PMON-Prozeß restauriert dagegen die Datenbank, wenn ein Benutzerprozeß mit einem Transaktionsfehler beendet wird. Dabei werden z.B. Systemresourcen freigegeben, Sperre aufgehoben, Datenänderungen rückgängig gemacht.

Die Arbeitsweise der Serverprozesse kann durch verschiedene Konfigurationen bestimmt werden. Ein *dedicated* konfigurierter Serverprozeß bearbeitet nur die Anfragen von einem Benutzerprozeß, während ein *shared* (multi-threaded) konfigurierter Serverprozeß Anfragen von mehreren Benutzerprozessen mit Hilfe eines Dispatcher-Prozesses bearbeiten kann. Der Server ist erst sinnvollerweise als *shared* Server zu konfigurieren, wenn das Datenbanksystem mit einer hohen Anzahl von gleichzeitigen Transaktionen fertig werden muß, z.B. wenn viele Benutzer parallel damit arbeiten. Der Benutzerprozeß kommuniziert typischerweise mit einem Prozeß (Server- oder Dispatcherprozeß) auf der Seite des Servers (*two task*-Konfiguration), um an die Daten heranzukommen. Diese Konstellation kann man auch ändern, indem man den Code des Serverprozesses mit dem Code des Benutzerprozeß kombiniert, in einer sog. *single task*-Konfiguration. Der Benutzerprozeß übernimmt die Funktionalität des normalen Serverprozesses und hat dadurch den direkten Zugriff auf den SGA. Es bestehen keine Kosten für die Kommunikation zwischen Prozessen. Dafür sind aber Schutzmechanismen für gemeinsam benutzte Daten im SGA zu implementieren.

Die Speicherstruktur der Daten wirkt besonders auf die Performanz der Transaktionen. Das Datenbanksystem wird mehr Zeit brauchen, nach gewünschten Daten durchzusuchen, wenn die Daten verstreut im Hintergrundsspeicher liegen. Auch hier läßt ORACLE dem Anwender genügend Freiraum, seine Datenbank selbst zu gestalten. Zu einer ORACLE-Datenbank gehören eine physikalische und eine logische Speicherstruktur. Die physikalische Struktur wird vom Betriebssystem verwaltet. Es sind die Dateien, die eine ORACLE-Datenbank ausmachen. Jede ORACLE-Datenbank wird auf einer oder mehreren Datendateien (Data Files), zwei oder mehreren Log-Dateien (Redo Log Files) und einer oder mehreren Kontrolldateien (Control Files) aufgebaut. Die Lage dieser Dateien spiegelt die allgemeine Datenorganisation der Datenbank wider. Die Hierarchie der Einheiten, in denen die

Datenobjekte abgespeichert werden, bildet die logische Struktur der Datenbank. Die logische Struktur sagt darüber aus, wie der der Datenbank zur Verfügung stehende physikalische Speicherplatz für verschiedene Gruppen von Daten organisiert wird. Dazu gehören die Einheiten Tablespace, Segment, Extent und Data Block. Sie werden von der ORACLE-Instanz selbst verwaltet.

Abbildung 14 Logische und physikalische Struktur einer ORACLE-Datenbank




Die größten logischen Speichereinheiten einer ORACLE-Datenbank sind Tablespaces. Beim Erzeugen eines Tablespaces muß man immer den Dateinamen bzw. die Dateinamen angeben, unter denen die Daten auf der Festplatte abgelegt werden. Nur hier kommt die physikalische Struktur mit der logischen Struktur in Berührung, weil man daran sehen kann, welche Dateien im Hintergrundsspeicher zur Datenbank gehören. ORACLE benutzt Tablespaces, um größere Mengen voneinander unabhängiger Daten unterzugliedern. Durch die Organisation von Tablespaces auf verschiedenen Festplatten kann man I/O-Zugriffe verteilen und Zeit für Applikation gewinnen. ORACLE erzeugt automatisch das Tablespace SYSTEM, wo das Data Dictionary für die ganze Datenbank abgespeichert wird. Das Data Dictionary enthält verschiedene Tabellen, Views mit Informationen über die physikalische, logische Struktur der Datenbank, sowie dynamische Tabellen mit statistischen Informationen, die ORACLE während der Laufzeit gesammelt hat. Um immer exakt die Struktur und den Zustand der Datenbank zu reflektieren, wird das Data Dictionary automatisch von ORACLE aktualisiert.

Zur Speicherung von Daten gleicher Struktur werden Segmente innerhalb Tablespaces gebildet. Jedes Segment hat einen Typ, der die enthaltene Datengruppe indiziert. Tabellen bzw. Indizes werden in separaten Segmenten abgelegt. Außerdem gibt es Rollback-Segmente, Cache-Segmente, Cluster-Segmente. Segmente bestehen aus einem oder mehreren Extents, die als zusammenhängende Speicherblöcke innerhalb der Tablespaces allokiert werden. Die kleinsten logischen Speichereinheiten sind ORACLE-Datenblöcke (Data Blocks), aus denen jedes Extent besteht. Alle Lese- bzw. Schreiboperationen in ORACLE sind block-orientiert. Die Größe eines ORACLE-Blocks kann man beim Erzeugen der Datenbank variabel setzen (DB_BLOCK_SIZE). Um aber unnötige I/O-Operationen zu vermeiden, soll man die Größe eines ORACLE-Blocks als ein Vielfaches von Betriebssystemblöcken setzen.

Jede Operation zur Allokation von Speicherplatz auf der Festplatte erfolgt in ORACLE auf Extent-Basis, d.h. wenn ein Rollback-Segment voll ist, wird er um ein weiteres Extent erweitert, wenn ein Segment einer Tabelle oder eines Index voll ist, wird ein neues Extent hinzugefügt. Ein Extent bedeutet einen kontinuierlichen Speicherblock, ein Segment aber nicht. Für bessere Performanz ist wünschenswert, daß eine Tabelle bzw. ein Index nicht verstreut, also auf mehreren Extents, liegt, sondern auf ein einziges Extent hineinpaßt. Für das Anwachsen der Daten muß aber auch genug Freiraum im Extent vorhanden sein. Als Option stellt ORACLE eine STORAGE-Klausel für speicherallokierende SQL-Befehle wie CREATE TABLESPACE, CREATE TABLE, CREATE INDEX, CREATE ROLLBACK SEGMENT mit folgenden voreingestellten Werten zur Verfügung:

```
STORAGE (
   INITIAL      10K
   NEXT         10K
   MINEXTENTS   1
   MAXEXTENTS   121
   PCTINCREASE  50
)
```

Das erste und zweite Extent haben jeweils die Größe von 10 KB. Aber jedes danach allokierte Extent hat die Größe, die um PCTINCREASE Prozent größer als das vorherige ist (hier 50%). Maximal darf ein Segment aus 121 Extents bestehen. Wenn man in INITIAL einen genügend hohen Wert eingibt, ist dann sehr wahrscheinlich, daß die Tabelle auf ein einziges Extent hineinpaßt und interne Fragmentierung dadurch vermieden wird.

Die ORACLE-Blöcke enthalten die eigentlichen Datensätze. Auch hier spielt die Fragmentierung eine große Rolle. Wenn der Block voll ist und ein Datensatz beim Aktualisieren länger wird, muß der Datensatz gebrochen und auf verschiedene Blöcke verteilt werden (chained row). Es dauert beim Laden sowie Ändern solcher Datensätze natürlich länger. ORACLE stellt zwei platzverwaltende Parameter für die Datenblöcke zur Verfügung: PCTFREE (voreingestellt: 10) und PCTUSED (voreingestellt: 40). PCTFREE besagt, wieviel Prozent des Blocks für zukünftige UPDATEs der vorhandenen Datensätze in diesem Block frei bleiben muß, wenn ORACLE INSERT-Operationen in diesem Block ausführt. PCTUSED besagt, wieviel Prozent des Blocks als freier Platz erreicht werden, bevor ORACLE wieder bei INSERTs neue Datensätze in diesen Block hineinschreibt. Ein niedriger Wert in PCTFREE und ein hoher Wert in PCTUSED würden in einer sehr effizienten Abspeicherung der Daten resultieren. Dabei muß man aber mit der Gefahr von Datensatz-Chaining rechnen.

### 4.1.2 Optimierungsmöglichkeiten des Servers

(Literatur zu diesem Abschnitt: [Corrigan94], [OraSerTun96])

Da ORACLE dem Anwender große Freiheit bei der Gestaltung seines Datenbankssystems in Form von Parametern anbietet, ist es sehr wichtig, diese Größen entsprechend dem Datenbestand bzw. der Systemumgebung zu berechnen und dem Datenbankserver zu übergeben. Bei Vernachlässigung, eine passende Konfiguration für den Server zu erarbeiten, können z.B. die zur Verfügung stehenden Systemresourcen nicht optimal genutzt werden und es führt oft zu unerwartet schlechter Performanz der Anwendungen. Die Optimierungsüberlegungen in diesem Abschnitt beschäftigen sich vor allem mit einer angemessenen Parametrierung, um die beste Performanz der ORACLE-Instanz zu erzielen.

Der kritischste Bereich einer ORACLE-Instanz ist der SGA. Im SGA liegen die Puffer, die zur Speicherung temporären Daten bestimmt sind. Darunter sind der Database Cache-Puffer, der Shared Pool-Puffer und der Redo Log-Puffer von Interesse. Im Server Manager kann man mit dem Befehl *show sga* den Inhalt des SGAs anzeigen lassen:

```
SVRMGR> show sga
Total System Global Area      31200124 bytes
Fixed Size                       39732 bytes
Variable Size                 16988232 bytes
Database Buffers              13107200 bytes
Redo Buffers                   1064960 bytes
```

Der Database Cache-Puffer enthält die Datenblöcke, die vom Server in den Transaktionen geladen sind. Je größer der Puffer ist, desto kleiner ist die Wahrscheinlichkeit, daß der Server beim Suchen nach gewünschten Daten einen neuen Datenblock von der Festplatte hineinlesen muß. Das Ziel dabei ist zeitaufwendige I/O-Zugriffe zu vermeiden. Während der Laufzeit zählt ORACLE alle Zugriffe mit und speichert die Anzahl in der dynamischen Tabelle V$SYSSTAT. Man kann mit Hilfe dieser Tabelle die Trefferquote berechnen, mit der die Zugriffe auf den Database Cache-Puffer erfolgt haben:

```
SQL> SELECT SUBSTR(name, 1, 20), value
     FROM v$sysstat
     WHERE name IN ('db block gets', 'consistent gets', 'physical reads');
SUBSTR(NAME,1,20)         VALUE
-------------------- ----------
db block gets            295262
consistent gets        17112213
physical reads           775364
```

Die Werte von *db block gets* und *consistent gets* stellt die Anzahl aller Anforderungen auf Datenblökke dar, während *physical reads* zählt die Anfragen, in denen die Daten neu von Dateien hochgeladen werden müssen. Die Trefferquote im Database Cache-Puffer kann wie folgt berechnet werden:

$$\textit{Trefferquote} = 1 - (\textit{physical reads} / (\textit{db block gets} + \textit{consistent gets})) \quad (1)$$

Nach der Formel bekommt man:

$$\textit{Trefferquote} = 1 - (775364 / (295262 + 17112213)) = 1 - 0.04 = 0.96 = 96\% \quad (2)$$

Dies ist ein ganz guter Wert und auch der Größe des Puffers angemessen: 13107200 B = 13 MB. Wenn diese Quote sehr niedrig ausfällt (60%-70%), sollte man schon die Größe des Puffers erhöhen. Es geht wieder auch nur, wenn noch kein Paging- und Swapping-Verhalten im System verursacht wird. Mit dem Parameter DB_BLOCK_BUFFERS kann man die Größe des Puffers in der Startdatei INIT.ORA setzen. Für einen sehr großen Database Cache-Puffer ist der DBWR-Prozeß manchmal überfordert, Datenblöcke vom Puffer auf die Festplatte zu kopieren, besonders wenn die Festplatten bzw. die I/O-Kontrollhardware sehr langsam sind. Die Lösung dafür ist, die Datenbank auf verschiedene Festplatten im System zu verteilen und die Anzahl der DBWR-Prozesse (DB_WRITERS) entsprechend zu erhöhen.

Der Redo Log-Puffer ist ein Ringpuffer, der die Redo-Informationen enthält. Dadurch werden die Lesekonsistenz in der Datenbank bzw. die Restaurierung der Datenbank im Fehlerfall gewährleistet. Da es ein Ringpuffer ist, muß er regelmäßig auf die Festplatte kopiert werden, um nicht überschrieben zu werden. Der Hintergrundsprozeß LGWR schreibt den Inhalt des Redo Log-Puffers in die Online Redo Log-Dateien, wenn der Puffer zu einem Drittel seiner Größe gefüllt ist oder eine Transaktion durch

COMMIT beendet wird. Mit Hilfe der folgenden Anfrage kann man die Anzahl der Forderungen nach Speicherplatz im Redo Log-Puffer erfahren:

```
SQL> SELECT SUBSTR(name, 1, 30), value
     FROM v$sysstat
     WHERE name = 'redo log space requests'
SUBSTR(name,1,30)                    VALUE
------------------------------ ----------
redo log space requests                121
```

Wenn dieser Wert sich während der Laufzeit der Transaktionen ständig erhöht, ist sicherlich der Redo Log-Puffer zu klein konfiguriert. Die Folge ist, daß Prozesse auf freien Speicherplatz im Puffer warten müssen. Eine Vergrößerung des Puffers (LOG_BUFFER) bedeutet auch, daß der LGWR-Prozeß seltener aktiviert wird, dadurch kann man Zeit für andere Aktionen gewinnen. Dabei muß man aber beachten, daß ein zu großer Redo Log-Puffer die Dauer der Schreiboperationen des LGWR-Prozesses auch negativ beeinflussen kann. Der CKPT-Prozeß macht die Zeitstempel auf die Daten und Kontrolldateien nach jedem Checkpoint. Wenn er nicht aktiv ist (CHECKPOINT_PROCESS = FALSE), muß der LGWR-Prozeß diese Arbeit übernehmen. Für ein System mit langsamen Festplatten soll man den CKPT-Prozeß immer aktivieren, um den LGWR-Prozeß von seiner Arbeit zu entlasten.

Die Allokation von Speicherplatz im Redo Log-Puffer erfolgt durch das sog. Redo Allocation Latch. Der Benutzerprozeß muß das Redo Latch bekommen, um Platz im Puffer für sich zu allokieren. Da es im System nur ein Redo Allocation Latch gibt, kann immer nur ein Prozeß zu einer bestimmten Zeit Platz im Redo Log-Puffer allokieren. Wenn der Redo-Eintrag sehr groß ist (z.B. bei LONG RAW-Spalten), muß der Prozeß das Redo Copy Latch anfordern, bevor der Eintrag in den Puffer kopiert werden kann. ORACLE sammelt Statistik über Latch-Aktivitäten während seiner Laufzeit und speichert in der dynamischen Tabelle V$LATCH:

```
SQL> SELECT SUBSTR(ln.name, 1, 20), gets, misses,
          immediate_gets, immediate_misses
     FROM v$latch l, v$latchname ln
     WHERE ln.name IN ('redo allocation', 'redo copy')
     AND ln.latch# = l.latch#
SUBSTR(LN.NAME,1,20)       GETS     MISSES IMMEDIATE_GETS IMMEDIATE_MISSES
-------------------- ---------- ---------- -------------- ----------------
redo allocation        11425194        259              0                0
redo copy                    0          0              0                0
```

Wenn das Verhältnis von *misses* zu *gets* bzw. von *immediate_misses* zu *immediate_gets* mehr als 1% beträgt, besteht ein Engpaß mit den Latches im System, d.h. Prozesse halten das Latch zu lange und hindern dabei andere Prozesse, daran zu kommen. Um das Problem zu lösen, muß man den Startparameter LOG_SMALL_ENTRY_MAX_SIZE verkleinern. Der Parameter schreibt vor, wie groß ein Redo-Eintrag ist, der in den Redo Log-Puffer über das Redo Allocation Latch geschrieben wird. Ist der Eintrag klein, kann ein Prozeß das Latch nicht lange besetzen.

Der Shared Pool-Puffer ist der Puffer, wo der Server die Daten über die Datenbank (Data Dictionary) und die kompilierten PL/SQL-Blöcke bzw. SQL-Anfragen temporär ablegt. Der Speicherbereich stellt auch einen Cache-Speicher dar. Der Aufwand, um Daten vom Data Dictionary (Tabelleninformationen, Trigger, Stored Procedures) wieder in den Hauptspeicher zu bringen bzw. neue SQL-Statements zu parsen und zu kompilieren, kostet mehr Zeit als eine fehlgeschlagene Leseaktion auf den Database Cache-Puffer, deshalb ist die entsprechende Größe des Shared Pool-Puffers ausschlagge-

bend für hohe Performanz der ORACLE-Instanz. Zwei Hauptbestandteile des Shared Pool-Puffers sind einmal der Data Dictionary Cache, wo die Informationen über Elemente der Datenbank wie Tabellen, Indizes, Dateien, Constraints zeitweise abgelegt werden, und zum andern der Library Cache, wo sich die geparsten und kompilierten SQL-Statements, PL/SQL-Blöcke befinden. Sie stehen also für sofortige Ausführung zur Verfügung. Die Trefferquote im Data Dictionary Cache kann man mit Hilfe der Informationen aus der dynamischen Tabelle V$ROWCACHE berechnen:

```
SQL> SELECT SUM(gets-getmisses-usage-fixed) / SUM(gets) “Row Cache”
     FROM v$rowcache;
Row Cache
----------
,957408333
```

Die Trefferquote im Library Cache kann man wie folgt von der dynamischen Tabelle V$LIBRARYCACHE herauslesen:

```
SQL> SELECT SUM(pins-reloads) / SUM(pins) “Lib Cache”
     FROM v$librarycache;
Lib Cache
----------
,999574904
```

Beide Werte repräsentieren eine sehr gute Situation im Shared Pool-Puffer. Anhand dieser Werte kann man die Größe des Shared Pool-Puffers entsprechend variieren (SHARED_POOL_SIZE). Hier gilt wieder, daß der Puffer nicht allzu viel Speicher vom Betriebssystem wegnehmen darf, ansonsten nimmt man das Swapping- und Paging-Risiko in Kauf.

Für häufige und auf große Tabellen angewandte Sortierqueries muß der Sortierbereich auch entsprechend groß angelegt werden (SORT_AREA_SIZE). Von der dynamischen Tabelle V$SYSSTAT kann man Informationen über die Anzahl der Sortierungen erhalten, die ganz im Sortierspeicher bzw. nicht im Sortierspeicher durchgeführt werden können:

```
SQL> SELECT SUBSTR(name, 1, 20), value FROM v$sysstat
     WHERE name IN (‘sorts (memory)’, ‘sorts (disk)’);
SUBSTR(NAME,1,20)         VALUE
-------------------- ----------
sorts (memory)            16348
sorts (disk)                  1
```

In diesem Fall ist kein Grund zur Sorge, da die Anzahl der Sortierungen, die nicht im Speicher durchgeführt werden, ziemlich gering ist. Andernfalls muß man schon einen größeren Wert für den Sortierbereich angeben, um Festplattenzugriffe zu vermeiden.

Die genannten Parameter aus der Startdatei INIT.ORA sind die wichtigsten Parameter, die man sich beim Installieren einer neuen Datenbank anschauen und setzen muß, um sie der Systemumgebung anzupassen und die beste Performanz insgesamt herauszuholen. Diese Parameter wirken direkt auf das Betriebssystem, da ORACLE dem Betriebssystem dadurch Speicher wegnimmt. Zu vermeiden ist, daß zu viel Speicher für ORACLE allokiert wird und das System seine Zeit für Swapping und Paging-Aktivitäten verwendet. Nach dem Ändern dieser Parameter ist während der Laufzeit von ORACLE das Verhalten des Betriebssystem mit Hilfe von statistischen Werkzeugen zu beobachten (siehe 5.2 ***Softwareumgebung und Werkzeuge,*** Seite 57). Im Anhang C werden außerdem alle Startparameter von ORACLE mit einer kurzen Beschreibung aufgelistet, die bestimmte Wirkungen auf die Per-

formanz der ORACLE-Instanz haben. Sie sind nützlich für eine detailliertere Konfiguration des Datenbankservers.

Ein weiterer Aspekt bei der Performanzoptimierung einer ORACLE-Datenbank ist die Aufgabe, die Daten im Hintergrundsspeicher gut zu organisieren. Auf unterschiedliche Datengruppen wird unterschiedlich oft zugegriffen. Je mehr datenmanipulierende Anfragen im Datenbanksystem abgearbeitet werden, desto mehr Änderungsinformationen werden gespeichert. Dies betrifft sowohl die Online Redo Log-Dateien als auch die Rollback-Segmente der Datenbank. Die Rollback-Segmente sind in der Regel auch der schreibintensivste Bereich in der Datenbank. Die Schreiboperationen erfolgen sequentiell auf der Festplatte und verlangen einen größeren Zeitaufwand, wenn die Festplatte sich noch mit anderen Transaktionen beschäftigen muß. In einer guten Konfiguration müssen die Redo Log-Dateien und die Datendateien auf verschiedenen Festplatten liegen. Dies bringt nicht nur mehr Performanz, sondern auch mehr Sicherheit mit sich, weil die Wahrscheinlichkeit bei einem Plattenfehler, die Redo Log-Dateien und zugleich die Datendateien zu verlieren, geringer wird. Obwohl der Ladeprozeß in der Regel sequentiell abläuft, d.h. zuerst wird der Index evaluiert und dann werden die Daten geladen, kann man auch zu besseren Antwortzeiten kommen, wenn die Tablespaces mit den Indizes und den Daten separat voneinander auf verschiedenen Festplatten liegen. Dadurch vermeidet man unnötige Bewegungen von Leseköpfen der Festplatte. Es ist aber zu vermuten, daß eine Verbesserung dabei erst sichtbar wird, wenn man mit großen Mengen von zu ladenden Daten zu tun hat. Um eine Statistik darüber zu bekommen, wie die Lese- und Schreibzugriffe auf einzelne Dateien der Datenbank verteilt sind, kann man in der Datenbankumgebung wie folgt abfragen:

```
SQL> SELECT SUBSTR(name, 1, 40), phyrds, phywrts
     FROM v$datafile df, v$filestat fs
     WHERE df.file# = fs.file#
     ORDER by phywrts;
SUBSTR(NAME,1,40)                          PHYRDS    PHYWRTS
---------------------------------------- ---------- ----------
/u01/oradata/emp1/data/PRIME_FX.dbf          148243       1612
/u02/oradata/emp1/index/PRIME_BA_IX.dbf         745       1989
/u01/oradata/emp1/system/system01.dbf         28242       3110
/u04/oradata/emp1/roll/IMDBA_emp1_rb_02.        186       6569
/u04/oradata/emp1/roll/IMDBA_emp1_rb_01.        929       6698
```

Man muß hier zwischen Zeitansprüchen der Lese- und Schreiboperationen abwägen, um daraus eine angemessene Verteilung der Tablespaces auf unterschiedlich schnelle Platten zu finden. Erfahrungsgemäß soll man die Redo Log-Dateien immer auf der schnellsten Platte plazieren, die Rollback-Segmente bzw. die Datendateien, auf die am meisten zugegriffen wird, auf einer anderen.

Um Fragmentierung in Tablespaces und den dadurch verursachten unnötigen I/O-Aufwand zu vermeiden, müssen die Tabellen und die Indizes idealerweise immer auf einem einzigen Extent abgelegt werden. Mit folgender Anfrage kann man den aktuellen Zustand der Speicherplatzbelegung in der Datenbank erfahren:

```
SQL> SELECT SUBSTR(tablespace_name, 1, 15), SUBSTR(segment_name, 1, 15),
     SUBSTR(segment_type, 1, 10), COUNT(*), SUM(bytes)
     FROM user_extents
     GROUP BY tablespace_name, segment_name, segment_type
     ORDER BY COUNT(*);
SUBSTR(TABLESPA SUBSTR(SEGMENT_ SUBSTR(SEG   COUNT(*) SUM(BYTES)
```

```
-------------- -------------- ---------- ---------- ----------
PRIME_FX_IX    UNQ_MTN_FXTANAM INDEX              15    1536000
PRIME_FX       INPUTVAR        TABLE              37    3788800
PRIME_FX       FXWSEGD         TABLE              42    4300800
PRIME_FX       DISPDES         TABLE              48    4915200
PRIME_FX       FXWSEGS         TABLE              54   11059200
```

Die aufgelisteten Tabellen bzw. Indizes bestehen aus zu vielen Extents, was eine hohe Fragmentierung des Tablespaces und zugleich eine schlechte Situation für Zugriffe bedeutet. Die Lösung für solche Situation ist für Tabellen:

- Tabelle umbenennen (RENAME *tab* TO *temp*).
- Tabelle erneut mit einem neuen INITIAL-Wert erzeugen, der mindestens so groß wie die Summe der Größe aller Extents der Tabelle ist (CREATE TABLE *tab* (...) STORAGE (INITIAL *new* ...)).
- Daten von der umbenannten Tabelle übernehmen (INSERT INTO *tab* SELECT * FROM *temp*).
- temporäre Tabelle löschen (DROP TABLE *temp*).

für Indizes:

- Index löschen (DROP INDEX *ind*).
- Index erneut mit einem angemessenen INITIAL Wert erzeugen (CREATE INDEX *ind* ON *tab* (*col*, ...) STORAGE (INITIAL *new* ...)).

Mit dem Befehl ANALYZE veranlaßt man den Server, ein bestimmtes Element der Datenbank (Tabelle, Index) zu analysieren. Die Ergebnisse findet man dann in der Tabelle USER_TABLES oder DBA_TABLES wieder. Diese Informationen sind sehr wichtig z.B. dafür, die Parameter PCTFREE, PCTUSED angemessen für einzelne Tabellen zu setzen.

```
SQL> ANALYZE TABLE varlo COMPUTE STATISTICS
Table analyzed.
SQL> SELECT SUBSTR(table_name, 1, 10), num_rows, blocks, empty_blocks,
       avg_space, chain_cnt , avg_row_len
     FROM user_tables
     WHERE table_name = 'VARLO';
SUBST   NUM_ROWS     BLOCKS EMPTY_BLOCKS  AVG_SPACE  CHAIN_CNT AVG_ROW_LEN
----- ---------- ---------- ------------ ---------- ---------- -----------
VARLO       4426        430           69        372          0         150
```

In der Tabelle gibt es insgesamt 4 426 Datensätze. Die Tabelle belegt 430 Datenblöcke und im aktuellen Extent sind 69 Datenblöcke frei. In den Datenblöcke sind durchschnittlich 372 Bytes frei. Es gibt keinen Datensatz, der über mehreren Datenblöcken verstreut liegt (chained rows), und die Datensätze haben eine durchschnittliche Länge von 150 Bytes. Wenn der durchschnittliche freie Platz in den Datenblöcken zu groß ist, daß mehrere (mehr als 4) Datensätze (von durchschnittlicher Länge) noch hineinpassen können, dann kann man den PCTFREE-Wert noch erhöhen. Wenn aber die Aufsplitterung der Datensätze (chaining) generell in der Tabelle dadurch versursacht wird, dann soll man schon einen kleineren Wert dafür wählen.

### 4.1.3 Optimierungsmöglichkeiten des Clients

(Literatur zu diesem Abschnitt: [Corrigan94], [OraSerTun96], [Shasha92])

Mit einzelnen SQL-Statements kommuniziert man mit dem Datenbankserver. Dies passiert, wenn man manuell die Anfrage am Kommandoprompt von SQL*Plus selbst eingibt oder, wenn eine Applikation ihre eingebetteten SQL-Statements bzw. PL/SQL-Blöcke an den Server schickt. Die Stellen des Clients, an denen optimiert werden kann, sind deshalb die SQL-Statements. Es gibt verschiedene Arten, wie man ein SQL-Statement formulieren kann, weshalb sich eine Optimierung in diesem Sinne immer auf das aktuelle Statement bezieht. Dazu sind folgende Performanzaspekte zu beachten.

Um kleinere Transaktionen mit ihrem Vorteil, daß die Systemresourcen nicht so sehr herangezogen werden, zu unterstützen, stellt ORACLE noch einen sog. diskreten Transaktionstyp zur Verfügung. Das Konzept der diskreten Transaktionen basiert darauf, daß die Transaktion nur wenige Datenblöcke und nie den gleichen Datenblock mehr als einmal modifiziert, daß die neu modifizierten Spaltenwerte für die Transaktion nicht von Interesse sind, und daß keine von den zu modifizierenden Spalten vom Typ LONG sind. Die Nutzung von diskreten Transaktionen kann eine bessere Performanz bringen, da die Undo-Informationen gesondert abgespeichert werden. Der Einsatzbereich für diskrete Transaktionen ist aber wegen ihres Funktionalitätsumfangs stark beschränkt. Der Prozeduraufruf zu einer diskreten Transaktion steht im Package DBMS_TRANSACTIONS bereit und kann innerhalb eines PL/SQL-Blocks wie folgt realisiert werden:

```
BEGIN
   dbms_transaction.begin_discrete_transaction;
   ...
EXCEPTION
   WHEN dbms_transaction.discrete_transaction_failed THEN
   ROLLBACK;
END;
```

Unter den internen Datentypen, die ORACLE für die Definition der Tabellenspalten benutzt, gibt es einen Sondertyp: ROWID. Ein ROWID-Eintrag hat das Format `BBBBBBBB.RRRR.FFFF` und enthält die interne Adresse eines Datensatzes mit Blocknummer (`BBBB`-Feld), Datensatznummer (`RRRR`-Feld) und Dateinummer (`FFFF`-Feld). Der Zugriff über ROWID ist der einzig schnellste Weg, um an die Daten zu kommen. Da es sich um eine Besonderheit von ORACLE handelt, soll man trotz der Performanzvorteile den Gebrauch von ROWID vermeiden, wenn die Priorität für die Portierbarkeit der Datenbank höher liegt.

In einer Transaktion soll man die Anzahl der Anfragen so gering wie möglich halten, ggf. um den Preis einer höheren Komplexität der einzelnen Anfragen. Das hängt mit dem Verfahren von ORACLE zusammen, SQL-Statements abzuarbeiten. Jedes Mal, wenn ein SQL-Statement ausgeführt wird, muß ORACLE mehrere interne Bearbeitungsschritte durchgehen:

- Cursor öffnen
- Statement parsen, auf syntaktische Fehler suchen
- Variablen binden
- Felder mit Typen beschreiben
- Statement ausführen, in der Datenbank suchen
- Datensätze holen (Fetch)

– Cursor schließen

([OraSerTun96], Appendix B-1 SQL Processing Concept)

Für den gleichen Bestand an zu ladenden Daten gilt das einfache Prinzip: mit je weniger Anfragen man auskommt, desto schneller läuft der gesamte Vorgang. Die Kommunikation zum Server wird besonders zeitkritisch, wenn der Client Anfragen von einer entfernten Maschine über das Netzwerk an den Server schickt. Im folgenden wird illustriert, wie man am besten Daten von der Datenbank herausholen kann:

Erste Möglichkeit mit zwei SQL-Statements:

```
SELECT name, version FROM worldmap WHERE wmapid = 299;
SELECT name, version FROM worldmap WHERE wmapid = 381;
```

Zweite Möglichkeit mit Hilfe eines PL/SQL Blocks:

```
DECLARE
  CURSOR wmapcursor(wmap NUMBER) IS
    SELECT name, version FROM worldmap
    WHERE wmapid = wmap;
BEGIN
  OPEN wmapcursor(299);
  FETCH wmapcursor INTO ..., ...;
  CLOSE wmapcursor;
  OPEN wmapcursor(381);
  FETCH wmapcursor INTO ..., ...;
  CLOSE wmapcursor;
END;
```

Dritte Möglichkeit mit Hilfe eines Table-Joins:

```
SELECT A.name, A.version, B.name, B.version
FROM worldmap A, worldmap B
WHERE A.wmapid = 299 AND B.wmapid = 381;
```

Mit der zweiten Möglichkeit kann man schon eine Verbesserung der Performanz sehen, da zwei Statements auf denselben Cursor zurückgeht, der nur einmal geparst und kompiliert wird. Es sind jedoch noch zwei Fetch-Operationen nötig. In der dritten Möglichkeit wird das Statement einmal geparst, kompiliert und auch nur eine Fetch-Operation durchgeführt. Das Prinzip gilt natürlich nicht nur für Ladeoperationen. Man soll in einer zeitkritischen Applikation immer versuchen, unabhängige Anfragen so weit wie möglich zusammenzufassen, um den Server nicht zu belasten bzw. den Kommunikationsaufwand zu minimieren. Die Verwendung von Host Arrays ist daher immer zu bevorzugen.

Für Spalten einer großen Tabelle, die oft in Table-Joins als Join-Spalten vorkommen, soll man Indizes anlegen (CREATE INDEX). Für die Primärschlüsselspalten und diejenigen Spalten, auf die ein UNIQUE-Constraint definiert ist, werden Indizes automatisch von ORACLE erzeugt. Dabei muß man beachten, eine angemessene Größe von Speicherplatz für die Indizes anzugeben, um Fragmentierung zu vermeiden. In ORACLE ab Version 7.3.3 steht eine neue Implementierung für Indizes zur Verfügung, nämlich die Bitmap-Indizes. Die gängige Implementierung auf B-Baum-Basis eignet sich besonders gut für hohe Anzahl von eindeutigen Spaltenwerten, also für Spalten mit hoher Kardinalität. Wenn aber eine Spalte mit sehr niedriger Kardinalität zu indizieren ist, d.h. die Spalte enthält trotz großer Anzahl der Datensätze nur wenige unterschiedliche Werte, kann ein auf die Spalte erzeugter

Bitmap-Index im Vergleich zu einem B-Baum-Index große Performanzvorteile bringen. Indizes können im allgemeinen die Datensuche beschleunigen. Sie verlangsamen aber die UPDATE- bzw. INSERT-Operationen, weil die Indizes dabei auch aktualisiert werden. Wenn man solche Operationen auf eine große Menge von Daten anwendet, ist es besser, bewußt das SQL-Statement so zu formulieren, daß die Operationen die betroffenen Indizes nicht mit aktualisieren. Erst nachdem diese Operationen erfolgreich beendet sind, sind die Indizes wieder aufzubauen (ALTER INDEX *ind* REBUILD). Um Indizes bewußt anzusprechen bzw. zu vermeiden, soll man die Formulierung der Bedingungen in den WHERE-Klauseln der SQL-Statements beachten, in denen die indizierten Spalten vorkommen. Jede Berechnung mit Index-Spalten wird den Index für diese Spalten ausschalten. Dazu zählen die normalen arithmetischen Funktionen (+,-,*, /), die Zeichenkettenkonkatenation (||) oder die logische Negation (NOT). Für eine Datensuche über Indizes muß man entsprechend solche Berechnungen bzw. Manipulationen der Index-Spalten vermeiden.

Für einzelne SQL-Statements hat die Art und Weise, wie man die Bedingungen in den WHERE-Klauseln formuliert, großen Einfluß auf die Performanz. In ORACLE Version 6 steht ein regelorientierter SQL-Optimierer zur Verfügung, der die eingegebenen SQL-Befehle vor deren Ausführung analysiert und sie entsprechend ändert, um einen schnelleren Zugriffsweg zu bekommen. Die Arbeitsweise des Optimierers ist, die Suchbedingungen in den Befehlen nach den vorgegebenen Performanzprioritäten zu untersuchen und ggf. umzuordnen. Man soll sich dieser Performanzprioritäten bewußt sein, um eigene SQL-Statements gezielt für bessere Performanz zu formulieren:

Tabelle 4 Performanzrangliste[1]

| Rank | Conditions |
|---|---|
| *1* | *ROWID = constant* |
| *2* | *Entire UNIQUE concatenated index = constant* |
| *3* | *UNIQUE indexed column = constant* |
| *4* | *Entire cluster key = corresponding cluster key of another table in the same cluster* |
| *5* | *Entire cluster key = constant* |
| *6* | *Entire non-UNIQUE concatenated index = constant* |
| *7* | *Non-UNIQUE single-column index merge* |
| *8a* | *More than one column of leading concatenated index = constant* |
| *8b* | *First column of leading concatenated index = constant* |
| *9* | *Indexed column BETWEEN low and high value or indexed column LIKE “C%” (bounded range)* |
| *10* | *sort / merge within table joins* |
| *11* | *MAX / MIN of single indexed columns* |
| *12* | *ORDER BY entire index* |
| *13* | *Full-table scan* |
| *14* | *Unindexed column = constant or column IS NULL / LIKE “%C%”* |

In ORACLE Version 7 kann der SQL-Optimierer sowohl regelorientiert als auch kostenorientiert arbeiten. Das Verhalten des Optimierers kann mit Hilfe der Startparameter OPTIMIZER_GOAL (ALL_ROW | FIRST_ROW) und OPTIMIZER_MODE (CHOOSE | RULE) festgelegt werden. Der

[1] Für eine genauere Analyse von einzelnen Bedingungstypen siehe [Corrigan94], Seite 101ff.

regelorientierte Optimierer existiert jedoch nur, um Applikationen zu unterstützen, die in ORACLE Version 6 entwickelt wurden. Alle neuen Optimierungsfunktionalitäten gehen auf kostenorientierte Betrachtungen zurück, denen die statistischen Informationen über Tabellen, Clusters oder Indizes zugrundeliegen. Der kostenorientierte Optimierer ist sozusagen intelligenter as der regelorientierte, da der kostenorientiere Optimierer die Bedingungen nach den Kosten der Zugriffe auf den aktuellen Datenbestand umordnet. Um die kostenorientierte Optimierungsfunktionalitäten voll auszunutzen, soll man vorher die Statistiken darüber mittels des ANALYZE-Befehls erstellen, die danach in verschiedenen ORACLE-eigenen Tabellen für den Optimierer bereitliegen. Der Nachteil ist nur, daß der Vorgang, alle Tabellen eines Datenschemas zu analysieren, ziemlich lange dauern kann. Wenn der Datenbestand sich oft ändert, muß die Analyse auch entsprechend oft wiederholt werden.

Wie schon erwähnt wird, werden alle SQL-Statements vom ORACLE-Server geparst und in Form von Ausführungsplänen im Library Cache (im Shared Pool-Puffer) für späteren Aufruf temporär behalten. ORACLE führt verschiedene Statistiken über einzelne Statements in der dynamischen Tabelle V$SQLAREA:

```
SQL> SELECT SUBSTR(sql_text,1,30), executions,
       buffer_gets, disk_reads, parse_calls
  FROM v$sqlarea
  ORDER BY executions
SUBSTR(SQL_TEXT,1,30)          EXECUTIONS BUFFER_GETS DISK_READS PARSE_CALLS
------------------------------ ---------- ----------- ---------- -----------
SELECT COUNT(*)   FROM GLOFIGD      79674      159348        101        2030
INSERT INTO object values (:s1     147574    75840571   47467609        2267
SELECT COUNT(*)   FROM VARVA       149590      448937          0      105186
INSERT INTO varva VALUES (:s1:     149590      755749       5159      105085
SELECT VARID   FROM OBJECT  WH     154710      443159        346          74
DELETE FROM varva WHERE valid      154710     1582999       8030          74
```

Schlechte Situationen liegen vor, wenn die Verhältnisse der übrigen Werten zu *Executions* sehr hoch sind. Die betroffenen SQL-Statements muß man noch mal unter die Lupe nehmen und ggf. umschreiben, um Performanzkosten geringzuhalten. Man kann die maximale Anzahl der geöffneten Cursor in einer Session erhöhen, um unnötige *Parse_Calls* zu reduzieren. Der Nachteil ist aber, daß viel Speicher vom SGA dadurch belegt wird. In *Disk_Reads* werden die Blöcke gezählt, die von einem Cursor in allen Aufrufen von der Festplatte in den Speicher gelesen sind. Parallel dazu werden in *Buffer_Gets* die Puffer gezählt, die die Cursor während ihrer Ausführungen angefordert und bekommen haben.

## 4.2 Performanzanforderung

Das Laden ist wohl die häufigste Operation, die auf die Worldmaps angewandt wird. Es hängt damit zusammen, daß die Datenbank zur Konstruktion bzw. zum Test der konstruierten Worldmaps angelegt wird. Zu Beginn jeder Arbeitssession, in der der Benutzer eine vorhandene Worldmap editieren oder testen will, wird die entsprechende Worldmap geladen. Wenn die Arbeitssession endet, wird die Worldmap bzw. die Differenz zu Originaldaten wieder zurück in die Datenbank geschrieben. Anders als im Online-Betrieb, wo die entsprechende Worldmap nur einmal beim Systemhochlauf geladen wird, und dann permanent im Speicher für Überwachung und Kontrolle bleibt, passiert das Laden einer Worldmap im Test- bzw. Editierbetrieb viel häufiger. Die Operationen Ändern, Löschen und Einfügen betreffen in der Regel nur einen Teil der Worldmapdaten. Oft werden nur einzelne Objekte in

Segmenten geändert, gelöscht oder neu eingefügt, weshalb die Aufgaben, eine ganze Worldmap zu löschen oder komplett neu einzufügen (der schlimmste Fall für Ändern), nicht eine so hohe Priorität wie das Laden einer Worldmap haben.

Die Worldmap wird von der Datenbank in den Speicher geladen, um dann am Bildschirm gezeichnet zu werden. Deshalb kommen die meisten Datenbankanfragen beim Laden einer Worldmap von einer Applikation auf der Betriebssystemebene. Die Routinen, die die Graphikprimitive auf dem Bildschirm zeichnen, beanspruchen nicht so viel Zeit im Vergleich zum Ladeprozeß. Beim Zeichnen sind die darzustellende Daten bereits im Hauptspeicher. Außerdem muß die Applikation nicht mit anderen Applikationen kommunizieren und eventuell auf Daten warten. Die Darstellungszeit einer Worldmap leidet daher nicht unter großen Zeitschwankungen. Der Ladeprozeß hängt dagegen viel davon ab, wie belastet das Betriebssystem und besonders die Datenbank sind, da der Aufwand, mit der Datenbank zu kommunizieren und auf die gewünschten Daten zu warten, auch Zeit bedeutet.

Entscheidend für den Ladeprozeß ist, wie die Worldmapdaten abgelegt und wie sie geladen werden. Eine Ablagerung mit weniger Referenzebenen, mit weniger Relationen bedeutet weniger Zugriffe, um die benötigten Daten zu laden. Deshalb vereinigt man verwandte Relationen, wenn es mit der Performanz kritisch wird, wie der Entwurf gezeigt hat. Die Schemata verschiedener Optimierungen haben folgende Gemeinsamkeiten. Erstens werden die Objekte der Worldmaps in einer eigenen Relation aufgefaßt. Zweitens werden die Variablenbeschreibungen nach Typen auch in separaten Relationen abgelegt. Die Mindestkosten bestehen offensichtlich darin, alle Objekte einer Worldmap und alle Variablenbeschreibungen der dynamischen Objekte zu laden. Die Anzahl der Datenbankzugriffe ist in diesem Fall die Summe der Anzahl der Objekte und der Anzahl der Variablen. Die Worldmap können theoretisch bis zu hunderttausend Objekte enthalten. Deshalb muß der Ladeprozeß innerhalb von wenigen Minuten schlimmstenfalls bis zu hunderttausend Datenbankzugriffe schaffen. Eine hohe Performanz hängt dann nicht nur vom Datenbanksystem, sondern auch vom Ladealgorithmus.

Tabelle 5 Mindestladezeit einiger Beispielsworldmaps[1]

| Worldmap | Objects | Variables | Reads | 1 ms / Read | 2 ms / Read | 4 ms / Read |
|---|---|---|---|---|---|---|
| *Small* | *1 944* | *665* | *2 609* | *2,609* | *5,218* | *10,436* |
| *Medium* | *8 037* | *6 501* | *14 538* | *14,538* | *29,076* | *58,152* |
| *Big* | *50 000* | *40 000* | *90 000* | *90,000* | *180,000* | *360,000* |
| *Huge* | *100 000* | *80 000* | *180 000* | *180,000* | *360,000* | *720,000* |

Im schlimmsten Fall kommt man schon an die Grenze der Unakzeptierbarkeit, wie die Tabelle gezeigt hat. Pro Datensatz braucht das System 4 Millisekunde (0.004 Sekunde). Es kostet dann für eine überdimensionierte Worldmap mit 50 000 Objekten und 40 000 Variablen mindestens 360 Sekunden (6 Minuten), bzw. für eine monströse Worldmap mit 100 000 Objekten und 80 000 Variablen mindestens 720 Sekunden (12 Minuten).

Nach drei Optimierungen haben wir drei Schemata bekommen, die auch bereit zur Implementierung sind. Das erste ist am wünschenswertesten, denn es ist am detailliertesten. Jedes Element einer Worldmap wird extra gespeichert und sichtbar von jeder Zugriffsebene, innerhalb der Datenbankumgebung wie von SQL*Plus aus oder außerhalb der Datenbankumgebung wie von PROC/C++ Appli-

[1] Alle Zeiten, die in den Tabellen zu finden sind, werden stets mit Sekunden angegeben.

kationen aus. Mit jeder weiteren Optimierung verliert das Schema an Granularität und damit an Detailliertheit. Wie die Annahmen von Mindestladezeiten für verschiedene Worldmapgrößen schon darauf hindeutet, wird das am besten für Performanz optimierte Schema, nämlich das Schema der 3. Optimierung bei den übergroßen Worldmaps versagen. Der Ausweg ist nicht der, weiter zu optimieren und Objekte zu gruppieren, Entitäten zu vereinigen, denn man würde sich ansonsten wieder in der Anfangssituation befinden.

Da muß man sich mit zwei Fragen auseinandersetzen. Die erste ist, ob die Datenbank generell auf Worldmaps aller Größen vorbereitet sein muß, und die zweite ist, wie oft solche gigantische Worldmaps in der Realität vorkommen. Wenn die Datenbank generell unabhängig von der Größe der Worldmaps hohe Performanz liefern muß, dann ist der Ausweg derjenige, Worldmaps entsprechend nach bestimmten Kriterien in kleinere Worldmaps zu teilen und das schnellste Schema zu wählen (das dritte). Wenn die sehr großen Worldmaps in der Realität aber recht selten vorkommt, dann kann man das Schema wählen, das bei den normalen Worldmaps die erwartete Performanz und die gewünschte hohe Granularität liefert.

## 4.3 Grundlage des Performanztests

### 4.3.1 Datenstruktur

Zum Testen der verschiedenen Datenschemata werden entsprechende Funktionen zum Laden, Löschen und Einfügen von Worldmaps geschrieben, auf die Zeitmessungen angewandt werden, um Aussagen über die Performanz einzelner Implementierungen machen zu können. Die Algorithmen für diese Funktionen und insbesondere der Algorithmus für die Ladefunktion hängen nicht nur von dem Schema ab, in dem die Daten in der Datenbank gelagert sind, sondern auch von der Struktur, in der die geladenen Daten im Hauptspeicher abgelegt werden.

Unter dem Begriff Datenstruktur wird hier die Struktur einer Worldmap im Hauptspeicher verstanden, über die die auf Worldmaps angewandten Operationen erfolgen. Die Datenstruktur als solche spielt eine entscheidende Rolle bei der Performanzmessung. Eine sehr performanzorientierte Datenstruktur kann eine komplizierte Verwaltung im Hauptspeicher verursachen. Umgekehrt verlangt eine hierarchische Datenstruktur gewiß mehr Zeit für die Transaktionen, da die Daten zweckmäßig in kleineren Mengen gruppiert und nacheinander in mehreren Schritten geladen werden müssen. Um eine Worldmap in den Speicher zu laden, müssen die Daten von verschiedenen Relationen geladen und in der definierten Struktur im Hauptspeicher abgelegt werden. Dabei werden alle Spalten der Relationen gleichzeitig ausgelesen. Um solche Leseoperationen zu unterstützen, werden Strukturen als Datentypen definiert, deren Felder den Spalten der zugehörigen Relation entsprechen. Bei den Relationen, in denen NULL-Werte erlaubt sind, müssen zusätzlich noch INDICATOR-Strukturen definiert werden. Im folgenden werden Datentypen für die auf Relation SEGMENT angewandte Operationen definiert:

```
typedef struct {
    int   segmid;
    int   planid;
    int   wmapid;
    char  name[17];
    long  version;
    char  comm[80];
} segment_t;
```

```
typedef struct {
    short segmid_ind, planid_ind, wmapid_ind,
          name_ind, version_ind, comm_ind;
} segment_ind_t;
```

Jede Datenstruktur für Worldmaps baut auf Feldern solcher Datentypen. Die Frage hier ist nur, wie werden diese einzelnen Datenfelder in der Worldmap-Datenstruktur organisiert.

Die einfachste Datenstruktur ist die, in der eine Worldmap in unabhängigen Listen ihrer Bestandteile abgespeichert wird, d.h. es gibt eine Liste von Planes, eine Liste von Segmenten, eine Liste von Objekten, eine Liste von Message-Variablen, usw. Die Daten werden nicht in ihrer Abhängigkeit voneinander gespeichert, sondern einfach so, wie sie in verschiedenen Relationen der Datenbank vorkommen. Diese Datenstruktur gibt wohl die beste Performanz, besonders beim Laden einer Worldmap. Der Grund dafür ist, daß der Ladeprozeß nur aus wenigen Tabellenanfragen zusammenbaut und man mit einer Anfrage alle Datensätze aus einer Tabelle herauslesen kann, die zur aktuellen Worldmap gehören. Das Problem ist aber riesig, denn man muß im Speicher nach zusammenhängenden Elementen durchsuchen, z.B. welche Graphikrepräsentation, welche Variablenbeschreibung ein Objekt hat oder welche Objekte zu einem Segment gehören.

Eine bessere Datenstruktur wäre, die Daten zu einem Objekt zu gruppieren und zusammen zu speichern. Eine einheitliche Darstellung für ein Objekt besteht zuerst aus seiner eigenen Beschreibung, wie sie in der Relation OBJECT enthalten ist. Außerdem gehören dazu noch eine Graphikrepräsentation, die am Bildschirm gezeichnet wird, und eine Variablenbeschreibung, wenn das Objekt eine Variable ist.

```
typedef struct {
    object_t       object; object_ind_t object_ind;
    graphicdata_t  graphicpart;
    variabledata_t variablepart;
} objectdata_t;
```

Die Worldmap-Struktur besteht dann aus einer Liste von Planes, einer Liste von Segmenten, und einer Liste von Objekten. Der Ladeprozeß für diese Datenstruktur ist schon einbißchen langsamer, weil zu jedem Objekt eine zusätzliche Anfrage für die Graphikrepräsentation und noch eine Anfrage für die Variablenbeschreibung, wenn es ein dynamisches Objekt ist, nötig sind. Die Möglichkeit, die Objektdaten zusammen mittels einer Table-Join-Anfrage zu laden besteht hier nicht, da die Graphikrepräsentationen sowie die Variablenbeschreibungen der Objekte in unterschiedlichen Tabellen liegen.

Die am besten organisierte Datenstruktur ist die, in der Objekte, als Einheiten mit Graphikrepräsentation und Variablenbeschreibung, unter ihrem zugehörigen Segment, und Segmente wiederum unter ihrer zugehörigen Plane abgelegt werden:

```
typedef struct {
    segment_t     segment;   segment_ind_t   segment_ind;
    objectdata_t *objects;
    int           objnum;
} segmdata_t;

typedef struct {
    plane_t      plane;     plane_ind_t     plane_ind;
    segmdata_t  *segments;
```

```
    short        segmnum;
} plandata_t;

typedef struct {
    worldmap_t   worldmap;  worldmap_ind_t  worldmap_ind;
    plandata_t  *planes;
    short        plannum;
    glofigrep_t *glofigs;
    short        gfignum;
} wmapdata_t;
```

Die Objekte müssen jetzt per Segment geladen werden. Ähnlich dazu müssen die Daten zu den Segmenten auch per Plane geladen werden. So entsteht eine Menge zusätzlicher Datenbankanfragen bzw. hoher Sortieraufwand, wenn man trotzdem alles auf einmal laden möchte. Dies führt dazu, daß diese Datenstruktur die meiste Zeit für die Worldmap-Operationen im Vergleich zu den oben genannten Datenstrukturen fordert.

Diese Datenstruktur wird jedoch ausgewählt, um die Testfunktionen an der Worldmap-Datenbank zu realisieren und durchzuführen. Der Grund dafür ist, daß eine hierarchisch organisierte Struktur der Worldmaps für die Implementation der Worldmap-Operationen übersichtlicher und einfacher ist und deshalb später in den neuen Applikationen gewünscht wird, in denen Worldmaps zu bearbeiten sind. Außerdem macht die Gruppierung der zueinander gehörenden Daten es möglich, Operationen, die auf unabhängige Teile einer Worldmap (z.B. Plane oder Segment) gerichtet sind, parallel ausführen zu können.

### 4.3.2 Testverfahren

Das Hauptziel der Untersuchungen ist, Aussage darüber machen zu können, wie das entworfene Datenschema auf verschiedene Worldmap-Operationen (Laden, Löschen, Abspeichern) unter dem Gesichtspunkt der Performanz reagiert. Es wird besonders interessant sein, wenn diese Operationen auf eine große Worldmap angewendet werden. Im System sind keine sehr großen Worldmaps vorhanden. Die größten haben eine Größe von nur ein paar tausend Objekten (3 000). Der Ausweg hier ist, sie zuerst ins neue Datenschema zu konvertieren und dann übereinander zu kopieren. Man kann sich nun Worldmaps von realistischeren Größen (mit 10 000 - 20 000 Objekten) schaffen. Die Verteilung von verschiedenen Worldmap-Elementen wird dadurch auch negativ beeinflußt, aber die Dauer des Ladevorgangs liegt nicht daran, daß diese Worldmap mehr Objekte mit Text oder mit Polygonzug als Graphikrepräsentation, mehr Message-Variablenbeschreibungen als normal haben. Das Entscheidende ist leicht einzusehen: die Dauer liegt daran, wieviele Objekte, wieviele Graphikrepräsentationen bzw. wieviele Variablenbeschreibungen insgesamt zu laden sind.

Die drei Worldmap-Operationen Laden, Löschen und Abspeichern müssen als freie Funktionen implementiert werden und können beliebig aufgerufen werden. Die Dauer dieser Operationen wird dann allein entscheiden, wie gut ein Datenschema ist und ob es auch den Performanzanforderungen genügt. Da das Löschen und Abspeichern von Worldmaps ein nicht so hohe Priorität wie das Laden ist, konzentrieren sich die Untersuchungen meistens auf den Ladeprozeß. Die datenmanipulierenden Operationen (DELETE, INSERT) hängen mit dem Aufruf der Trigger zusammen. Indem man sie ausschaltet (ALTER TABLE *tab* DISABLE ALL TRIGGERS), kann man zu einer besseren Performanz für das Löschen bzw. Abspeichern von Worldmaps kommen. Aber dies entspricht nicht der Rea-

lität, in der die Datenbank betrieben wird. Nur für eine Untersuchung, wie der Zeitanspruch von Datenbanktriggern ist, kann es interessant sein.

Der Ladeprozeß ist selbst in mehreren Stufen zu realisieren. Mann kann die Zeit messen, die der Ladevorgang in einer bestimmten Stufe verbringt. Dadurch kann man Aussage darüber machen, für welchen Teil der Worldmap der Vorgang besonders viel Zeit bzw. wenig Zeit beansprucht. Zeitmessungen sind für die Routinen durchzuführen, in denen von den Worldmap-Daten nur Segmente, oder Segmente einschließlich der Objektbeschreibungen, oder Objekte mit bzw. ohne Graphikrepräsentation, mit bzw. ohne Variablenbeschreibung geladen werden. Für diese Untersuchung ist eine hierarchische Worldmap-Struktur im Hauptspeicher vorausgesetzt.

Es soll noch getestet werden, wie der aktuelle Zustand des Servers (Parametrierung) auf die Operationen wirkt. Besonders interessant ist, wie groß der Einfluß der Struktur der Worldmap-Daten im Hintergrundsspeicher auf die Operationen ist. Dazu gehören die Performanzaussagen in den Fällen, wo die Tabellen und Indizes wie voreingestellt bzw. optimiert abgespeichert werden. Dabei spielen sicherlich die Größe einzelner Tabellen bzw. die Größe der zu ladenden Worldmap eine entscheidende Rolle. In einer günstigeren Lage der Datenbank, z.B. im Falle, wenn die zur Datenbank gehörenden Dateien auf verschiedene Festplatten verteilt werden, werden die datenmanipulierenden Operationen (Löschen oder Abspeichern von Worldmaps) sicher viel von der I/O-Performanz des gesamten Systems profitieren können. Aber das Laden einer Worldmap ist nicht datenmanipulierend, das bedeutet, keine Redo-Informationen müssen abgespeichert werden, keine Datenblöcke müssen geschrieben bzw. aktualisiert werden. Dabei werden nur Datenblöcke in den Hauptspeicher geladen. Im Falle von sehr großen Worldmaps, wobei große Mengen von Daten von der Datenbank herauszulesen sind, kann der Ladeprozeß sicherlich auch viel Performanz gewinnen, wenn man die Indizes und die Daten getrennt voneinander auf verschiedenen Festplatten plaziert. Das Optimale wäre bei einer parallel konfigurierten ORACLE-Datenbank erreicht. Da ein Datenbestand bzw. gleiche Datenbestände von mehreren ORACLE-Instanzen verwaltet werden, kann der Ladeprozeß in parallele Anfragen aufgeteilt werden. Dazu ist aber auch der Aufwand zu berücksichtigen, Prozesse zu synchronisieren und gegeneinander zu sperren, um die Konsistenz der Daten zu halten.

Die SQL-Statements in den Testfunktionen sollen auf das Notwendigste reduziert werden. Verschiedene Anfragen werden soweit wie möglich kombiniert bzw. als CURSOR-Anfragen realisiert. Dadurch werden der Aufwand und die Zeitkosten darauf beschränkt, was die Operationen auch wirklich nötig haben.

# 5 Testumgebung

## 5.1 Hardwareumgebung und Datenbanksystem

Die Performanztests werden auf verschiedenen Maschinen mit folgenden Konfigurationen durchgeführt:

- SYSTEM 1

  Hardware: SUN ULTRA2

  - Betriebssystem : SunOS Version 5.5.1
  - Prozessor : 2 x 168 MHz ultrasparc Prozessor

  - Hauptspeicher : 512 MB
  - Festplatte : 9,0 GB intern

- SYSTEM 2

  Hardware: SUN ULTRA2

  - Betriebssystem : SunOS Version 5.5.1
  - Prozessor : 1 x 271 MHz ultrasparc Prozessor
  - Hauptspeicher : 256 MB
  - Festplatte : 9,0 GB intern

Auf beiden Systemen werden die Datenbank automatisch erzeugt. Deshalb haben sie folgende Gemeinsamkeiten:

- Datenbank: ORACLE Server Version 7.3.2.3.0
  - Die Festplatte wird in 4 Partitionen geteilt. Auf einer Partition liegt der Datenbankserver mit seinen Kontrolldateien. Auf anderen Partitionen liegen verteilt die Rollback-, Index-, Nutzer- bzw. andere Tablespaces. Die gesamte Datenbank liegt eigentlich auf einer einzigen Festplatte, was gegen die Richtlinien für ein schnelles Datenbanksystem spricht. Der Grund dafür liegt im Mangel an Anschlüssen bzw. an zur Verfügung stehender Hardware.
  - Der Server arbeitet im *Single Instance*-Modus, d.h nur eine ORACLE-Instanz verwaltet die Datenbank.
  - Die Kommunikation zwischen Server und Client erfolgt über das *Two Task*-Protokoll. Die Server- und Clientprozesse sind selbständige Prozesse.
  - Die Serverprozesse werden als *dedicated* Prozesse konfiguriert. Der Client-Prozeß kommuniziert direkt mit dem Server-Prozeß, um Daten zu laden bzw. zu manipulieren. Dadurch kann man die Systemresourcen am besten ausnutzen, da jeder Server-Prozeß sich nur mit einem Client-Prozeß beschäftigt und nicht mit mehreren Client-Prozessen wie im Falle der *shared*-Konfiguration.

## 5.2 Softwareumgebung und Werkzeuge

Die Tests werden direkt auf der Maschine durchgeführt, auf der sich der Datenbankserver (die ORACLE-Instanz) befindet. Zur Kommunikation zwischen Server- und Clientprozessen wird zwar SQL*Net herangezogen aber der Netzwerkoverhead wird minimal gehalten. Der Zeitaufwand wird entsprechend auf die Zeiten für die Kommunikation zwischen Prozessen auf derselben Maschine reduziert. Daraus ist zu schließen, daß die Ergebnisse bei einer Client/Server-Umgebung, in der die Kommunikation zwischen Client- und Serverprozessen über ein Netzwerk erfolgen muß, schlechter sein werden.

Normalerweise werden die Tests in einem ruhigen Zustand der Maschine durchgeführt (meistens in der Nacht), d.h. die anderen Prozesse, die sich im System befinden, haben den *sleep*-Status und im Hauptspeicher ist noch genügend freier Platz für Testprogramme übriggeblieben, um Swapping und Paging zu minimieren. Die Prozessorbelastung vor dem Start der Testprogramme ist in der Regel vernachlässigbar (1-5%). Das Ziel ist, zu Ergebnissen zu kommen, die nicht unter großen Schwankungen

leiden. Mit Hilfe von den Ergebnissen kann man dann Annahmen für verschiedene Worldmap-Größen treffen.

Alle Testfunktionen bzw. -programme werden in C entwickelt. Die Quellendateien werden zuerst vom ORACLE PROC/C++ Precompiler geparst, um die eingebetteten SQL-Statements in den entsprechenden C-Code umzuwandeln. Die entstehenden C-Dateien kann man dann wie normal übersetzen und linken.

Die Zeiten, die die Testprogramme als Meßergebnisse ausgeben, werden mit Hilfe der C-Funktion *gethrtime* (get high resolution time) erfaßt. Diese Funktion, die in der C-Bibliothek des Betriebssystems zur Verfügung steht, liefert die aktuelle Zeit in Genauigkeit von Nanosekunden als deren Anzahl von einem willkürlichen Zeitpunkt in der Vergangenheit. Das Ergebnis wird in einer 64-bit (long long) mit Vorzeichen besetzten Integerzahl speichert. Die Funktion eignet sich idealerweise für Performanzmessungen, wo ein praktisches und akkurates Verfahren benötigt wird. Im folgenden Programmfragment wird z.B. berechnet, wieviel Zeit im Durchschnitt ein *getpid*-Aufruf kostet:

```
hrtime_t start, end;
int i, iters = 100;

start = gethrtime();
for (i = 0; i < iters; i++) getpid();
end = gethrtime();
printf("Avg getpid() time = %lld nsec\n", (end - start) / iters);
```

Für Zwecke unserer Messungen genügt eine Genauigkeit in Millisekunden.

Während der Laufzeit der Testprogramme werden verschiedene Programme zur Beobachtung des Verhaltens des Betriebssystems benutzt. Das Programm *top* zeigt allgemeine Informationen zu laufenden Prozessen an:

```
im1:/home/jupi/do/Diplom>top
last pid: 21783;  load averages:  0.84,  0.58,  0.36
341 processes: 335 sleeping, 3 stopped, 3 on cpu
CPU states:  1.7% idle, 15.9% user, 17.5% kernel, 64.9% iowait,  0.0% swap
Memory: 498M real, 12M free, 584M swap, 708M free swap

   PID USERNAME PRI NICE  SIZE   RES STATE   TIME   WCPU    CPU COMMAND
21618 okeeffe    3    0   44M   37M cpu     0:32 20.98% 16.82% oracle
21616 do        16    0 4488K 3720K cpu     0:13  5.29%  4.84% performance
21677 do        18    0 1640K 1536K cpu     0:00  0.87%  0.88% top
```

Wichtige Punkte sind erstens CPU-Status, der zeigt, wie die Prozessorleistung auf verschiedene Systemaufgaben verteilt ist, und zweitens der Prozentsatz des Prozessoranspruchs für die Prozesse, die an der Testaktion beteiligt sind: der Serverprozeß *oracle* und der Clientprozeß *performance*.

Die Möglichkeit dazu, die Aktivitäten des Betriebssystems in bezug auf die Speicherverwaltung zu beobachten, bietet u.a. der Befehl *vmstat* an:

```
im1:/home/jupi/do/Diplom> vmstat -S 5 5
procs     memory            page            disk          faults      cpu
r b w   swap  free  si so pi po fr de sr s0 s1 s4 s5   in   sy   cs us sy id
0 0 24  2384  1408   0  0 16  6 10  0  2  1  0  2  6  240  213  290  8  3 90
0 0 15 698312 50360  0  0 108 0  0  0  0  0  0  1 18  596 3578  361 79 10 11
```

```
0 0 15 697152 47248  0  0 196 0  0  0  0  1  0 14 22  729 3048  411 63 14 22
0 0 15 708464 56032  0  0  0  0  0  0  0 19  0  8 41  825 2921  501 51  7 42
0 0 15 708336 55920  0  0  0  0  0  0  0  1  0  0  4  541 3417  391 48  5 47
```

In unserem Beispiel werden Systeminformationen fünf Mal in Intervallen von fünf Sekunden gesammelt. Interessant sind die Werte *memory free*, der angibt, wieviel Seiten von 1 KB Größe im Hauptspeicher noch frei sind, *page si*, *page so* (swap-ins, swap-outs) bzw. *page pi, page po* (page-ins, page-outs), die angeben, wieviel Seiten von 1 KB Größe eingeladen bzw. ausgeladen werden, und *cpu sy* (cpu system), der angibt, wie der CPU belastet wird, um Systemaktivitäten wie Swapping und Paging durchzuführen. Der aktuelle Zustand ist ganz gut, da das System noch ziemlich viel freien Hauptspeicher (55 MB) hat und der Prozessor nicht mit sehr viel Systemaktivitäten belastet wird (5%).

Mit dem Befehl *iostat* kann man den aktuellen Zustand der I/O-Aktivitäten im Betriebssystem erfahren:

```
im1:/home/jupi/do/Diplom> iostat -xtc 5 1
                    extended disk statistics       tty         cpu
disk   r/s   w/s    Kr/s    Kw/s wait actv  svc_t  %w  %b  tin tout us sy wt id
sd0    0.0   0.0     0.0     0.0  0.0  0.0    0.0   0   0    0   85 48 37 15  0
sd1    0.0   0.0     0.0     0.0  0.0  0.0    0.0   0   0
sd4   17.8  56.0   152.6   446.8  0.0  2.4   32.7   0  90
sd5    9.4   4.4    48.8    86.8  0.0  0.2   12.4   0  13
```

Hier wird eine detaillierte Statistik über einem Intervall von fünf Sekunden erstellt. Auf die Platte *sd4* wird am meisten zugegriffen. Es erfolgen durchschnittlich 17 Read- und 56 Write-Operationen pro Sekunde. Die Transferraten betragen pro Sekunde 152 KB bei Read- und 446 KB bei Write-Operationen. Die durchschnittliche verstrichene Zeit für eine Operation beträgt 32 Millisekunden (*svc_t*, service time). Die Zeit, in der die Festplatte aktiv gewesen ist, beträgt 90% (*%b*, busy) und die Zeit, die das Betriebssystem dazu verbracht hat, auf die I/O-Aktivitäten zu warten, beträgt 15% (*cpu wt*, cpu wait) des gesamten Intervalls. Mit Hilfe dieses Programms kann man den Festplattenzugriff in verschiedenen Datenbanktransaktionen analysieren und dementsprechend die Dateien auf die Festplatten verteilen, um eine bessere Lastbalancierung aller im System vorhandenen Platten zu erzielen.

ORACLE stellt seinerseits dem Anwender auch verschiedene Hilfsmittel zur Verfügung, die Informationen über den Ablauf der internen Vorgänge liefern, wie z.B, den EXPLAIN PLAN-Befehl, mit dem man sich den Abarbeitungsweg eines SQL-Statements, genau so wie der Optimierer ihn dann ausführt, anzeigen lassen kann, oder den SQL_TRACE-Mechanismus, mit dem man die Performanzinformationen von den ausgeführten SQL-Statements in einer eigenen Datei für spätere Analyse festhalten kann.

## 5.3 Testdaten

Der Generierung der neuen Worldmaps, die dann zu Testzwecken verwendet werden, liegt der Aufbau der zwei größten im System zur Verfügung stehenden Worldmaps mit den Nummern 381 (*110 KV*) und 299 (*US Net*) zugrunde. Sie werden unter den Nummern 0 und 1 in unserer Testdatenbank abgespeichert. Um zu einer noch größeren Worldmap zu kommen, werden diese Worldmaps zuerst übereinander kopiert und unter einer neuen Worldmap-Nummer abgelegt (2). Diese neu erzeugte Worldmap wird als Ausgangsworldmap für Neugenerierung der größeren Worldmaps benutzt. Wir führen die Tests an folgenden Worldmaps durch:

Tabelle 6 Aufbau der Testworldmaps

| Bestandteil | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| *Planes* | *9* | *5* | *14* | *28* | *42* | *56* | *70* |
| *Segmente* | *17* | *16* | *33* | *66* | *99* | *132* | *165* |
| *Objekte* | *2 970* | *1 944* | *4 914* | *9 828* | *14 742* | *19 656* | *24 570* |
| *statischen Objekte* | *415* | *665* | *1 080* | *2 160* | *3 240* | *4 320* | *5 400* |
| *dynamischen Objekte* | *2 555* | *1279* | *3 834* | *7 668* | *11 502* | *15 336* | *19 170* |
| *Globalfiguren* | *60* | *10* | *67* | *67* | *67* | *67* | *67* |
| *Objekte mit Globalfigur-Graphikrep.* | *1 852* | *801* | *2 653* | *5 306* | *7 959* | *10 612* | *13 265* |
| *Objekte mit Lokalfigur-Graphikrep.* | *302* | *244* | *546* | *1 092* | *1 638* | *2 184* | *2 730* |
| *Objekte mit Polyline-Graphikrep.* | *161* | *121* | *282* | *564* | *846* | *1 128* | *1 410* |
| *Objekte mit Polygon-Graphikrep.* | *68* | *232* | *300* | *600* | *900* | *1 200* | *1 500* |
| *Objekte mit Text-Graphikrep.* | *543* | *538* | *1 081* | *2 162* | *3 243* | *4 324* | *5 405* |
| *Objekte mit Arc-Graphikrep.* | *31* | *8* | *39* | *78* | *117* | *156* | *195* |
| *Objekte mit Curve-Graphikrep.* | *13* | *0* | *13* | *26* | *39* | *52* | *65* |
| *Mask-Variablen* | *2* | *0* | *2* | *4* | *6* | *8* | *10* |
| *Message-Variable* | *1 635* | *532* | *2 167* | *4 334* | *6 501* | *8 668* | *10 835* |
| *Keybox-Variablen* | *45* | *69* | *114* | *228* | *342* | *456* | *570* |
| *Value-Variablen* | *860* | *678* | *1 538* | *3 076* | *4 614* | *5152* | *7 690* |

Für jede Testreihe werden also die zwei existierenden Worldmaps in eine neue Worldmap kopiert. Dadurch vergrößert sich die Datenbank immer um eine bestimmte Anzahl von Objekten, Graphik- bzw. Variablenbeschreibungen. Die Anzahl der Planes per Worldmap erhöht sich dabei und ist nicht mehr realistisch. Die Ladekosten für die Plane-Datensätze liegen aber im Bereich von wenigen Sekunden und spielen deshalb keine große Rolle für den gesamten Vorgang. Realistisch ist dagegen die Anzahl der Segmente per Worldmap, die in der Regel die Anzahl der Unterstationen darstellt, die zu visualisieren und zu kontrollieren sind. Die Anzahl der Objekte per Segment der Ausgangsworldmaps prägt auch die Anzahl der Objekte per Segment der generierten Worldmaps. Folgende Statistik geht aus dem Inhalt der Worldmaps hervor:

Tabelle 7 Bestandteile der Testworldmaps

| Verhältnis | Statistik |
|---|---|
| *Objekte / Segment* | *149* |
| *statische Objekte / Objekte* | *21 %* |
| *dynamische Objekte / Objekte* | *79 %* |
| *Objekte mit Globalfigur-Graphikrep. / Objekte* | *53 %* |
| *Objekte mit Lokalfigur-Graphikrep. / Objekte* | *11 %* |
| *Objekte mit Polyline-Graphikrep. / Objekte* | *0,5 %* |
| *Objekte mit Polygon-Graphikrep. / Objekte* | *0,6 %* |
| *Objekte mit Text-Graphikrep. / Objekte* | *21 %* |
| *Message-Variablen / Objekte* | *44 %* |

| Verhältnis | Statistik |
|---|---|
| *Keybox-Variablen / Objekte* | *2 %* |
| *Value-Variablen / Objekte* | *31 %* |

# 6 Durchführung der Tests und Ergebnisse

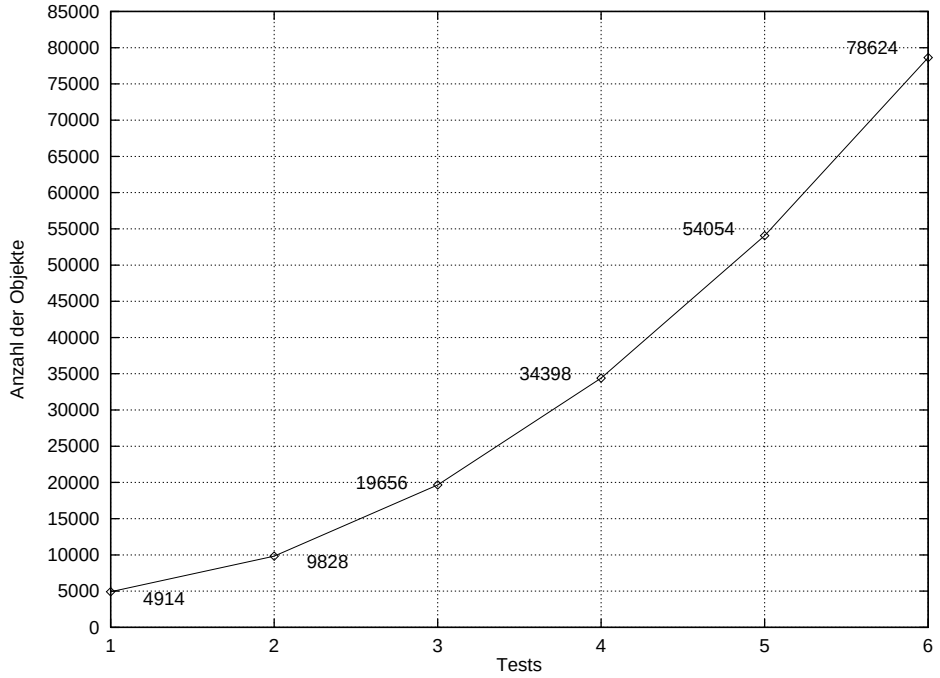
## 6.1 Die erste Testreihe

In der ersten Testreihe werden die Testfunktionen auf die Worldmap-Daten angewendet, ohne irgendeine Optimierung am Datenbanksystem vorgenommen zu haben. Es werden keine Änderungen an den Startparametern der ORACLE-Instanz vorgenommen, die in der Datei INIT.ORA zu finden sind. Da die Datenbank bereits Daten für andere Zwecke enthält und im Laufe der Zeit Optimierungen daran gemacht wurden, stellen die Parameter eine ziemlich gute Anpassung der Datenbank an das Betriebssystem dar. Nach der ersten Testreihe müssen die wichtigsten Parameter jedoch nochmals auf Vorteile und Nachteile analysiert werden, um weitere Optimierungen zu ermöglichen.

### 6.1.1 Der Datenbestand und das Verhalten der Meßzeiten

In der ersten Testreihe wird zu jedem Test eine neue Worldmap hinzugeneriert, die ein Vielfaches von der Ausgangsworldmap (2) ist. Das Vergrößern des Datenbestands der Datenbank wird im folgenden Bild gezeigt:

Abbildung 15 Der Datenbestand in der Datenbank in jedem Test




Die Anzahl der Objekte, die sich nach dem 6. Test in der Datenbank befinden, kann für die Objektanzahl einer sehr großen Worldmap stehen.

Die Tests haben ergeben, daß die Performanz der Operationen, ganze Worldmaps zu löschen oder einzufügen, in Anwendung auf das erste Datenschema sehr schlecht ist. Die notwendige Zeit für diese Operationen erreicht die Stunden-Grenze schon bei einer Worldmap mit ca. 10 000 Objekten. Der Grund dafür ist, daß das Datenbanksystem die zugehörigen Trigger jedes Mal aktiviert, wenn ein Datensatz einer Tabelle geändert wird, um die Konsistenz der Daten nachzuprüfen. Der Anteil der Globalfigur- bzw. der Lokalfigurgraphikrepräsentationen in Testworldmaps ist ziemlich klein im Vergleich zu anderen Graphikrepräsentationsformen (siehe Tabelle 6 ***Aufbau der Testworldmaps,*** Seite 60). Deshalb ist es zu vermuten, daß das zweite Schema, in dem die Applikationen die Globalfiguren und Lokalfiguren nur mit einem SQL-Befehl (INSERT, SELECT, DELETE) laden bzw. ändern können, auch keine sehr großen Performanzvorteile bringt. Aus diesem Grund wird nur das dritte Datenschema als Vergleichsschema zusammen mit dem ersten Datenschema implementiert. Die Tests werden parallel an beiden Schemata durchgeführt.

#### 6.1.1.1 Der Ladevorgang

In den Ladetests ist die Streuung der Meßzeiten vernachlässigbar. Die Meßzeiten in der an einer Worldmap durchgeführten Testserie unterscheiden sich nur um wenige Sekunden. Der Grund dafür ist, daß der Vorgang keine Schreibaktivitäten in der Datenbank verursacht. Die physikalische Struktur der Worldmap-Daten auf der Festplatte wird dabei nicht geändert. Der Server führt nur Lesezugriffe aus, welche nicht mit anderen Zugriffen konkurrieren müssen, die in der Datenbank stattfinden, wenn Daten geändert bzw. neu eingefügt werden.

Es ist zuerst zu beobachten, daß die Meßzeiten für die Operationen an derselben Worldmap nach jedem Test schlechter werden, d.h. der Datenbestand der Datenbank hat einen direkten Einfluß auf die Performanz der Operationen ausgeübt. Zur genaueren Untersuchung wird der Ladeprozeß in mehreren Stufen implementiert. Die Zeiten, die in den einzelnen Ladestufen gemessen werden, sagen dann aus, in welcher Stufe der Datenbestand am deutlichsten seine Wirkung auf die Operation zeigt. Im Falle des ersten Datenschemas muß die Applikation sowohl die graphische Beschreibung als auch die Variablenbeschreibung für einzelne Objekte laden. Im Abbildung 16 sieht man, daß die Zeit dafür, graphische Beschreibungen zu laden und die Zeit dafür, Variablenbeschreibungen zu laden für ein und dieselbe Worldmap über die Tests konstant bleiben. Dagegen wächst die Zeit dafür, die allgemeinen Objektbeschreibungen zu laden.

Abbildung 16 Zeitkosten für Ladestufen im 1. Datenschema[1]

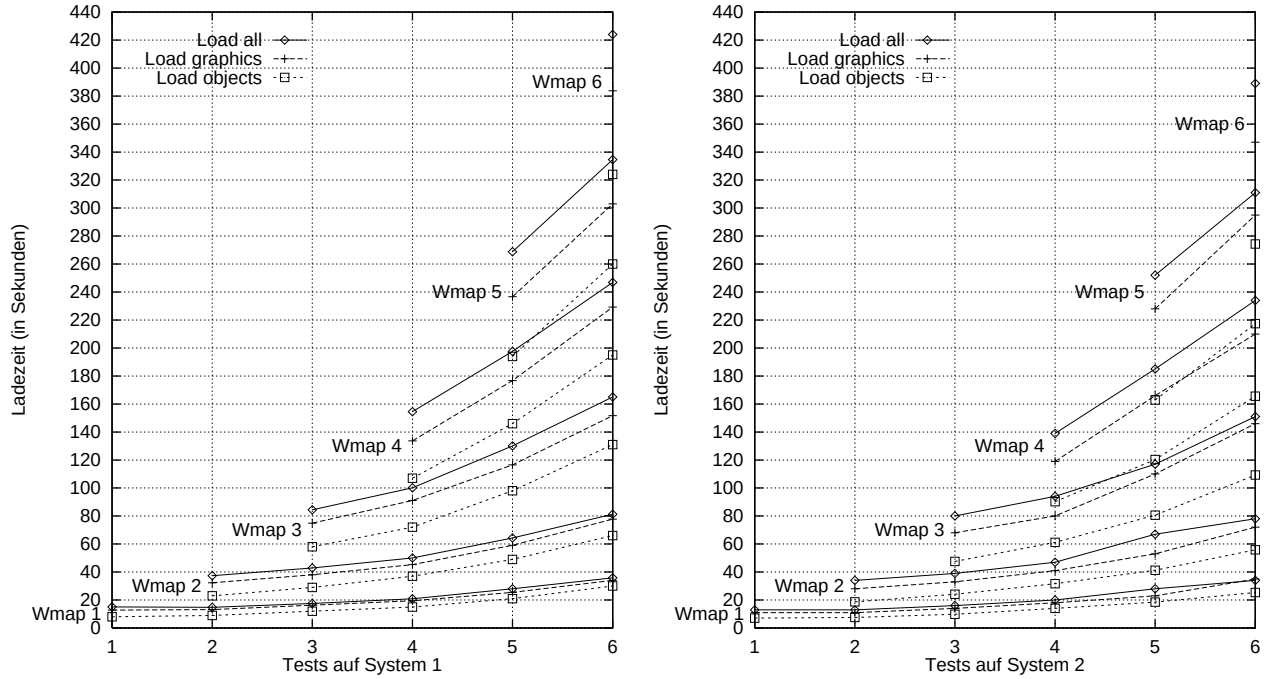
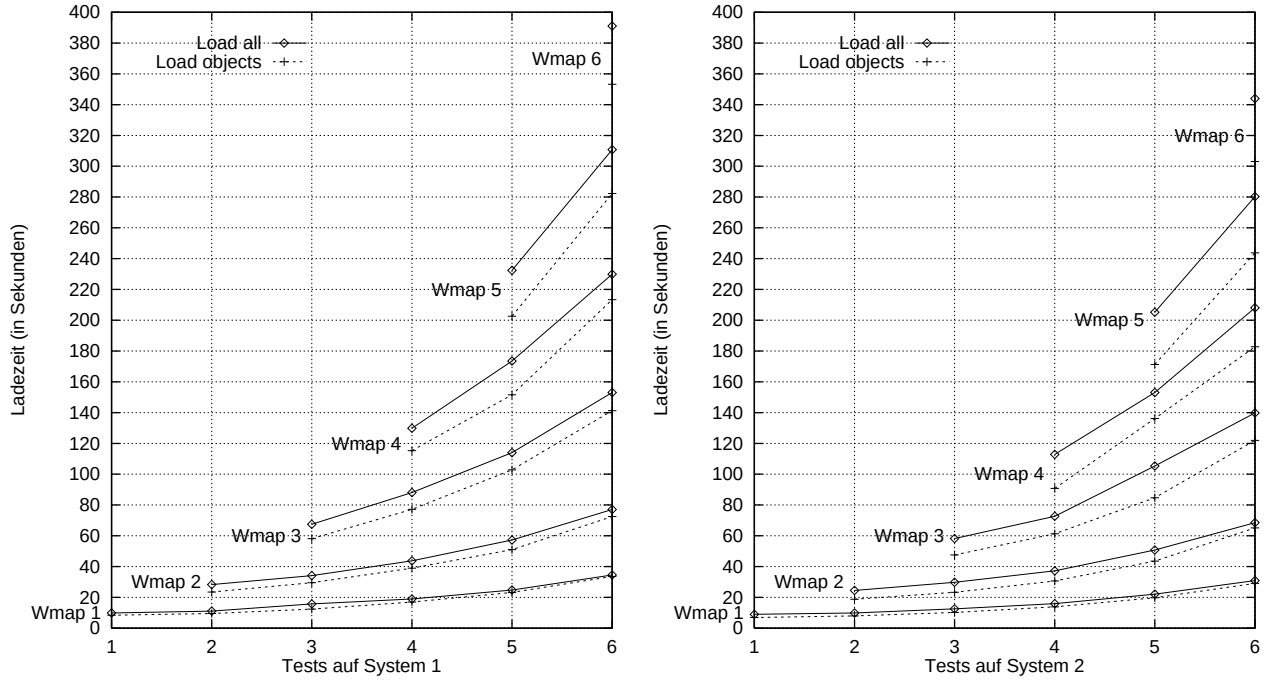



Abbildung 17 Zeitkosten für Ladestufen im 3. Datenschema[2]




1 Die Ergebnisse für Worldmap 0 werden nicht im Diagramm dargestellt, da die Kurven für Worldmap 0 und 1 stark ineinander laufen und nicht mehr deutlich voneinander zu unterscheiden sind.

2 Das Diagramm enthält nicht die Testergebnisse für Worldmap 0.

Im dritten Datenschema werden die Graphikbeschreibungen schon in einer LONG RAW-Spalte der Tabelle OBJECT abgespeichert. Deshalb braucht die Applikation sie nicht durch zusätzliche Datenbankzugriffe zu laden. Außerdem gilt, daß die Zeit dafür, die Variablenbeschreibungen der Objekte ein und derselben Worldmap zu laden, in beiden Datenschemata gleich bleibt (Abbildung 17). Für die Untersuchung auf Ladezeiten der Worldmaps werden die Worldmaps in den Tests nicht geändert. Daraus folgt, daß die physikalische Speicherlage der Worldmap-Daten unmodifiziert bleibt. Das extreme Wachstum der Tabelle OBJECT (Abbildung 15) ist deshalb verantwortlich für die Verschlechterung der Ladezeiten der Worldmaps. Die Zeit dafür, die OBJECT-Datensätze zu laden, macht auch den größten Teil der gesamten Ladezeit einer Worldmap aus.

Aufgrund des belastungsfreien Zustands der Testmaschine ist es möglich, eine weitere Regelmäßigkeit zu beobachten. Die Zeit dafür, gleiche Datenmengen von einem konstanten Datenbestand der Datenbank (in einem Test) zu laden, ist gleich. Die (stufenweise) Ladezeit einer Worldmap, die durch Kopieren von anderen Worldmaps entstand, ist die Summe der (stufenweisen) Ladezeiten der Ausgangsworldmaps. Aus dem 6. Test auf dem Testsystem 1 mit dem ersten Datenschema gehen folgende Werte hervor:

Tabelle 8 Das 6. Test auf System 1 mit dem 1. Datenschema: Laden

| Worldmap | Load all | Load graphics | Load objects | Load segments |
|---|---|---|---|---|
| *0* | *46,935187* | *43,134802* | *38,295978* | *2,544045* |
| *1* | *35,831823* | *34,001215* | *30,189495* | *1,961508* |
| *2* | *81,291943* | *77,772200* | *66,884353* | *2,740237* |
| *3* | *165,025793* | *151,708690* | *131,988789* | *3,127563* |
| *4* | *246,864862* | *229,224188* | *195,384022* | *3,379453* |
| *5* | *334,581263* | *302,957750* | *260,902914* | *3,675800* |
| *6* | *423,960804* | *383,758302* | *324,503276* | *4,022272* |

Aus Worldmaps 0 und 1 ist Worldmap 2 entstanden. Aus einem Vielfachen von Worldmap 2 sind die anderen Worldmaps entstanden. Dies spiegelt sich auch in den Meßzeiten wider. Es geht darauf zurück, daß die gleiche Anzahl von Zugriffen für die gleiche Datenmenge benötigt wird. Die Zugriffszeiten werden schlechter, wenn die Tabellen größer werden, und bleiben auch konstant (konvergieren), wenn der Datenbestand in den Tabellen sich nicht ändert.

#### 6.1.1.2 Der Lösch- und Einfügevorgang

Die Wordmaps werden in den Ladetests nicht manipuliert, d.h. sie werden einmal generiert und die Ladefunktionen werden darauf angewendet. Dies hat große Performanzvorteile für den Ladevorgang gebracht. Beim Generieren bzw. Anlegen einer Worldmap muß der Datenbankserver immer Speicherplatz in Tablespaces für neue Datensätze allokieren, weil die Extents noch ganz leer oder die allokierten Datenblöcke schon voll sind. Die Situation ist sehr günstig für die komplette Abspeicherung einer Worldmap: ORACLE füllt die neuen leeren Blöcke nacheinander mit Daten und läßt nur so viel Platz in jedem Block frei, wie der Parameter PCTFREE vorschreibt. Die Worldmap wird auf einer minimalen Anzahl von Datenblöcken abgelegt. Der Ladeprozeß braucht auch nur eine minimale Anzahl von Datenblöcken in den Hauptspeicher zu lesen. Die Manipulationen an wenigen Datensätzen können die gesamte physikalische Lage der Worldmap auch nicht sehr viel beeinflussen.

Das Löschen zerstört aber diese optimale Lage der Daten. Nach jedem DELETE bzw. UPDATE-Befehl prüft ORACLE den betroffenen Datenblock, ob der freie Platz schon die Grenze von PCTUSED erreicht, und hängt ihn an den Anfang der Free-Liste des Segments angehängt, die für spätere INSERT-Operationen zur Verfügung steht. Wenn eine Worldmap in die Datenbank zu importieren ist und eine große Datenmenge schon in der Datenbank vorliegt, besteht eine hohe Wahrscheinlichkeit, daß ORACLE von (halb-vollem) Datenblock zu Datenblock geht und neue Datensätze hineinschreibt. Die Daten liegen dann auf mehr Datenblöcken verstreut als in der Ausgangssituation. Das verursacht natürlich zusätzliche Festplattenaktivität für die auf diejenige Worldmap angewandten Operationen. Man kann z.B. die Anzahl der Datenblöcke der Tabelle OBJECT, die Datensätze einer bestimmten Worldmap enthalten, durch folgenden Befehl abfragen:

```
SQL> SELECT wmapid, COUNT(DISTINCT(SUBSTR(rowid, 1, 8)))
     FROM object
     GROUP BY wmapid;
```

Nach mehrmaliger Anwendung der Lösch- und Einfüge-Operationen auf verschiedene Worldmaps haben wir das folgende Ergebnis:

Tabelle 9 Anzahl der Datenblöcke für OBJECT-Datensätze von Testworldmaps

| Wmap | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| *1. Insert* | *210* | *137* | *349* | *698* | *1 048* | *1 397* | *1 745* |
| *2. Insert* | *219* | *142* | *362* | *724* | *1 086* | *1 447* | *1 809* |
| *3. Insert* | *217* | *141* | *361* | *722* | *1 082* | *1 441* | *1 804* |
| *4. Insert* | *217* | *143* | *360* | *721* | *1 081* | *1 442* | *1 800* |

Die Anzahl der Datenblöcke beim Anlegen der Worldmaps (1. Insert) ist am geringsten. Je größer die Worldmap ist, d.h. je mehr OBJECT-Datensätze zu der Worldmap gehören, desto schlechter kann ihre Festplattenbelegung nach dem Löschen und Einfügen werden.

Wie wir erwartet haben, ist das erste Datenschema besonders schlecht für UPDATE- und INSERT-Operationen geeignet. Die Trigger der Graphikprimitiv- und Variablenbeschreibungs-Tabellen prüfen, daß kein Datensatz gelöscht wird bzw. eine andere Nummer bekommt, während ihn ein Datensatz der Tabelle OBJECT noch referenziert. Der Trigger der Tabelle OBJECT prüft dagegen, daß ein eingefügter bzw. geänderter Datensatz immer eine gültige Referenz zu Graphikprimitiv- und Variablenbeschreibungs-Tabellen besitzt. Solche Referenzierungen sind normalerweise durch Fremdschlüssel zu realisieren und die Ausführungszeiten der Trigger werden somit vermieden. In unserer Modellierung liegt aber eine exklusive Beziehung von der referenzierenden Entität zu den referenzierten Entitäten vor, was die Definition von Fremdschlüsseln unmöglich macht. Aufgrund der vorauszusehenden hohen Zeitkosten werden die Operationen zum Löschen und Einfügen nicht auf Worldmaps 5 und 6 im ersten Datenschema angewendet. Für Worldmap 4 mit 14 742 Objekten braucht die Applikation schon eine unakzeptierbare Zeit: über eine Stunde.

Abbildung 18 Zeit zum Löschen der Worldmaps im 1. Datenschema

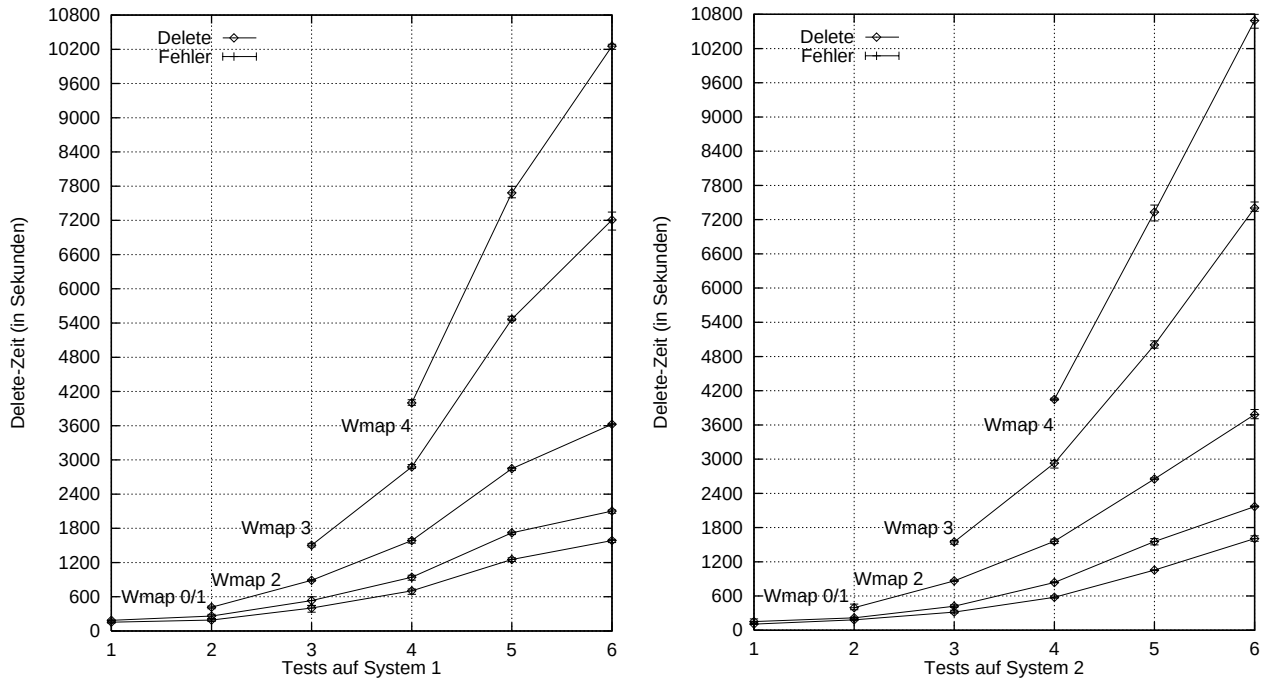
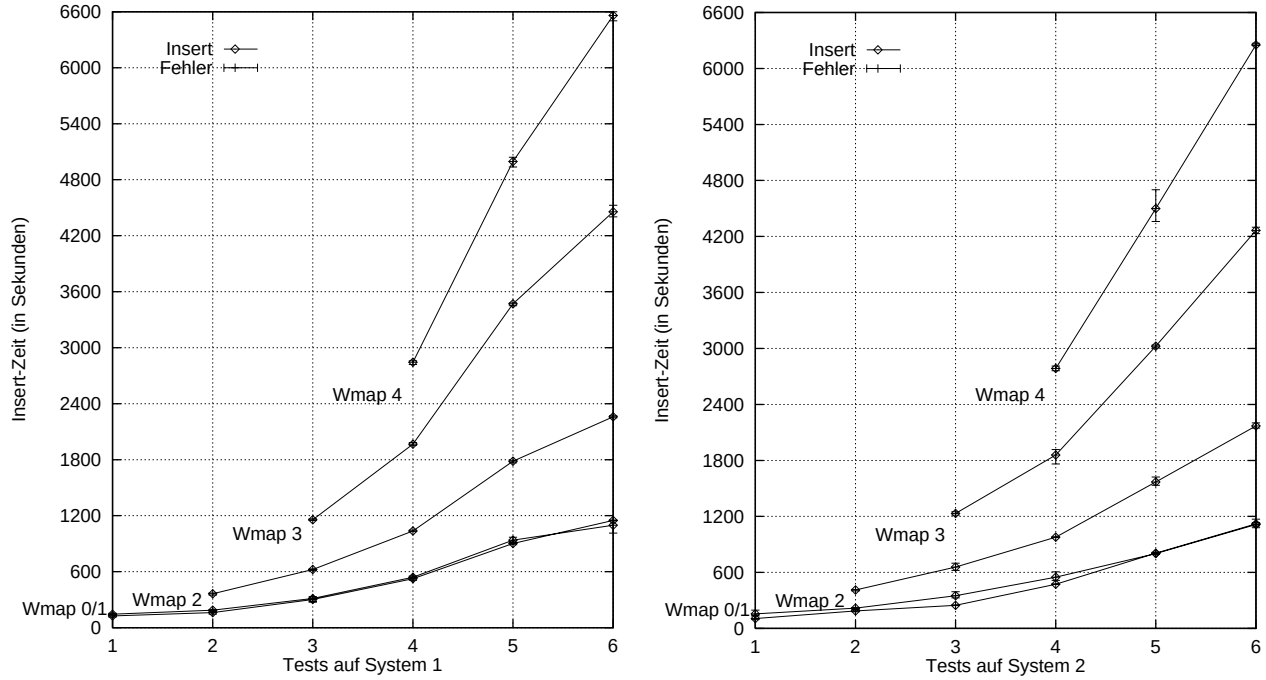



Abbildung 19 Zeit zum Einfügen der Worldmaps im 1. Datenschema




Mit dem dritten Datenschema sind auch große Performanzverbesserungen der DELETE- und INSERT-Operationen zu beobachten, da nur die Trigger der Tabelle OBJECT und der Variablenbeschreibungs-Tabellen dabei aktiv sind.

Abbildung 20 Zeit zum Löschen der Worldmaps im 3. Datenschema

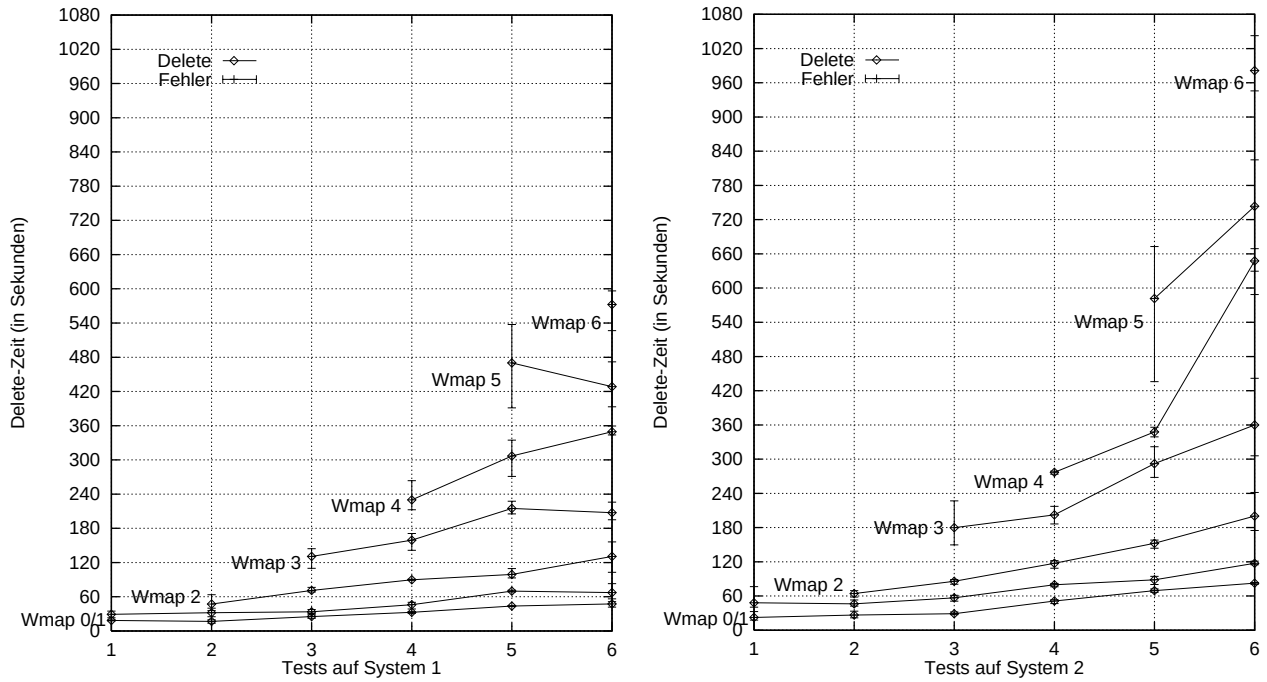
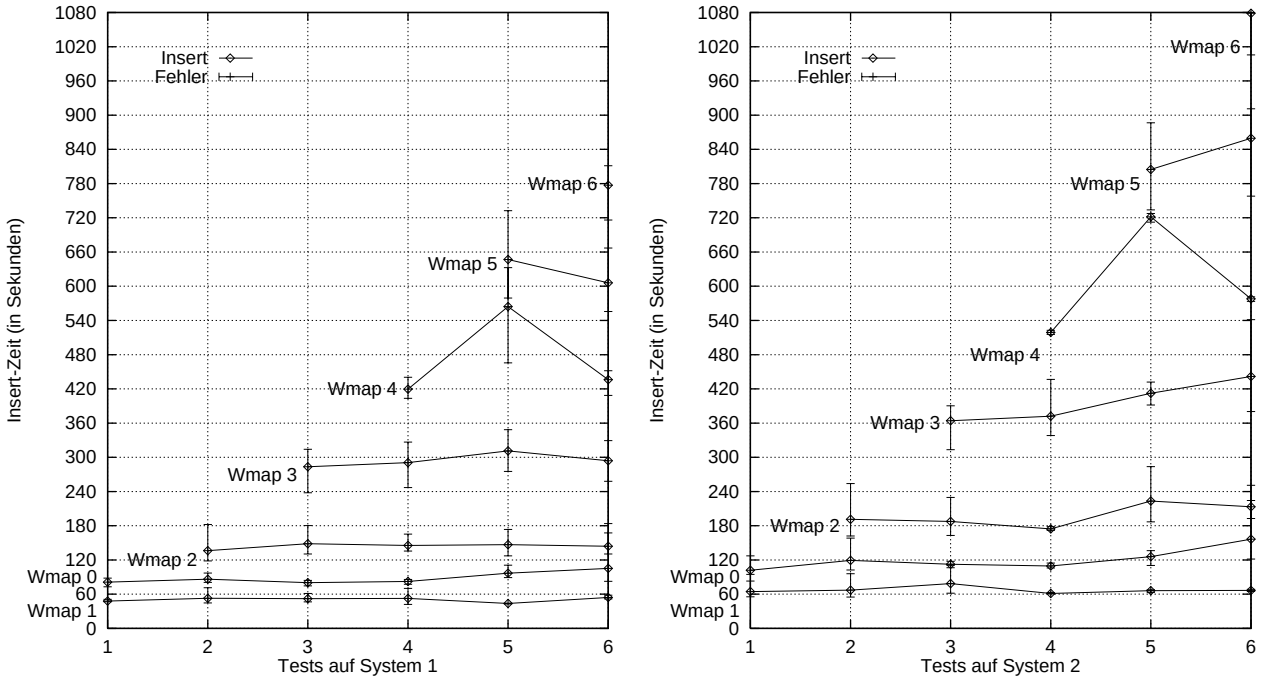



Abbildung 21 Zeit zum Einfügen der Worldmaps im 3. Datenschema




Die Meßzeiten weisen eine hohe Streuung auf. In den Diagrammen werden deshalb Toleranzbereiche (*Fehler*) zusätzlich zu den durchschnittlichen Werten gezeichnet. Hier sind einige Unregelmäßigkeiten zu beobachten. Die Zeiten verändern sich trotz des Wachtums des Datenbestands sehr wenig (bei Worldmap 0 und 1), es entsteht sogar eine Verbesserung des Durchschnittswerts beim Einfügen von

Worldmap 4. Der Fehlerbereich für die Meßzeiten wächst auch, wenn die zu löschende oder zu speichernde Worldmap größer wird.

In einem Test gilt es wieder, daß die Zeitkosten für die Worldmaps, die durch Kopieren von kleineren Worldmaps entstanden sind, in der Größenordnung der Summe der Zeiten liegt, die die Applikation bei den Teilworldmaps für die entsprechende Operation benötigt. Auf dem Testsystem 1 mit dem 3. Datenschema haben wir folgende Meßzeiten im 6. Test bekommen:

Tabelle 10 Das 6. Test auf System 2 mit dem 3. Datenschema: Löschen und Einfügen

| Worldmap | Delete-Zeit | Insert-Zeit |
|---|---|---|
| *0* | *67,252757* | *105,344351* |
| *1* | *47,451037* | *54,326331* |
| *2* | *130,629680* | *144,113619* |
| *3* | *207,394418* | *293,864093* |
| *4* | *349,357547* | *436,182919* |
| *5* | *428,480752* | *605,888933* |
| *6* | *572,431063* | *777,070895* |

### 6.1.2 Zusammenfassung der 1. Testreihe

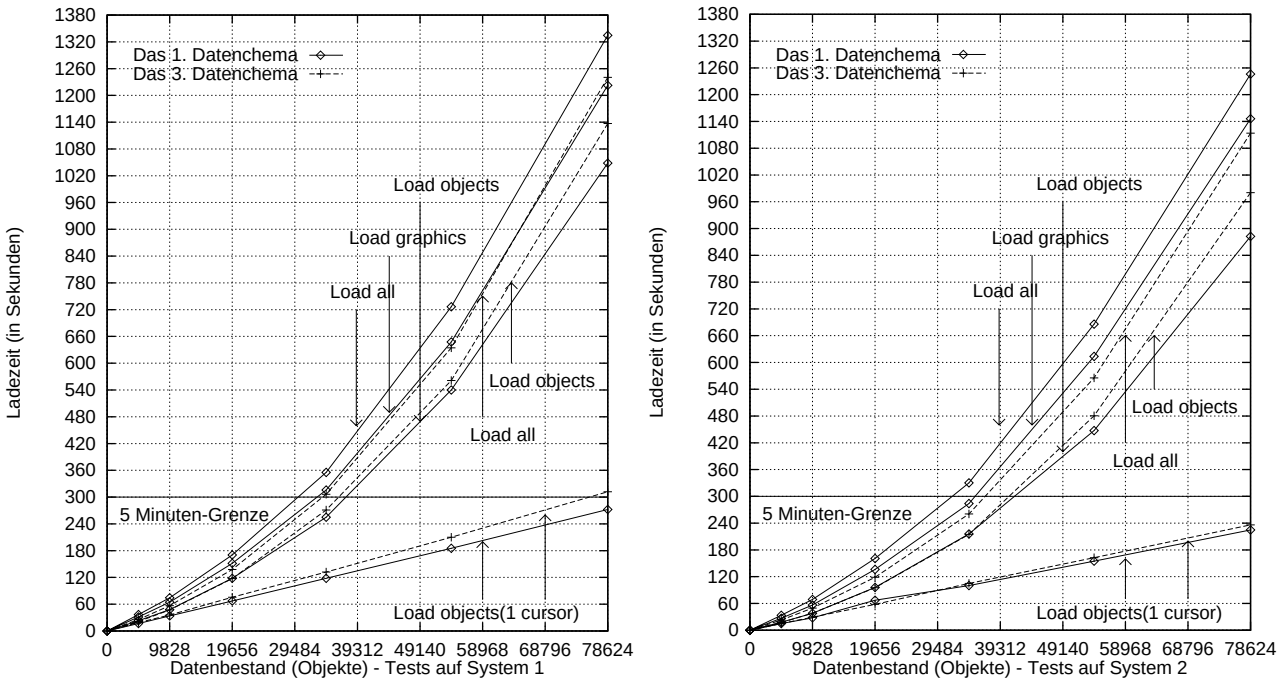
Unter der Annahme, daß der gesamte Datenbestand in der Datenbank in jedem Test nur zu einer einzigen Worldmap gehört, haben wir folgende Spezifikation für die Testworldmaps:

Tabelle 11 Worldmaps mit dem gesamten Datenbestand in einzelnen Tests

| Bestandteil | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| *Planes* | *14* | *28* | *56* | *98* | *154* | *224* |
| *Segmente* | *33* | *66* | *132* | *231* | *363* | *528* |
| *Objekte* | *4 914* | *9 828* | *19 656* | *34 398* | *54 054* | *78 624* |
| *statischen Objekte* | *1 080* | *2 160* | *4 320* | *7 560* | *11 880* | *17 280* |
| *dynamischen Objekte* | *3 834* | *7 668* | *15 336* | *26 838* | *42 174* | *61 344* |

Die Zeitkosten für den Ladeprozeß dieser Worldmaps werden durch die Summe der durchschnittlichen Ladezeiten aller Worldmaps in einzelnen Tests berechnet. Dabei ist die Zeit dafür, allgemeine Daten für Segmente und Planes zu laden, in der Regel vernachlässigbar im Vergleich zu den Ladezeiten in anderen Stufen. Sie beträgt z.B. im 6. Test, in dem alle Worldmaps 528 Segmente und 224 Planes zusammenbringen, insgesamt auch nur 21 Sekunden im ersten Datenschema bzw. 17 Sekunden im dritten Datenschema. Im folgenden Diagramm werden die anderen Ladestufen gezeichnet:

Abbildung 22 Ladezeit für den gesamten Datenbestand




Die notwendige Zeit dafür, die OBJECT-Datensätze im dritten Datenschema zu laden, hebt sich im Vergleich zum ersten Datenschema leicht ab. Das wird dadurch erklärt, daß die Ladefunktion für das dritte Datenschema noch andere Aktionen unternehmen muß, um die LONG RAW-Spalten so abzulegen, daß der Speicherplatz nicht verschwendet wird. Die Wirtsvariable, die für Speicherung der Werte einer LONG RAW-Spalte bestimmt ist, muß jede Länge akzeptieren, und diese kann im schlimmsten Fall zwei Gigabytes betragen. Die Zeitkosten dafür werden erst auffällig bei großer Anzahl der Wiederholungen dieser Aktionen (80 000). Der Ladevorgang erzielt bei den Worldmaps im dritten Datenschema insgesamt aber eine kleine Performanzverbesserung im Vergleich zu den Worldmaps im ersten Datenschema.

Die Spezifikation dieser Worldmaps weist eine hohe Anzahl der Segmente auf. Da der Abspeicherung der Worldmaps eine hierarchische Struktur im Hauptspeicher zugrundeliegt, wird die Ladefunktion so implementiert, daß zu jedem Segment ein Cursor geöffnet wird und die allgemeinen Objektbeschreibungen, die zu dem Segment gehören, mittels dieses Cursors geladen werden. Eine hohe Anzahl von Segmenten bedeutet auch eine hohe Anzahl von Cursor. Um einen Vergleich zu ermöglichen, wird noch untersucht, wie das Verhältnis zwischen der notwendigen Zeit und der Anzahl der Objekte ist, die nur in einem einzigen Cursor geladen werden. Man kann erkennen, daß die Zeit linear mit der Anzahl der Objekte anwächst, jedoch erst bei 80 000 Objekten die 5-Minuten-Grenze erreicht (Abbildung 22). Die großen Differenzen zu den Zeiten dafür, Objekte segmentsweise im ersten sowie im dritten Datenschema zu laden, sagen aus, wie kostspielig die Cursor beim hierarchischen Laden der Worldmap-Daten sind.

Die Ladezeiten für einzelne Worldmaps scheinen gegen ORACLE's Behauptung "Retrieval performance of indexed data remains almost constant, even as new rows are inserted" ([Oracle-SerCon]) zu sprechen, da sie in den Tests nicht gleich geblieben sind. Die Datensätze werden nur über den Primärschlüssel der OBJECT-Tabelle geladen, der aus den Worldmap-, Plane-, Segment- und Objekt-Num-

mern besteht und für den ein Index schon automatisch von ORACLE angelegt wird, um die Eindeutigkeit zu erzwingen. Der konkrete Grund für die Verschlechterung der Ladenzeiten der Worldmaps beim größeren Datenbestand besteht nun darin, daß ORACLE länger braucht, einen Cursor zu öffnen und die Daten daraus zu lesen, wenn die Tabelle gewachsen ist. Nach dem 6. Test mit dem dritten Datenschema haben wir folgende Ergebnisse:

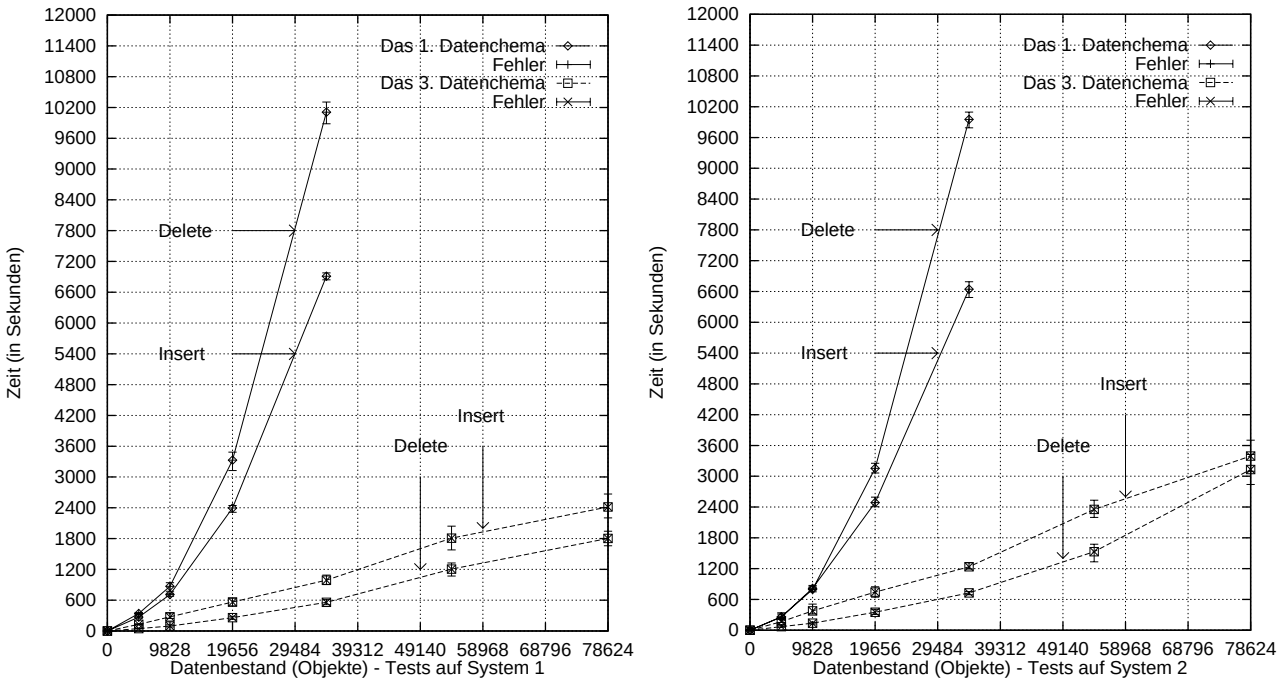
Tabelle 12 Datenbestand in der Tabelle OBJECT und Cursorzeit

| Test | Objekte | Rate | 1 Cursor | 1 Cur. / 1 Seg. | Cursor | Zeit / 1 Cur. | Rate |
|---|---|---|---|---|---|---|---|
| *1* | *4 914* | *1* | *17,307505* | *21,258932* | *33* | *0,119740* | *1* |
| *2* | *9 828* | *2* | *35,009449* | *47,270099* | *66* | *0,185767* | *1,5* |
| *3* | *19 656* | *4* | *70,553004* | *117,500021* | *132* | *0,355659* | *2,9* |
| *4* | *34 398* | *7* | *121,521217* | *271,126109* | *231* | *0,647640* | *5,4* |
| *5* | *54 054* | *11* | *193,193758* | *561,134959* | *363* | *1,013612* | *8,4* |
| *6* | *78 624* | *16* | *286,141880* | *1136,997200* | *528* | *1,611468* | *13,4* |

In der 4. Spalte sieht man, wie lange die Applikation braucht, um alle OBJECT-Datensätze in einem Cursor zu laden. In der 5. Spalte ist die notwendige Zeit, die gleiche Anzahl von Objekten aber segmentsweise zu laden. Die Differenz stellt die Zeit dar, die ORACLE für das Parsen und Öffnen von Cursor braucht. Der Zuwachs der Cursorzeit beträgt ca. 80% verhältnismäßig zu dem Zuwachs der Tabelle OBJECT. Für bessere Performanz des Ladevorgangs der Worldmaps mit großer Anzahl von Segmenten muß der Algorithmus umgeändert werden, um die Anzahl der zu öffnenden Cursor zu verringern. Die Objekte werden z.B. nicht segmentsweise sondern planeweise oder in einem einzigen Cursor geladen und erst dann im Hauptspeicher in die Segmente einsortiert werden.

Da Operationen Löschen und Einfügen in den 5. und 6. Tests nicht an Worldmaps 5 und 6 im 1. Datenschema angewendet werden, wird das Diagramm der Zeitsumme nur für die 4 ersten Tests im ersten Datenschema erstellt. Es ist aber deutlich zu sehen, daß die benötigte Zeit sehr stark mit der Anzahl der Objekte wächst. Das Diagramm zeigt außerdem eine Besonderheit des 3. Datenschemas. Hier liegt der Zeitaufwand, eine Worldmap zu löschen, unter dem Zeitaufwand, sie wieder in die Datenbank einzufügen.

Abbildung 23 Zeit zum Löschen und Einfügen des gesamten Datenbestands




Das Testsystem 2 hat nur einen Prozessor, der aber leistungsfähiger als die beiden auf dem Testsystem 1. Beim Laden werden die Suchezugriffe sequentiell auf der Festplatte durchgeführt. Nur die Leistung des der Applikation zugewiesenen Prozessors und die Leistung der Festplatte bestimmen die Performanz des Vorgangs. Die Festplatten der Testsysteme, auf denen sich die Datenbank befindet, sind aber gleich. Der Ladevorgang auf dem Testsystem 2 zeigt gewisse Performanzverbesserungen im Vergleich zum Testsystem 1, was auf die Leistung des Prozessors des Testsystems 2 zurückzuführen ist. Beim Löschen und Speichern von Worldmaps sind nicht nur der DBWR-Prozeß aktiv, sondern auch der LGWR- und ARCH-Prozeß. Neue Daten müssen in den Datendateien festgehalten werden, Rollback- bzw. Redo-Informationen müssen auch auf die Festplatte geschrieben werden. Auf dem Testsystem 1 muß der Kernel unterschiedliche Aktionen durchführen, um die Lese- und Schreibzugriffe zwischen zwei Prozessoren zu koordinieren. In Abbildung 23 ist deutlich zu sehen, daß die Operationen zum Löschen und Einfügen von Worldmaps im dritten Datenschema auf dem Testsystem 1 bessere Performanz erzielt hat als auf dem Testsystem 2. Die Worldmaps, die im ersten Datenschema gespeichert sind, verursachen wegen ihrer detaillierten Zerlegung allzu viel Festplattenaktivität, was dazu führt, daß die Operationen trotz unterschiedlicher Ausstattungen der Testsysteme keine großen Performanzunterschiede aufweisen.

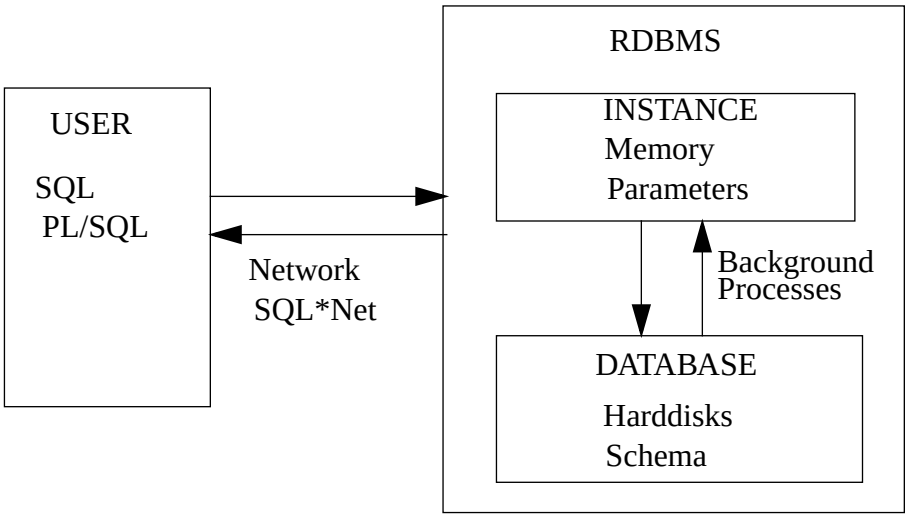
Mit dem dritten Datenschema kann man nur eine geringe Performanzverbesserung beim Laden der Worldmaps im Vergleich zum ersten Datenschema gewinnen (ca. 2 Minuten bei 80 000 Objekten). Für Modifizierungen der Worldmaps braucht man mit dem ersten Datenschema aber unakzeptierbare Zeitaufwände. Aus diesem Grund wird es als nicht nutzbar bewertet. Das dritten Datenschema eignet sich angesichts der Performanz der Ladezeiten der Worldmaps für eine spätere Implementierung auch nur, wenn weitere Optimierungen vorgenommen werden.

## 6.2 Die zweite Testreihe

In dieser Testreihe wird nicht mehr mit dem ersten Datenschema getestet. Einige zusätzliche Untersuchungen werden auch ausschließlich mit dem dritten Datenschema durchgeführt. Das Testverfahren bleibt ansonsten identisch wie in der letzten Testreihe.

### 6.2.1 Optimierungsstrategie und Beschränkung

Abbildung 24 Typische Betriebsumgebung eines Datenbanksystems




Der Benutzerprozeß kommuniziert mit dem Datenbanksystem über ein Netzwerk, um Daten zu laden oder zu manipulieren. Dabei führt die Kontrollinstanz des Datenbanksystems einerseits aus, was in den Anfragen vom Benutzerprozeß steht. Andererseits hält sie den Datenbestand während dieser Änderungsaktionen stets in einem konsistenten Zustand. Daraus ergibt sich folgende Optimierungsstrategie für die gesamte Datenbankumgebung:

- Optimierungen für ORACLE-Instanz
  - Hardware: Prozessor, Hauptspeicher
  - Software: Konfiguration mittels Startparameter
- Optimierungen für Datenbank
  - Hardware: I/O-Kontroller, Festplatten
  - Software: Datenschema, Speicherorganisation
- Optimierungen für Kommunikationsschnittstelle
  - Hardware: Kontroller, Kabel
  - Software: SQL*Net
- Optimierungen für Benutzerprozeß
  - Hardware: Prozessor, Hauptspeicher
  - Software: Datenstruktur, Algorithmus

In der Modellierung werden schon mehrere Datenschemata für die Implementierung vorgeschlagen. Die erste Testreihe hat bewiesen, daß das erste und zweite Datenschema nicht geeignet sind. Das dritte Datenschema bleibt als die letzte Möglichkeit. Ziel der Optimierung ist, es brauchbar zu machen. Über Datenstrukturen für Worldmaps wurde auch bereits diskutiert (siehe 4.3.1 ***Datenstruktur,*** Seite 53). Man kann nur die Algorithmen bzw. Implementierungsweise der Worldmap-Operationen ändern, um eine bessere Performanz für die Anwendungen zu erreichen.

Auf den konkreten Testsystemen bestehen aber verschiedene Beschränkungen, die Optimierungen bzw. Untersuchungen an den entsprechenden Stellen im Rahmen der Diplomarbeit nicht möglich machen. Sie sind als noch offene Stellen hier aufgezählt:

- Aufbau eines parallelen Servers

  Für kleine bis mittlere Worldmaps ist ein paralleler Server nicht sinnvoll. Für sehr große Worldmaps mit ca. 100 000 Objekten kann man aufwendige Operationen wie Laden, Löschen und Speichern in parallelen Subroutinen realisieren.

- Disk Striping

  Da es an Anschlußmöglichkeiten und Hardware (Kontroller, Festplatten) mangelt, wird die gesamte Datenbank zur Zeit auf einer einzigen Festplatte untergebracht. Folgende Verteilung der Daten soll aber mindestens erzielt werden:

  – Betriebssystem : +1 Festplatte
  – Redo Log-Dateien : +1 Festplatte
  – Index-Segmente : +1 Festplatte
  – Datensegmente : +1 Festplatte

### 6.2.2 Serverparametrierung und Optimierung

Die folgenden Beobachtungen werden nach der Durchführung der ersten Testreihe auf den Testsystemen gemacht.

- Der Database Cache-Puffer
  – Größe: 3 200 * 2 048 Bytes = 6 553 600 Bytes = 6 MB
  – Folgende Werte werden von der Tabelle V$SYSSTAT gelesen:

```
NAME                 SYSTEM 1      SYSTEM 2
----------------    ----------    ----------
db block gets         23366017      15889032
consistent gets     1886030754    1613003654
physical reads      1617478039    1298902441
```

  – Trefferquote: 15% auf Testsystem 1 und 21% auf Testsystem 2

  Der Puffer ist offenbar zu klein für Worldmap-Transaktionen konfiguriert. Bei den kleineren Worldmaps (1, 2) merkt man es kaum. Es ist aber anders, wenn man mit größeren Worldmaps (5, 6) zu tun hat. In diesen Fällen muß man größere Mengen an Datenblöcke in den Hauptspeicher bewegen.

- Der Redo Log-Puffer
  – Größe: 1 064 960 Bytes = 1 MB

– Folgende Werte werden von der Tabelle V$LATCH gelesen:

Auf dem Testsystem 1:

```
LATCHNAME             GETS     MISSES IMMEDIATE_GETS IMMEDIATE_MISSES
--------------- -------- ---------- -------------- ----------------
redo allocation  11001448        521              0                0
redo copy            129        122       10148959              297
```

Auf dem Testsystem 2:

```
redo allocation   7336007        151              0                0
redo copy               0          0              0                0
```

Es besteht kein Engpaß bei den Redo Latches. Es ist jedoch zu beachten, daß der Kopiervorgang von Redo Log-Einträgen in den Redo Log-Puffer meistens über das Redo Copy Latch. Der niedrige Wert in Parameter LOG_SMALL_ENTRY_MAX_SIZE (80) hat dies zur Folge.

■ Der Shared Pool-Puffer

– Größe: 15 000 000 Bytes = 14 MB

– Trefferquote im Data Dictionary Cache auf beiden Testsystemen: 99%

– Trefferquote im Library Cache auf beiden Testsystemen: 99%

Der Shared Pool-Puffer ist groß genug angelegt. Die Worldmap-Operationen greifen auf eine verhältnismäßig kleine Menge von Tabellen zu. Die SQL-Statements sind in der Regel einfach und wiederholen sich. Diese Aspekte spiegeln sich in Trefferquoten im Data Dictionary Cache- bzw. Library Cache-Puffer wider.

■ Der Sortiertbereich

– Größe: 500 000 Bytes = 500 KB

– Folgende Werte werden von der Tabelle V$SYSSTAT gelesen:

```
SORTS                  SYSTEM 1     SYSTEM 2
-------------------- ----------    ---------
sorts (memory)            78968       167125
sorts (disk)                  4           10
```

Der Bereich ist groß genug konfiguriert und ist daher auch keine Performanzhindernis für die Transaktionen.

Die meisten Parameter sind gewissermaßen schon gut mit dem Betriebssystem und mit der Datenbank abgestimmt. Der Database Cache-Puffer stellt das Hauptproblem seitens der ORACLE-Instanz für die Performanz der Worldmap-Transaktionen dar. Um das Verhalten des Servers in bezug auf den Database Cache-Puffer wird die ORACLE-Instanz für Testzwecke mit den Parametern DB_BLOCK_LRU_STATISTICS (TRUE) und DB_BLOCK_LRU_EXTENDED_STATISTICS (3500, d.h. wir haben die Absicht, den Puffer maximal um 3500 Blöcke zu erweitern) gestartet. Wenn diese zwei Parameter gesetzt werden, sammelt ORACLE Statistiken in der Tabelle SYS.X$KCBRBH, die die Performanz des vergrößerten Puffers schätzungsweise beschreiben. Man kommt zu einer überraschenden Erkenntnis: die Worldmap-Operationen können nur ganz wenig von einem größeren Datenpuffer profitieren.

Zuerst wird der Ladevorgang untersucht. Dabei werden alle Worldmaps (von 0 bis 6) zweimal geladen, um sicherzustellen, daß der Puffer im 2. Durchlauf schon etwas enthält. Die Kosten dafür werden in Anforderungen auf Datenblöcke repräsentiert:

```
NAME                            VALUE      VALUE
------------------------------ ---------- ----------
db block gets                      895       1498
consistent gets                 535399    1066871
physical reads                  164656     328244
```

Mit folgender Anfrage kann man erfahren, wieviel Treffer im Cache-Puffer gewonnen werden können, wenn der Puffer um eine bestimmte Anzahl (hier 500) von Blöcken vergrößert wird:

```
SVRMGR> SELECT 500*TRUNC(indx/500)+1 || ' to ' || 500*(TRUNC(indx/500)+1)
          Interval, SUM(count) Hits
        FROM sys.x$kcbrbh
        GROUP BY TRUNC(indx/500);
INTERVAL                          HITS       HITS
------------------------------ ---------- ----------
1 to 500                               2         16
501 to 1000                            0       3591
1001 to 1500                           0          0
1501 to 2000                           0          0
2001 to 2500                           0          0
2501 to 3000                           0          0
3001 to 3500                           0          0
------------------------------ ---------- ----------
1 to 3500                              2       3607
```

Das Löschen und Speichern von Worldmaps verursacht, daß eine große Anzahl von Datenblöcken gelesen bzw. geschrieben wird. Nachdem die Worldmaps gelöscht und wieder abgespeichert werden, haben wir folgende Bilanz von Pufferanforderungen:

```
NAME                            VALUE
------------------------------ ----------
db block gets                  1609471
consistent gets                7470320
physical reads                 5749178
```

Die Anzahl der Treffer, die wir mit 3 500 Blöcken mehr im Cache-Puffer gewinnen können, wird wie oben mit Hilfe der Tabelle X$KCBRBH auf 21 174 berechnet. Selbst dann erzielt man nur eine winzige Verringerung der physikalischen Lesezugriffe, die sowohl beim Laden als auch beim Löschen und Speichern einer Worldmap ausgeführt werden müssen. Die Trefferquote im Database Cache-Puffer wird dadurch nicht viel besser. Eine genaue Untersuchung der SQL-Befehle ist deshalb notwendig, um herauszufinden, welche von ihnen besonders viele Zugriffe auf die Festplatte verursachen, und sie einzeln zu optimieren, daß der I/O-Aufwand minimal gehalten wird (siehe 6.2.4 ***Optimierung in der Implementierung von Worldmap-Operationen,*** Seite 78).

Vor der Durchführung der 2. Testreihen werden folgende Startparameter der ORACLE-Instanz neu gesetzt:

- DB_BLOCK_BUFFERS
  - Neuer Wert: 6 400 * 2 048 Bytes = 13 107 200 Bytes = 13 MB

Der Puffer wird um 100% vergrößert, denn im normalen Betrieb der Datenbank kommen nicht nur Worldmap-Operationen vor. Außerdem müssen zusätzliche Puffertreffer gewonnen werden, während der Hauptspeicher noch nicht ausgenutzt ist.

- DB_FILE_MULTIBLOCK_READ_COUNT
  - Neuer Wert: 16

  Der Parameter zwingt den DBWR-Prozeß, mehr Datenblöcke in einem Schritt zu lesen, wenn eine sequentielle Suche auf der Festplatte erforderlich ist. Die Zugriffe im Ladevorgang von Worldmaps werden effektiver durchgeführt.

- DB_FILE_SIMULTANEOUS_WRITES
  - Neuer Wert: 8

  Die Anzahl der Schreibzugriffe, die der DBWR-Prozeß pro Datendatei in einem Schritt ausführt, wird auf den doppelten Wert gesetzt.

- DISTRIBUTED_TRANSACTIONS
  - Neuer Wert: 0

  Es liegt keine verteilte Datenbank vor, deshalb werden auch keine verteilten Transaktionen unterstützt. Keine Hintergrundsprozesse für verteilte Verarbeitung werden von der Instanz gestartet.

- LOG_ARCHIVE_START
  - Neuer Wert: FALSE

  Der Parameter schaltet den ARCH-Prozeß aus, dadurch werden Schreibzugriffe reduziert, wenn der Datenbestand sich während einer Transaktion ändert. Die Redo-Informationen werden schon in den Online Redo Log-Dateien festgehalten und eine Archivierung ist hier nicht nötig.

Wenn das System über mehrere Prozessoren (wie z.B. Testsystem 1) und mehrere Festplatten für die Organisation der Datenbank verfügt, kann die Anzahl der DBWR-Prozesse erhöht werden, um eine bessere Ausnutzung der Systemresourcen zu erreichen.

- DB_WRITERS
  - Neuer Wert: 2

(für Beschreibungen der Parameter, die auf verschiedene Weisen die Performanz der ORACLE-Instanz beeinflussen können, siehe Anhang C ***ORACLE's Startparameter,*** Seite 107)

### 6.2.3 Physikalische Lage der Datenbank und Optimierung

Die Lage der Datenbank auf der Festplatte hat sicherlich auch einen großen Einfluß auf die Transaktionen. Die physikalische Zugriffe auf die Dateien mit den Tablespaces, in denen die Daten-, Index- und Rollback-Segmenten allokiert sind, erreichen nach den Worldmap-Operationen folgende Dimensionen:

| DATAFILES | PHYRDS | PHYWRTS |
|---|---|---|
| /u01/oradata/emp1/data/HAI_WMAP.dbf | 219299452 | 378455 |
| /u02/oradata/emp1/index/HAI_INDX.dbf | 713539 | 915000 |
| /u04/oradata/emp1/roll/HAI_ROLL.dbf | 322 | 409902 |

Man kann natürlich mehr Performanz gewinnen, wenn diese Dateien separat auf verschiedenen Festplatten verteilt werden. Die Ladevorgänge verursachen große Anzahl von Lesezugriffen. Während der Tests wird beobachtet, daß die Festplatte beim Löschen bzw. Einfügen von Worldmaps äußerst belastet wird. Für eine bessere Zugriffszeit sind die Dateien auf drei verschiedenen Festplatten unterzubringen. Noch besser wäre, die Worldmap-Daten in zwei Tablespaces abzuspeichern und sie auf zwei unterschiedliche Festplatten zu verteilen. Die Tabelle OBJECT ist in der Regel sehr groß und darauf erfolgen die meisten Zugriffe.

Die Tabellen werden mit von ORACLE voreingestellten Parametern erzeugt. Diese Parameter verursachen eine sehr schlechte Lage der Daten auf der Festplatte. Hier ist die Situation der Tabellen im 6. Test mit dem dritten Datenschema:

```
SEGMENT          BYTES    EXTENTS
---------- ---------- ----------
OBJECT       14755840         17
VARVA         4382720         14
VARME         6563840         15
```

Alle Tabellen werden in einem Tablespace gespeichert. Die große Anzahl der Extents, die ORACLE für einzelne Tabellen allokiert hat, spricht für eine hohe Fragmentierung im Datentablespace. Der Parameter PCTINCREASE hat den voreingestellten Wert 50 und kann sehr gefährlich für schnell wachsende Tabellen sein, da jedes neu zu allokierendes Extent um 50% größer ist als das zuletzt allokierte Extents. Wenn das Tablespace nicht einen größeren kontinuierlichen freien Speicherblock hat, muß die Transaktion mit einem Fehler abgebrochen werden. Das passiert häufig bei den Rollback-Segmenten, da sie normalerweise nicht auf die größte Datenmenge eingestellt sind, die in einer Transaktion modifiziert werden kann. Es ist daher zwingend notwendig, genug freien Raum für das Wachstum der Rollback-Segmente in den Tablespaces zu lassen, oder mindestens ein großes Rollback-Segment für Update- bzw. Insert-intensive Transaktionen bereitzustellen. Zu Beginn jeder Transaktion kann man mit dem Befehl SET TRANSACTION USE ROLLBACK SEGMENT das gewünschte Rollback-Segment spezifizieren.

Nach dem letzten Test der ersten Testreihe werden die Größen von Tabellen und Datensätzen bzw. von Indizes mit dem Befehl ANALYZE berechnet. Die Optimierungen konzentrieren sich auf große Tabellen, da sie meistens Fragmentierung in den Index- und Daten-Tablespaces verursachen. Folgende Tabellen des 3. Datenschemas benötigen neue Speicherparameter:

Tabelle 13 Physikalische Struktur der Tabellen nach der 1. Testreihe

| Tabelle | Rows | Row Len | Chains | Total (Blocks) | Free / Blk | Rows / Blk |
|---|---|---|---|---|---|---|
| *SEGMENT* | *528* | *64* | *0* | *22* | *359* | *25* |
| *OBJECT* | *78 624* | *120* | *32* | *5 784* | *283* | *14* |
| *VARME* | *35 204* | *80* | *0* | *1 732* | *267* | *21* |
| *VARKE* | *1 893* | *50* | *0* | *60* | *273* | *32* |
| *VARVA* | *25 286* | *87* | *0* | *1 350* | *264* | *19* |

Die LONG RAW-Spalte der Tabelle OBJECT verursacht 32 chained rows. Dies ist nicht zu verhindern, da die Standardwerte PCTFREE 10 und PCTUSED 40 eine schon ziemlich freizügige Platzverwaltung den Datenblöcken darstellen. Die freie Gestaltung von graphischen Repräsentationsformen

wird dem Nutzer gewährleistet, was zur Folge hat, daß die Länge der OBJECT-Datensätze sehr variieren kann. Die Anzahl von den chained rows in der OBJECT-Tabelle stellt jedoch nur einen geringen Anteil an der Gesamtanzahl der Datensätze dar. Im Vergleich zur Tabelle OBJECT sind die Datensätze anderer Tabellen eher konstant, da die Felder durch Datentypen fester Länge definiert sind. Anbetracht der obigen Analyse können die Tabellen für die neue Testreihe mit folgenden Parametern erzeugt werden.

Tabelle 14 Parameter für die Erzeugung von Tabellen in der neuen Testreihe

| Tabelle | INITIAL | NEXT | PCTINCR | PCTFREE | PCTUSED |
|---|---|---|---|---|---|
| *SEGMENT* | *50 KB* | *10 KB* | *0* | *5* | *90* |
| *OBJECT* | *12 MB* | *1 MB* | *0* | *5* | *60* |
| *VARME* | *4 MB* | *1 MB* | *0* | *5* | *90* |
| *VARKE* | *124 KB* | *10 KB* | *0* | *5* | *90* |
| *VARVA* | *3 MB* | *1 MB* | *0* | *5* | *90* |

Das erste Extent, das beim Erzeugen einer Tabelle mit einer Größe von dem INITIAL-Wert allokiert wird, ist groß genug, um den gesamten Datenbestand der Tabelle in allen Tests aufnehmen zu können. Mit einem niedrigen Wert in PCTFREE hat man besonders Vorteil beim Anlegen der Worldmaps. Der Speicherplatz in den Datenblöcken wird optimal ausgenutzt und die Worldmap-Daten werden auf einer minimalen Anzahl von Datenblöcken untergebracht. Ähnlich wie Tabellen kann man auch einzelne Indizes der Tabellen zur Analyse ziehen und angemessene Parameter dafür aufstellen.

### 6.2.4 Optimierung in der Implementierung von Worldmap-Operationen

Um dem Anomalienverhalten im Database Cache-Puffer auf dem Grund zu gehen, werden alle Worldmap-Operationen auf eine Worldmap (3) angewendet und anschließend die Statistiken, die nur für die Worldmap-Operationen relevante SQL-Befehle betreffen, aus der Tabelle V$SQLAREA herausgenommen und analysiert.

Man muß zur Kenntnis nehmen, daß ORACLE Foreign Key-Constraints mittels zusätzlicher Abfragen realisiert, d.h. zusätzliche Cursor werden für die Prüfung auf Existenz der unter- bzw. übergeordneten Datensätze geöffnet. Sie werden bei der Änderung der entsprechenden Datensätze aktiv. Worldmap 3 enthält 9 828 Objekte und 66 Segmente (siehe Tabelle 6). Wenn ein neuer Datensatz in die Tabelle OBJECT eingefügt wird, stellt ORACLE mit folgender Abfrage sicher, daß das referenzierte Segment schon existiert:

```
STATEMENT                          EXECUTIONS BUFFER_GETS DISK_READS PARSE_CALLS
---------------------------------- ---------- ---------- ---------- ----------
select null from "PRIME"."SEGMENT"       9828      19652          0       9828
```

Wenn ein Datensatz von der Tabelle SEGMENT zu löschen ist, wird auch nachgesehen, daß keine Objekte von dem Segment mehr existieren:

```
STATEMENT                          EXECUTIONS BUFFER_GETS DISK_READS PARSE_CALLS
---------------------------------- ---------- ---------- ---------- ----------
select rowid from "PRIME"."OBJECT"         66      88242      70842          1
```

Mit dem EXPLAIN PLAN-Befehl erkennt man, daß die Suche im ersten Statement über den Primärschlüsselindex der Tabelle SEGMENT erfolgt, während das zweite Statement in einer Full Table-Su-

che resultiert. ORACLE fügt die Datenblöcke, die bei einer Full Table-Suche gelesen werden, nicht an den Anfang der LRU-Liste im Database Cache-Puffer, was zur Folge hat, daß die Blöcke trotz wiederholter Suche von der Festplatte zu laden sind. Dies erklärt die hohe Anzahl von *Disk_Reads* in der Durchführung vom zweiten Statement.

Foreign Key-Constraints halten die Integritätsbedingungen in der Datenbank aufrecht. Um aufwendige Plattenzugriffe zu verringern, ist deshalb in erster Linie Full Table-Suchen zu vermeiden und Zugriffe über Indizes zu erzwingen. Ein häufiger Fehler ist wie in diesem Fall, eine passende Spaltenreihenfolge für die Erzeugung von zusammengesetzten Indizes. Es führt dazu, daß die interne Bearbeitung vieler SQL-Statements über Full Table-Suche erfolgt. Wenn der Primärschlüsselindex mit der Reihenfolge *objid, segmid, planid, wmapid* für die Tabelle OBJECT erzeugt wird, erfolgt z.B. folgende Anfrage über einen Full Table-Zugriff:

```
SQL> EXPLAIN PLAN FOR
       SELECT * FROM object
       WHERE segmid = 1 AND planid = 1 AND wmapid = 3;
  EXPLAIN_PLAN
  ----------------------------------------------------------------------
   TABLE ACCESS FULL OBJECT
```

Der gleiche Befehl, wenn man die Reihenfolge von den Indexspalten zu *wmapid, planid, segmid, objid* umgeändert hat, resultiert in einer Index-Suche. Deshalb soll man bei der Erzeugung von zusammengesetzten Indizes immer die Spalten voranstellen, nach denen die Daten oft gesucht und sortiert werden.

```
SQL> EXPLAIN PLAN FOR
       SELECT * FROM object
       WHERE segmid = 1 AND planid = 1 AND wmapid = 3;
  EXPLAIN_PLAN
  ----------------------------------------------------------------------------
   INDEX FULL SCAN PKEY_OBJ
```

Wie aus den Beobachtungen der letzten Testreihe hervorgeht, besteht der Aufwand für das Laden einer Worldmap hauptsächlich darin, die Objekte zu laden. Der Mindestaufwand ist sozusagen die zu der Worldmap gehörenden Datensätze von der Tabelle OBJECT herauszulesen. Die hierarchische Weise beim Laden der Worldmaps hat in der 1. Testreihe gezeigt, daß viel Zeit dazu verwendet wird, Cursor zu öffnen und Objekte in kleinen Gruppen bezüglich des Segments zu laden. Daraus resultiert die Notwendigkeit eines neuen Algorithmus für die Ladefunktion, in der die Anzahl der Cursor für die OBJECT-Datensätze gering wie möglich gehalten wird. Dazu bieten sich zwei Möglichkeiten an. Die eine ist, Objekte bezüglich der zugehörigen Plane zu laden. Man reduziert die Anzahl der Cursor von der Anzahl der Segmente auf die Anzahl der Planes. Die zweite Möglichkeit ist, alle Objekte der Worldmaps in einem Schritt zu laden. Auf diese Weise kommt man am besten an den Mindestaufwand heran. Die Anzahl der Planes einer realen Worldmap steigt in der Regel nicht über 10 und die Anzahl der Segmente kann im schlimmsten Fall mehrere Hunderte betragen. In dieser Größenordnung ist der Aufwand, die Objekte in die Worldmap-Struktur im Hauptspeicher einzusortieren, nur geringfügig zu betrachten. In der Tabelle OBJECT stellt die Spalte der Worldmap-Nummern eine Spalte mit äußerst niedriger Kardinalität dar. Wenn man das Datenbanksystem in ORACLE ab Version 7.3.3 installiert, kann zusätzlich ein Bitmap-Index für diese Spalte erzeugt werden, um den Ladevorgang zu beschleunigen.

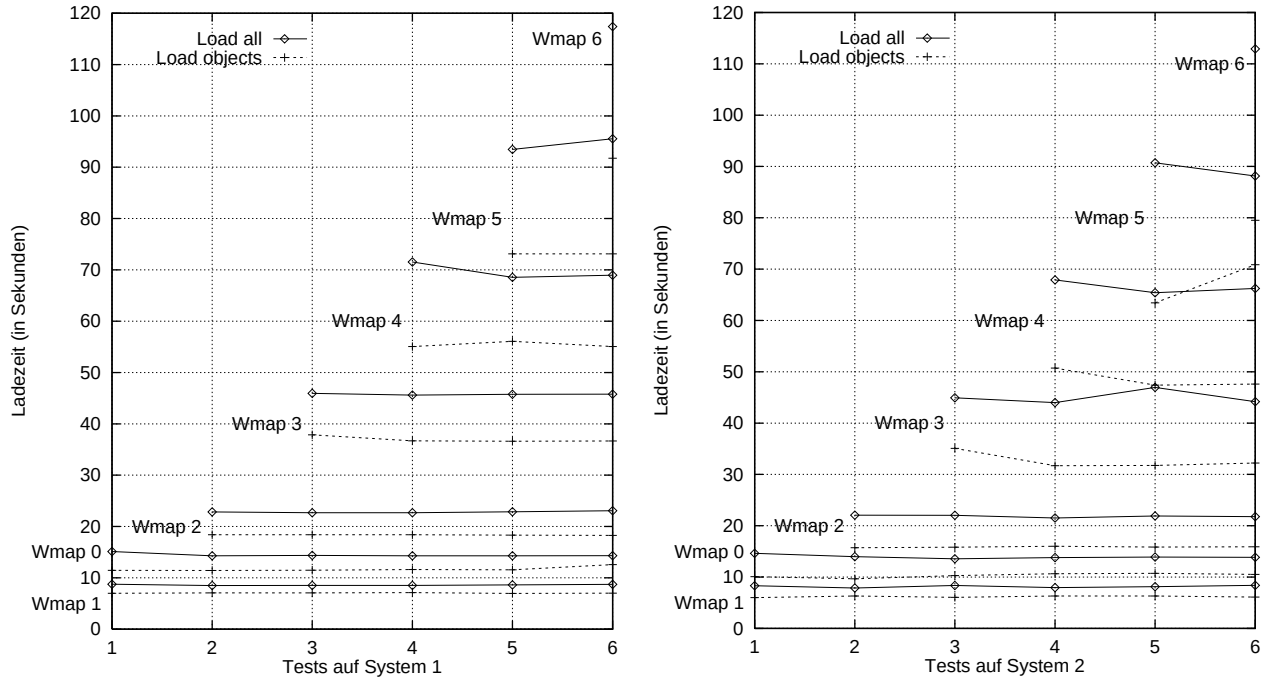
Wie die Ladefunktion läßt sich die Einfügefunktion nicht innerhalb des RDBMS implementieren. Die Funktion zum Löschen liefert dagegen keine Daten zurück und verlangt auch keine Eingaben, weshalb sie als Stored Procedure in kompilierter Form im RDBMS abgelegt werden kann. Dadurch gewinnt man die Zeit, die die ORACLE-Instanz sonst dazu verwendet, notwendige SQL-Statements zu parsen und Cursor dafür zu öffnen, denn die gesamte Routine liegt nach der Erzeugung schon in ausführbarer Form im Data Dictionary bereit. Indem man die Anzahl der Anfragen geringhält, werden die Kommunikation über Netzwerk und die dafür benötigte Zeit reduziert.

## 6.2.5 Das Verhalten der Meßzeiten

### 6.2.5.1 Der Ladevorgang

Nach den Optimierungen sehen wir eine bedeutende Verbesserung der Performanz aller Worldmap-Operationen. Vor allem bleiben die Ladezeiten der Worldmaps konstant, obwohl neue Worldmaps in die Datenbank inzwischen hinzugefügt werden.

Abbildung 25 Zeitkosten für Ladestufen im 3. Datenschema




Wie in der ersten Testreihe werden die Worldmaps generiert und die Ladefunktion wird anschließend darauf angewendet. Die Worldmaps befinden sich also in ihrer optimalen Lage auf der Festplatte. Die Ergebnisse einer Testserie (mehrmaliges Laden einer Worldmap) weichen kaum voneinander ab. Kleine Unregelmäßigkeiten treten bei den größeren Worldmaps (4, 5) auf. Die Zeiten bleiben jedoch in der gleichen Größenordnung. Aufgrund dessen, daß der Datenbestand einer neuen Worldmap sich auf den Datenbeständen existierender Worldmaps aufbaut, zeigen die Diagramme eine treppenweise Erhöhung der Ladezeiten der größeren Worldmaps. Die Tests auf dem System 2 ergeben eine Verbesserung der Performanz im Vergleich zum Testsystem 1. Sie geht auf die Leistungsfähigkeit des Prozessors des Testsystem 2 zurück.

### 6.2.5.2 Der Lösch- und Einfügevorgang

Beim Löschen von Worldmaps treten große Streuungen auf, die in der Regel mit der Größe der entsprechenden Worldmap zusammenhängen.

Abbildung 26 Zeit zum Löschen der Worldmaps im 3. Datenschema

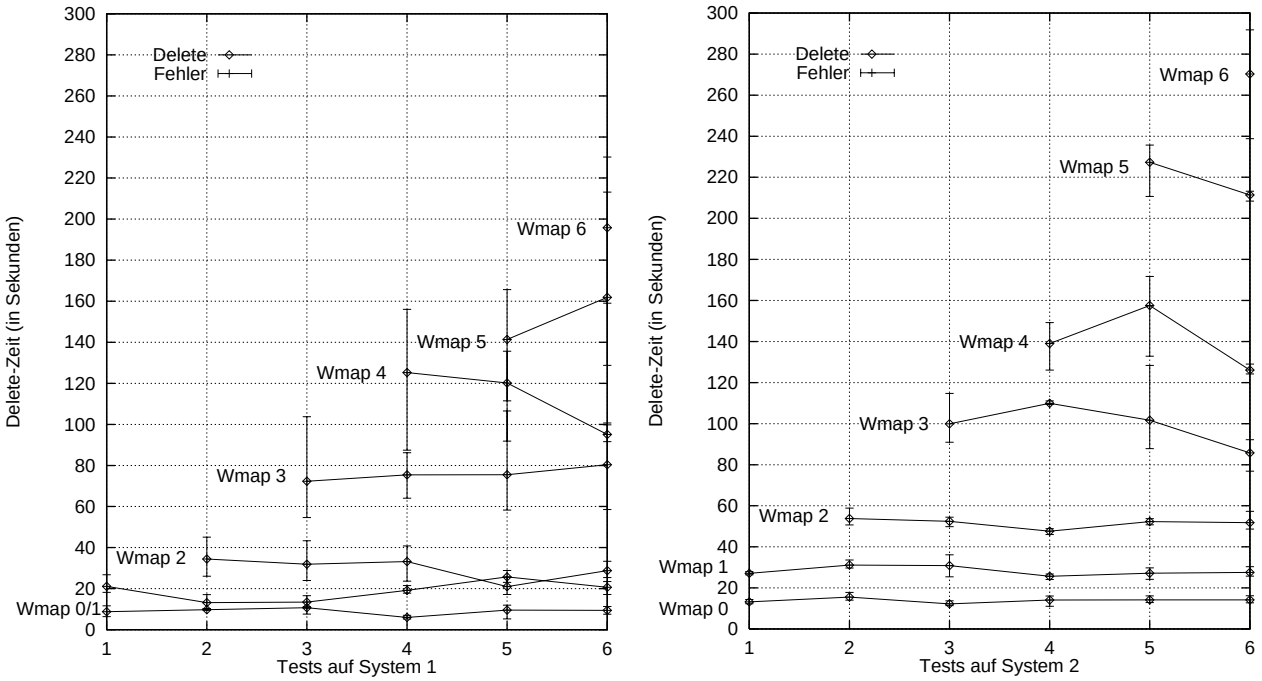
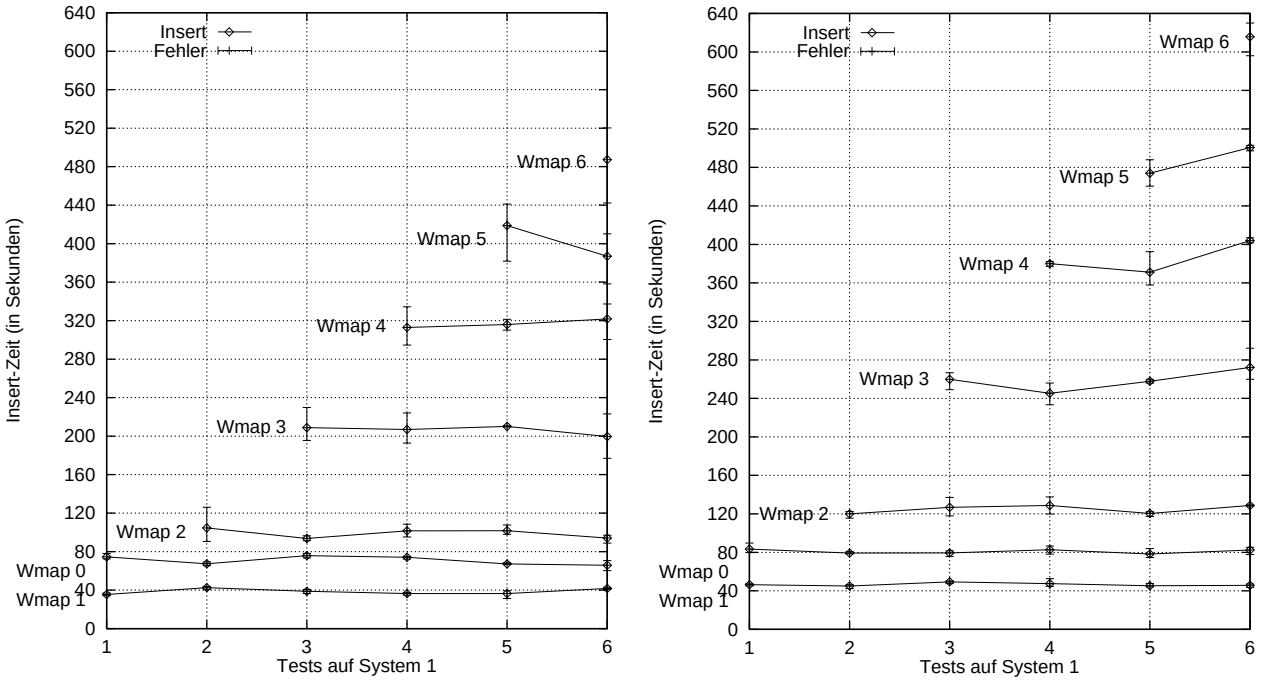



Abbildung 27 Zeit zum Einfügen der Worldmaps im 3. Datenschema



Die Ergebnisse der Einfügevorgänge zeigen dagegen schon eine deutlichere Regelmäßigkeit. Die Meßzeiten unterliegen geringeren Streuungen und bleiben auch konstant bei derselben Worldmap. Dieses Verhalten ist den Ergebnissen der Ladevorgänge ähnlich. Die Applikation kann auf dem Testsystem 1 sicherlich viel vom größeren Hauptspeicher und der Anzahl Prozessoren profitieren, was dazu führt, daß die Meßzeiten bei den I/O-aufwendigen Operationen auf dem Testsystem 1 generell besser sind als auf dem Testsystem 2, obwohl die Prozessoren des Testsystems 1 nicht so leistungsfähig wie der Prozessor des Testsystems 2.

### 6.2.6 Zusammenfassung der 2. Testreihe

Nach der Durchführung der 2. Testreihe ist das Anomalienverhalten im Database Cache-Puffer verschwunden. Die Trefferquote beträgt auf beiden Testsystemen 99%, das bedeutet, daß die Funktionen in der Regel nicht auf die benötigten Daten warten müssen:

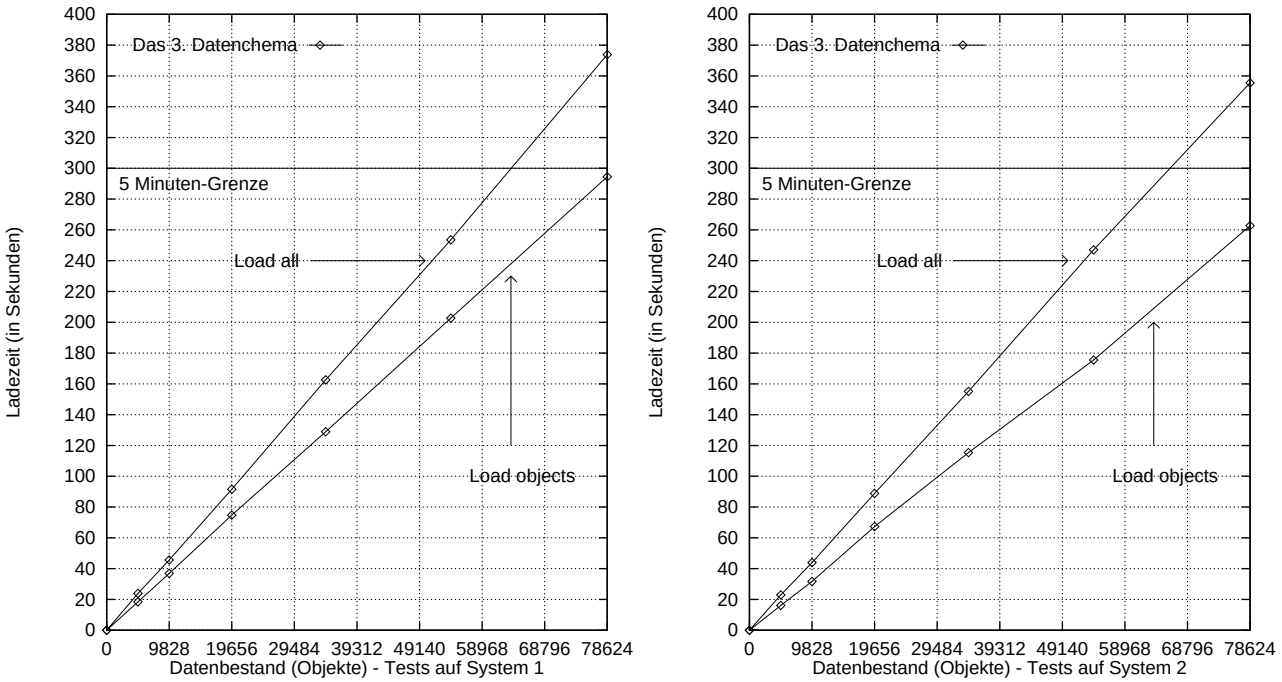
| NAME | SYSTEM 1 | SYSTEM 2 |
|---|---|---|
| db block gets | 15417540 | 9086302 |
| consistent gets | 82935584 | 17510944 |
| physical reads | 447661 | 94021 |

Die Statistiken aus der Tabelle V$SQLAREA spiegeln es auch wider. In den SQL-Befehlen, selbst denjenigen mit sehr großer Anzahl von Datenblockanforderungen, greift ORACLE ganz wenig auf die Festplatte zu:

| SQL_TEXT | EXECUTIONS | BUFFER_GETS | DISK_READS |
|---|---|---|---|
| begin wmappkg2.wmapdelete(:wmap); END | 63 | 7130073 | 14133 |
| DELETE FROM FXOBJECT WHERE WMAPID = :b1 | 63 | 3564260 | 4232 |
| insert into fxobject values (:s1:s2 ,:s | 451996 | 6117805 | 12892 |
| insert into varme values (:s1:s2 ,:s3:s | 199272 | 944426 | 1003 |
| insert into varva values (:s1:s2 ,:s3:s | 141496 | 621499 | 823 |
| select count(distinct subid) into :b0 | 91 | 1158208 | 3546 |
| select * from fxobject where wmapid=:b | 71 | 1102049 | 179 |

Die Cursor, die in der Tabelle V$SQLAREA während der ersten Testreihe registriert sind und dazu dienen, die Foreign Key-Contraints abzusichern, sind hier nicht mehr zu sehen. Es ist darauf zurückzuführen, daß die Constraints nicht mehr über Full Table-Suchen sondern über interne Index-Referenzierungen realisiert werden konnten.

Abbildung 28 Ladezeit für den gesamten Datenbestand

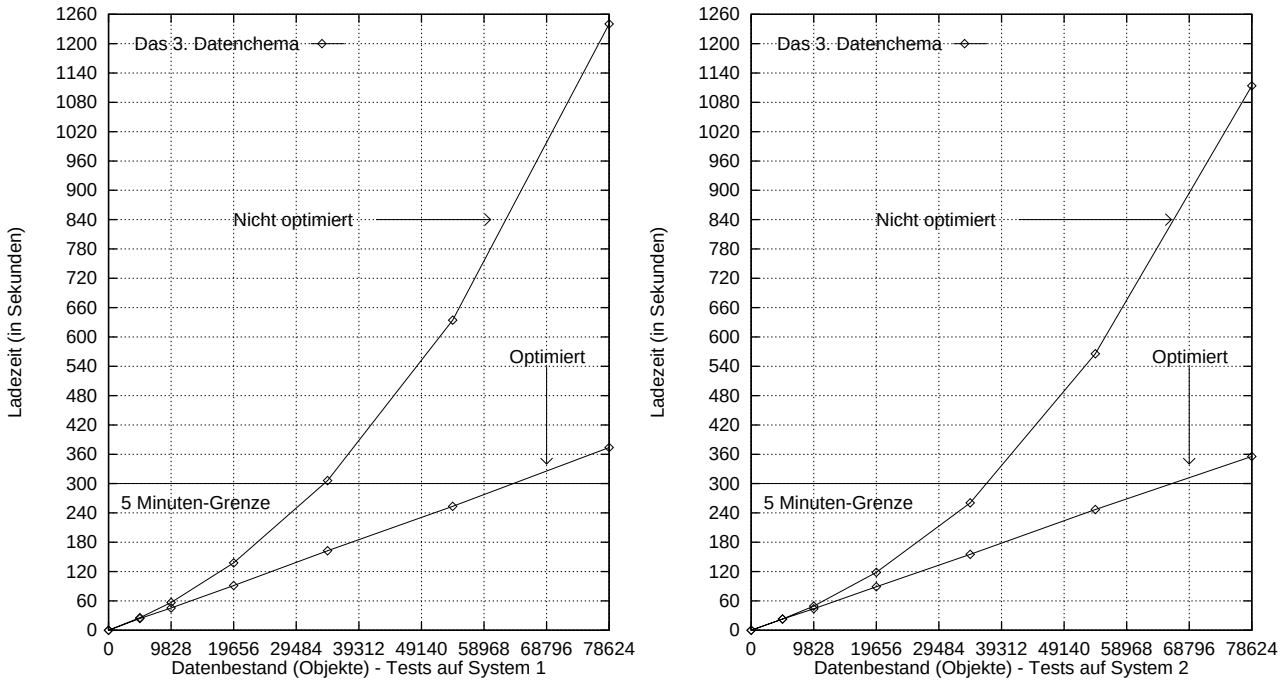



Die Summe der durchschnittlichen Ladezeiten der Worldmaps in einzelnen Tests zeigt eine lineares Wachstumsverhalten an, woraus man schließen kann, daß die notwendige Zeit für die Ladeoperation mit dem optimierten Algorithmus nicht mehr von der Anzahl der Segmente bzw. der Planes sondern im großen und ganzen nur von der Anzahl der Objekte und der Variablen abhängt. Wenn man die maximale Zeit, die für den Ladevorgang einer Worldmap als erlaubt gilt, auf 5 Minuten festsetzt, kann das Datebanksystem immer noch Worldmaps mit einer Größe von bis zu 60 000 Objekten bzw. 100 Unterstationen[1] unterstützen.

Im Vergleich zu der 1. Testreihe zeigen die Testergebnisse eine beachtliche Performanzverbesserung:

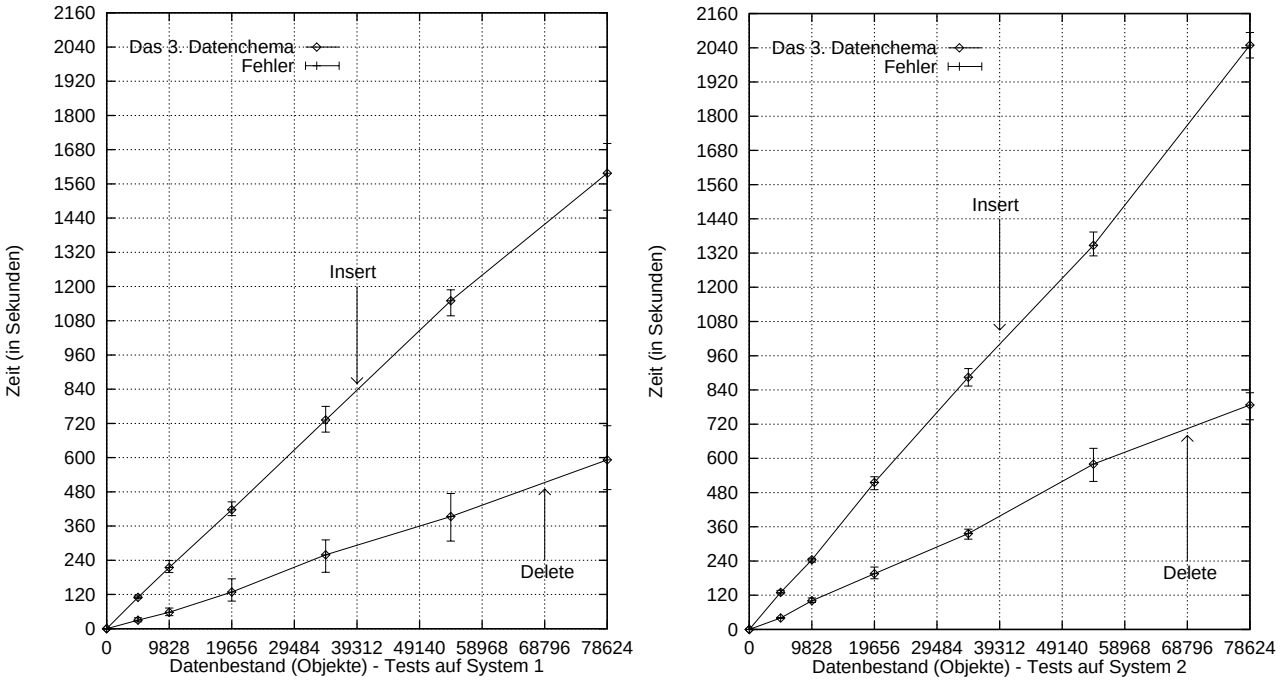
[1] Die maximale Anzahl der Variablen pro Segment wird von SINAUT Spectrum auf 500 beschränkt.

Abbildung 29 Ladezeit für den gesamten Datenbestand in beiden Testreihen




Trotz der Streuungen, denen die Meßzeiten beim Löschen und Einfügen der Testworldmaps unterliegen, weisen sie in Betrachtung auf den ganzen Datenbestand wie beim Laden ein lineares Wachstumsverhalten auf. Der Fehlerbereich der Werte vergrößert sich aber parallel dazu und kann bei einem Datenbestand von 80 000 Objekten mehrere Minuten betragen:

Abbildung 30 Zeit zum Löschen und Einfügen des gesamten Datenbestands



Mit den Optimierungen erzielen auch die Operationen zum Löschen und Einfügen von Worldmaps große Performanzverbesserungen:

Abbildung 31 Zeit zum Löschen des gesamten Datenbestands in beiden Testreihen

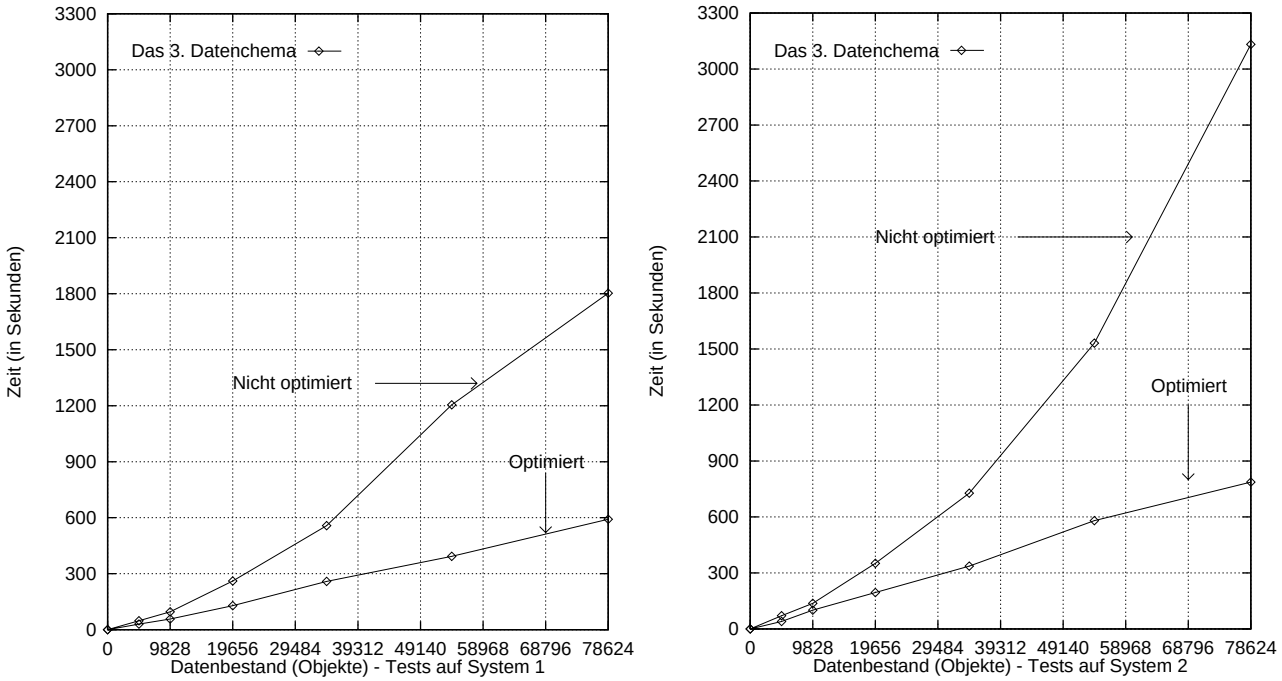
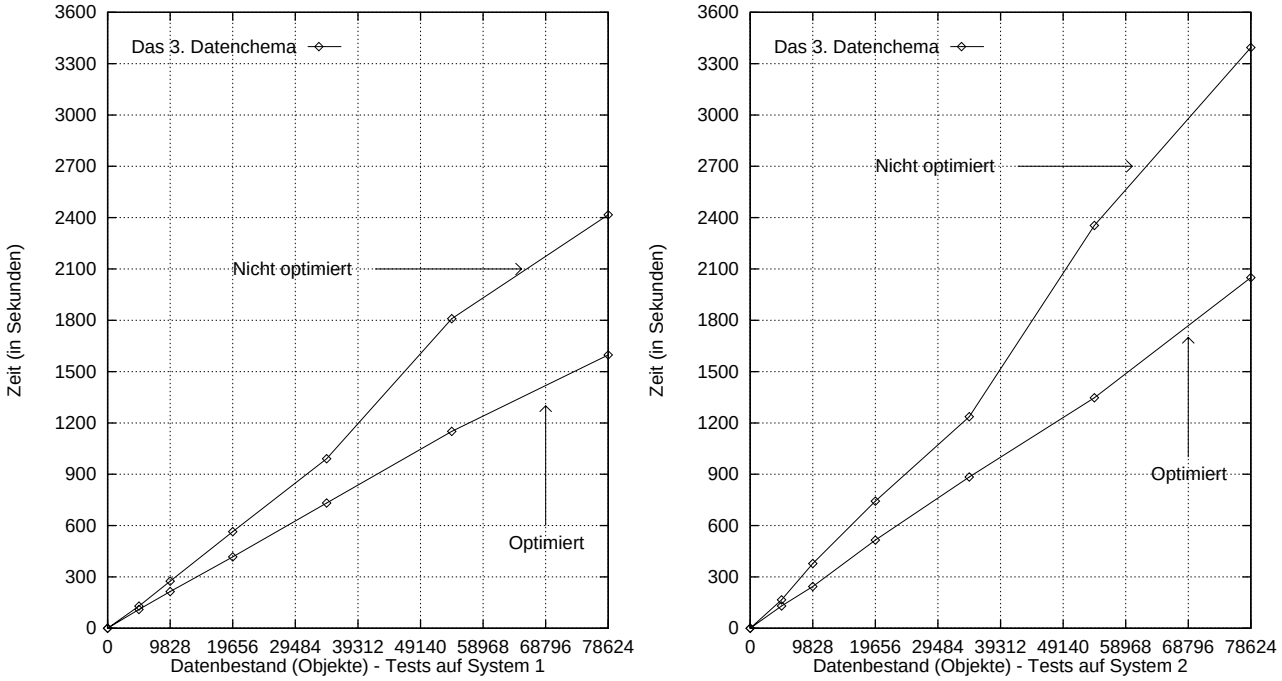



Abbildung 32 Zeit zum Einfügen des gesamten Datenbestands in beiden Testreihen



Aufgrund des regelmäßigen Verhaltens der neuen Meßzeiten kann man durchschnittliche Zeit pro Bearbeitungseinheit, welche eine Aktion mit einem OBJECT- und einem Variablenbeschreibungsdatensatz beinhaltet, aus den obigen Testergebnissen berechnen:

Tabelle 15 Durchschnittliche Zeit für Aktionen auf Objekte

| Aktionseinheit | System 1 | System 2 |
|---|---|---|
| *Laden eines Objekts* | *0,004754270* | *0,004521107* |
| *Löschen eines Objekts* | *0,007529156* | *0,010009589* |
| *Einfügen eines Objekts* | *0,020314017* | *0,026065542* |

Z.B. für eine Worldmap mit 100 000 Objekten würde die Applikation auf den Testsystemen folgende Zeiten in Anspruch nehmen:

- Laden auf System 1 = 100000 * 0,004754270 = 475,427 Sekunden
- Löschen auf System 1 = 100000 * 0,007529156 = 752,915 Sekunden
- Einfügen auf System 1 = 100000 * 0,020314017 = 2031,401 Sekunden
- Laden auf System 2 = 100000 * 0,004521107 = 452,110 Sekunden
- Löschen auf System 2 = 100000 * 0,010009589 = 1000,958 Sekunden
- Einfügen auf System 2 = 100000 * 0,026065542 = 2606,554 Sekunden

Diese Zahlen stellen natürlich nur Schätzungen dar. Es ist jedoch davon auszugehen, daß die Berechnungen für die Ladezeit wenig von den realen Werten abweichen werden, während man ein paar Minuten als Fehlertoleranzbereich zu den Lösch- bzw. Einfügeaktionen berücksichtigen muß.

# 7 Schlußwort

Peter Corrigan und Mark Gurry haben sich damit beschäftigt, mehrere Arbeitsumgebungen unter ORACLE-RDBMS zu optimieren, und daraus folgende Statistiken gezogen:

- Lokation der Performanzprobleme
  - Applikation : 60%
  - Design : 20%
  - Datenbank und Server : 17%
  - Betriebssystem : 2,5%
- Aufwand zur Behebung der Performanzprobleme
  - Design : 65%
  - Datenbank und Server : 15%
  - Applikation : 15%
  - Betriebssystem : 5%

(siehe [Corrigan94], What Causes Performance Problems, Seite 19)

Es ist interessant, mit Ergebnissen unserer Vorgehensweise zu messen bzw. zu vergleichen. In unserem Fall besteht die Motivation für die Modellierung und Implementierung einer neuen Struktur für die Worldmaps nicht in der Performanz der Applikationen, die damit im Zusammenhang stehen, sondern in verschiedenen Anforderungen, die an eine Datenbank gestellt werden: keine Redundanz, hohe Konsistenz und Integrität, Möglichkeit zur Selbstprotokollierung, kleineres Sperrgranulat. Der Datenbankentwurfsprozeß fordert auch hohen Aufwand, da die relevanten Daten genau eingegrenzt und studiert werden müssen. Dabei werden mehrere Datenschemata entworfen, die bereit für die Implementierung sind. Man spart dadurch den Aufwand, wenn ein implementiertes Datenschema den Performanzanforderungen nicht genügt und der Entwurf wieder von vorne zu beginnen ist. Wenn eine ganz neue Datenbank aufgebaut werden soll, ist es oft der Fall, daß der Server mit den voreingestellten Parametern den Applikationen nicht die gewünschte Performanz anbieten kann. Der Vorgang, eine angemessene Parametrierung für den Server zu finden, ist in der Regel von hohem Aufwand. Man muß die Parameter setzen und gezielt auf einzelne von ihnen testen, ggf. dann neu setzen. Dabei stellen die dynamischen Tabellen, in denen ORACLE während der Laufzeit verschiedene Statistiken über die Performanz der Instanz und die Nutzung der Systemresourcen ablegt, eine unverzichtbare Hilfsquelle dar.

Nach der 1. Testreihe sieht man, daß der Hauptgrund für die schlechte Performanz an der Implementierungsweise sowohl der Applikationen als auch des internen Datenschemas liegt. In dieser Testreihe bringt ein besseres Datenschema (das dritte Datenschema) nur eine kleine Performanzverbesserung für den Ladevorgang. Der Database Cache-Puffer scheint nur unbrauchbare Daten zu enthalten. Außerdem wachsen die Zeiten sehr stark im Verhältnis zum Bestand der Tabellen in der Datenbank. Eine Analyse der SQL-Befehle ergibt, daß die meisten von ihnen mittels Full-Table Scans ausgeführt werden, deren Performanz ja vom aktuellen Bestand der Tabellen abhängt. Wenn eine Applikation, in der eine konstante Menge an Daten zu bearbeiten ist, immer schlechtere Performanz bei wachsendem Datenbestand der Datenbank zeigt, sind die Anfragen auf Full-Table Scans zu untersuchen. Durch Erzeugung von neuen Indizes bzw. durch Reorganisation von zusammengesetzten Indizes, die den Zugriffsprioritäten der Indexkandidaten entspricht, kann man die Zugriffe über Index erzwingen. Die Performanz der Applikation hängt insgesamt dann nicht mehr so stark vom aktuellen Datenbestand der Datenbank ab. Der Aufwand dafür ist sicherlich auch nicht klein. Wenn das Performanzproblem auftritt und auf der Seite der Applikation vermutet wird, sind die Statistiken in der Tabelle V$SQLAREA unbedingt zur Analyse heranzuziehen. Sie hilft dabei, resourcenaufwendige Anfragen ausfindig zu machen und ihr Verhalten nach den Optimierungen zu prüfen.

Nach den Optimierungen haben die Testfunktionen große Verbesserungen gezeigt. Außerdem weisen die gemessenen Werte eine gewisse logische Regelmäßigkeit auf. Der Zeitaufwand wächst linear mit dem Datenbestand der Testworldmaps und ist selbst unabhängig vom aktuellen Bestand der Datenbank, was eine Abschätzung für eine beliebige Worldmap-Größe auf mit den Testsystemen vergleichbaren Systemen möglich macht. Die Applikationen sind in der Lage, in wenigen Minuten größere Worldmaps laden, die bis 100 000 Objekte enthalten können. Jedoch für eine bessere Antwortszeit sollen die Worldmaps aus nicht mehr als 50 000 bzw. 60 000 Objekten bestehen. Der Aspekt, unter dem wir die Testfunktionen implementiert und durchgeführt haben, ist die Untersuchung auf Performanz bei höchst-möglicher Datenbankbelastung: ganze Worldmaps werden geladen, gelöscht bzw. abgespeichert. Wenn die Arbeit mit sehr großen Worldmaps als eine alltägliche Anforderung zu unterstützen ist, muß man sich den noch offenen Optimierungsstellen zuwenden und z.B. die Datenbank auf leistungsfähigeren Rechnern installieren, parallele Server konfigurieren, mehr Festplatten dem Datenbanksystem zur Verfügung stellen. Segmente mit einer maximalen Größe von mehreren hunder-

ten Objekten, wie sie in SINAUT Spectrum vordefiniert ist, kann in wenigen Sekunden geladen werden. Deshalb ist es vielleicht eine bessere Lösung, den Ladevorgang *on demand* zu realisieren. Segmente bzw. Planes werden erst geladen, wenn sie im Dialog mit dem Benutzer verlangt werden. Man muß aber auch in Betracht ziehen, daß die Applikationen deswegen komplizierter und längere Wartezeiten dazwischen auch störend werden können. Obwohl die Tests schlechte Ergebnisse mit dem Löschen und Einfügen von Worldmaps gezeigt haben, passiert es in der Realität nicht so oft, daß komplette Worldmaps gelöscht bzw. abgespeichert werden. Wir haben den schlimmsten Fall untersucht. In der Regel betrifft die Änderung nur einen Teil der Worldmap, z.B. wenn ein neues Segment, eine neue Plane hinzugefügt oder gelöscht wird. Die Zeit dafür, ein tausend Objekte zu modifizieren, beschränkt sich dann nur auf wenige Sekunden. Unter den Bedingungen einer optimierten Datenbankumgebung ist somit die Implementierbarkeit des dritten Datenschemas gerechtfertigt.

# 8 Literaturverzeichnis

[Barker90]
Barker, R.: ***CASE*Methode - Entity Relationship Modelling***
Addison-Wesley Publishing Company, Wokingham 1990

[Batini92]
Batini, C.; Ceri, S.; Nawathe, Shamkant B.: ***Conceptual Database Design - An Entity Relationship Approach***
Benjamin/Cummings Publishing Company, Carlifornia 1992

[Beißler96]
Beißler, G.: ***Netzleittechnik - Script für Vorlesung an der Fachhochschule Coburg***
Siemens AG, EV NLT 2, Nürnberg 1996

[Corrigan94]
Corrigan, P.; Gurry, M.: ***ORACLE Performance Tuning***
O'Reilly & Associates - Inc, Sebastopol 1994

[Elmasri94]
Elmasri, R.; Nawathe, Shamkant B.: ***Fundamentals of Database Systems***
Benjamin/Cummings Publishing Company, California 1994

[Meier95]
Meier, A.: ***Relationale Datenbanken - Eine Einführung für die Praxis***
Springer-Verlag, Berlin 1995

[OraSerCon96]
ORACLE Corporation: ***ORACLE Server Concept,***
ORACLE Release 7.3, 1996

[OraDisSys96]
ORACLE Corporation: ***ORACLE Server Distributed Systems***
Volume 1: Distributed Data, ORACLE Release 7.3, 1996

[OraSerRef96]
ORACLE Corporation: ***ORACLE Server Reference Manual***
ORACLE Release 7.3, 1996

[OraSerTun96]
ORACLE Corporation: ***ORACLE Server Tuning***
ORACLE Release 7.3.3, 1996

[SieNetzleit98]
Siemens AG: ***Netzleittechnik - Funktionsbeschreibung***
Siemens AG, EV NL M, Nürnberg 1998

[SieGraEdit97]
Siemens AG: ***DBA Graphic Editors Reference***
Siemens AG, Nürnberg 1997

[Shasha92]
Shasha, D.: ***Database Tuning - A Principled Approach***
Prentice Hall, Englewood Cliffs, 1992

[Vossen94]
Vossen, G.: ***Datenmodelle, Datenbanksprachen und Datenbankmanagement-Systeme***
Addison-Wesley Publishing Company, Bonn 1994

# Anhang A Aktuelle Implementierung: Tabellen und LONG RAW-Strukturen

Tabelle 1 FXWMAPP

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | *+* | *record identifier* |
| *wmap* | *NUMBER* | *-* | *worldmap number* |
| *plane* | *NUMBER* | *-* | *plane number* |
| *data* | *LONG RAW* | *-* | *data* |

wobei der *data*-Eintrag die folgende C-Struktur darstellt:

```
  typedef struct {              /* relation fxWMapP                    */
    short       wmap;           /* 00 worldmap number              KEY */
    short       plane;          /* 02 plane number                 KEY */
    short       zoomL,zoomH;    /* 04 zoom factor range                */
    short       planeId;        /* 08 internal plane no. in WMapT      */
    short       spare;          /* 10                                  */
    } fxWMapP;                  /* 12                                  */
```

Diese Relation enthält organisatorische Daten für Planes der Worldmaps.

Tabelle 2 FXWMAPS

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | *+* | *record identifier* |
| *wmap* | *NUMBER* | *-* | *worldmap number* |
| *plane* | *NUMBER* | *-* | *plane number* |
| *segm* | *NUMBER* | *-* | *segment number* |
| *did* | *NUMBER* | *-* | *display identifier* |
| *data* | *LONG RAW* | *-* | *data* |

wobei der *data*-Eintrag die folgende C-Struktur darstellt:

```
  typedef struct {              /* relation fxWMapS                    */
    short       wmap;           /*  00 worldmap number             KEY */
    short       plane;          /*  02 plane number                KEY */
    short       segm;           /*  04 segment number              KEY */
    short       did;            /*  06 segment identifier (->WSegD,WSegS)*/
    char        comm[80];       /*  08 comment                         */
    } fxWMapS;                  /*  88                                 */
```

Diese Relation enthält organisatorische Daten für Segmente der Planes.

Tabelle 3 FXWMNAME

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | *+* | *record identifier* |

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *wmap* | *NUMBER* | - | *worldmap number* |
| *name* | *VARCHAR2(8)* | - | *worldmap name* |

Diese Relation enthält Namen der Worldmaps.

Tabelle 4 FXPLNAME

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | + | *record identifier* |
| *wmap* | *NUMBER* | - | *worldmap number* |
| *plane* | *NUMBER* | - | *plane number* |
| *name* | *VARCHAR2(8)* | - | *plane name* |

Diese Relation enthält Namen der Planes.

Tabelle 5 FXSGNAME

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | + | *record identifier* |
| *wmap* | *NUMBER* | - | *worldmap number* |
| *plane* | *NUMBER* | - | *plane number* |
| *segm* | *NUMBER* | - | *segment number* |
| *name* | *VARCHAR2(16)* | - | *segment name* |

Diese Relation enthält Namen der Segmente.

Tabelle 6 FXWMAPT

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | + | *record identifier* |
| *wmap* | *NUMBER* | - | *worldmap number* |
| *data* | *LONG RAW* | - | *data* |

wobei der *data*-Eintrag die folgende C-Struktur darstellt:

```
  typedef struct {              /* relation fxWMapT                    */
    short       wmap;           /* 00 worldmap root segment id.      KEY */
    short       version;        /* 02 worldmap version                   */
    float       Crange;         /* 04 user coordinate range              */
    long        REFdots;        /* 08 dots of reference monitor          */
    float       REFdist;        /* 12 distance view monitor of ref. mon. */
    tVRPmode    vrpmode;        /* 16 vrp - mode of worldmap             */
    char        bspare[4];      /* 20 spare                              */
    short       toolbar;        /* 24 toolbar identifier (->TOOLBAR)     */
    short       fnklink;        /* 26 fnklink identifier (->FNKLINK)     */
    short       sspare[1];      /* 28 spare                              */
    short       plNumb;         /* 30 number of planes                   */
```

```
    short       Bcol;          /* 32 background colour                  */
    short       WMopt;         /* 34 worldmap options (see below)       */
    short       zoomL,zoomH;   /* 36 zoom factor range (WMopt bit 2=1)  */
    Fpoint      sr[2];         /* 40 surrounding rectangle (zoomable)   */
    long        free;          /* 56 first free addr.  (rel.to record)  */
    } fxWMapT;                 /* 60                                    */
```

Diese Relation enthält Topologien der Worldmaps

Tabelle 7 FXWSEGS

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | *+* | *record identifier* |
| *segm* | *NUMBER* | *-* | *segm number* |
| *data* | *LONG RAW* | *-* | *data* |

wobei der *data*-Eintrag die folgende C-Struktur als Header besitzt:

```
  typedef struct {             /* relation fxWSegS                      */
    short       did;           /* 00 segment (display) identifier   KEY */
    short       res0;          /* 02 reserve                            */
    long        version;       /* 04 version number see VS_FXWSEGS      */
    short       didNext;       /* 08 next part if != 0                  */
    short       didMain;       /* 10 did of main segment                */
    short       WMdid;         /* 12 worldmap root segment (->WMapT)    */
    short       PlaneNo;       /* 14 internal plane no. in WMapT        */
    char        wmap[8];       /* 16 worldmap name                      */
    char        plane[8];      /* 24 plane name                         */
    char        segm[16];      /* 32 segment name                       */
    long        unit;          /* 48 first unit addr.      (rel.to segm)*/
    long        Lfig;          /* 52 first loc.figure addr.(rel.to segm)*/
    long        free;          /* 56 first free addr.      (rel.to segm)*/
    long        recsize;       /* 60 size of this record                */
    long        res1[4];       /* 64 reserve                            */
    } fxWSegS;                 /* 80                                    */
```

Diese Relation enthält allgemeine Daten zu Segmenten und zugleich graphische Beschreibung der statischen Objekte der einzelnen Segmente. Die Lokalfiguren bzw. die anderen Graphikobjekte werden nach dem Header nacheinander im *data*-Eintrag abgelegt.

Tabelle 8 FXWSEGD

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | *+* | *record identifier* |
| *segm* | *NUMBER* | *-* | *segm number* |
| *data* | *LONG RAW* | *-* | *data* |

wobei der *data*-Eintrag die folgende C-Struktur als Header besitzt:

```
  typedef struct {             /* relation fxWSegD                      */
    short       did;           /* 00 segment (display) identifier   KEY */
```

```
    short       res0;          /* 02 reserve                              */
    long        version;       /* 04 version number see VS_FXWSEGS        */
    short       didNext;       /* 08 next part if != 0                    */
    short       didMain;       /* 10 did of main segment                  */
    long        unit;          /* 12 first unit addr.      (rel.to segm)*/
    long        free;          /* 16 first free addr.      (rel.to segm)*/
    long        recsize;       /* 24 size of this record                  */
    long        res1[4];       /* 32 reserve                              */
    } fxWSegD;                 /* 48                                      */
```

Diese Relation enthält graphische Beschreibungen der dynamischen Objekte (außer der Lokalfiguren, sie werden im FXWSEGS abgespeichert). Sie werden nach dem Header nacheinander im *data*-Eintrag abgelegt.

Tabelle 9 DISPDES

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | *+* | *record identifier* |
| *segm* | *NUMBER* | *-* | *display identifier* |
| *data* | *LONG RAW* | *-* | *data* |

wobei der *data*-Eintrag die folgende C-Struktur als Header besitzt:

```
  typedef struct {             /* relation fxColours                    */
    short       did;           /*  00 display identifcation         KEY */
    short       segId;         /*  02 segment identifcation             */
    tdTyp       dTyp;          /*  04 display type                      */
    byte        rSeg;          /*  05 number of segment rows            */
    byte        cSeg;          /*  06 number of segment columns         */
    byte        DDvers;        /*  07 current display version           */
    short       xW;            /*  08 not used                          */
    short       yW;            /*  10 not used                          */
    tsDispVar   sDispVar;      /*  12 types of variables                */
    short       spare2;        /*  14 not used                          */
    tsDiGrVar   sDiGrVar;      /*  16 not used                          */
    ServiceId   ServiceMadi;   /*  18 not used                          */
    byte        spare3;        /*  20 not used                          */
    byte        hole;          /*  21 not used                          */
    short       sddmark;       /*  22 not used                          */
    tDirectory  Directory;     /*  24 directory information             */
    int         information[8154]; /* 52 not used                       */
    } DispDes;                 /* 32668                                 */
```

Diese Relation enthält die Displaybeschreibungen der dynamischen Objekte eines Segments. Sie werden nacheinander im *information*-Feld abgespeichert.

Tabelle 10 INPUTVAR

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | *+* | *record identifier* |

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *segm* | *NUMBER* | - | *display identifier* |
| *data* | *LONG RAW* | | *data* |

wobei der *data*-Eintrag die folgende C-Struktur besitzt:

```
  typedef struct {                  /* relation InputVar                */
    short    did;                    /* 00 segment identifier         KEY */
    short    entries;                /* 02 number of entries in array     */
    tInDes   InDes[ vipInpEnt ];     /* 04 input description array        */
    } InputVar;                      /*                                   */
```

Diese Relation enthält die Inputbescheibungen der dynamischen Objekte eines Segments. Sie werden nacheinander im *InDes*-Array abgespeichert.

Tabelle 11 FXFIGURE

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *kid* | *NUMBER* | + | *record identifier* |
| *fidx* | *NUMBER* | - | *global figure number* |
| *data* | *LONG RAW* | - | *data* |

wobei der *data*-Eintrag die folgende C-Struktur als Header besitzt:

```
  typedef struct {              /* relation fxFigure                    */
    short       fIdx;           /* 00 figure index                  KEY */
    short       version;        /* 02 fxfigure version (fxWMap_VERSION)  */
    long        free;           /* 04 next free addr. (rel.to record)    */
    Fpoint      sr[2];          /* 08 surrounding rectangle (zoomable)   */
    long        REFdots;        /* 24 dots of reference monitor          */
    float       REFdist;        /* 28 distance view monitor of ref. mon. */
    } fxFigure;                 /* 32                                    */
```

Diese Relation enthält allgemeine Daten zu den Globalfiguren und deren Aufbau, also die Graphikprimitive, aus denen eine Globalfigur besteht. Sie werden nach dem Header nacheinander im *data*-Eintrag abgespeichert.

# Anhang B Neue Implementierungen: Entitäten und Attribute

## Anhang B.1 Das erste Datenschema

Tabelle 1 WORLDMAP

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *wmapid* | *NUMBER* | *+* | *worldmap identifier* |
| *name* | *STRING(8)* | *+* | *worldmap name* |
| *crange* | *NUMBER* | *+* | *user coordinate range* |
| *refdots* | *NUMBER* | *-* | *dots of reference monitor* |
| *refdist* | *NUMBER* | *-* | *distance view of reference monitor* |
| *vrpmode* | *NUMBER* | *+* | *viewpoint mode of worldmap* |
| *toolbar* | *NUMBER* | *-* | *toolbar identifier* |
| *fnklink* | *NUMBER* | *-* | *fnklink identifier* |
| *bgcolor* | *NUMBER* | *+* | *background color* |
| *wmopt* | *NUMBER* | *-* | *worldmap option* |
| *zooml* | *NUMBER* | *+* | *zoom factor range* |
| *zoomh* | *NUMBER* | *+* | *zoom factor range* |
| *x1* | *NUMBER* | *+* | *surrounding rectangle* |
| *y1* | *NUMBER* | *+* | |
| *x2* | *NUMBER* | *+* | |
| *y2* | *NUMBER* | *+* | |

Tabelle 2 PLANE

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *planeid* | *NUMBER* | *+* | *plane identifier* |
| *wmapid* | *NUMBER* | *+* | *worldmap identifier* |
| *name* | *STRING(8)* | *+* | *plane name* |
| *zooml* | *NUMBER* | *+* | *zoom factor range* |
| *zoomh* | *NUMBER* | *+* | *zoom factor range* |

Tabelle 3 SEGMENT

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *segmid* | *NUMBER* | *+* | *segment identifier* |
| *planeid* | *NUMBER* | *+* | *plane identifier* |
| *wmapid* | *NUMBER* | *+* | *worldmap identifier* |
| *name* | *STRING (16)* | *+* | *segment name* |

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *version* | *NUMBER* | *+* | *segment version* |
| *comm* | *STRING(80)* | *-* | *comment of segment* |

Tabelle 4 OBJECT

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *objid* | *NUMBER* | *+* | *object identifier* |
| *planeid* | *NUMBER* | *+* | *plane identifier* |
| *segid* | *NUMBER* | *+* | *segment identifier* |
| *wmapid* | *NUMBER* | *+* | *worldmap identifier* |
| *reptype* | *STRING(4)* | *+* | *graphical representation type* |
| *repid* | *NUMBER* | *+* | *graphical representation identifier* |
| *vartype* | *STRING(4)* | *-* | *variable type (object is dyn)* |
| *varid* | *NUMBER* | *-* | *variable description identifier* |
| *zooml* | *NUMBER* | *+* | *zoom factor range* |
| *zoomh* | *NUMBER* | *+* | *zoom factor range* |
| *inpmode* | *NUMBER* | *-* | *input mode (object is dyn)* |
| *color* | *NUMBER* | *-* | *object color* |
| *color2* | *NUMBER* | *-* | *object color2 (objtype = LOCFIG)* |
| *x* | *NUMBER* | *-* | *location (object is dyn)* |
| *y* | *NUMBER* | *-* | |
| *x1* | *NUMBER* | *+* | *surrounding rectangle* |
| *y1* | *NUMBER* | *+* | |
| *x2* | *NUMBER* | *+* | |
| *y2* | *NUMBER* | *+* | |

Tabelle 5 GLOFIGDES

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *figid* | *NUMBER* | *+* | *global figure identifier* |
| *version* | *NUMBER* | *+* | *figure version* |
| *refdots* | *NUMBER* | *-* | *dots of reference monitor* |
| *refdist* | *NUMBER* | *-* | *distance view of reference monitor* |
| *x1* | *NUMBER* | *+* | *surrounding rectangle* |
| *y1* | *NUMBER* | *+* | |
| *x2* | *NUMBER* | *+* | |
| *y2* | *NUMBER* | *+* | |

Tabelle 6 GLOFIG

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *figid* | *NUMBER* | *+* | *global figure identifier* |
| *unitype* | *STRING* | *+* | *graphical unit type* |
| *unitid* | *NUMBER* | *+* | *graphical unit identifier* |

Tabelle 7 LOCFIG

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *figid* | *NUMBER* | *+* | *global figure identifier* |
| *unitype* | *STRING* | *+* | *graphical unit type* |
| *unitid* | *NUMBER* | *+* | *graphical unit identifier* |

Tabelle 8 PLINE

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *plineid* | *NUMBER* | *+* | *polyline identifier* |
| *width* | *NUMBER* | *-(1)* | *line width (default 1)* |
| *color* | *NUMBER* | *-(0)* | *line color (default 0)* |
| *texture* | *NUMBER* | *-(1)* | *line texture (default 1: solid)* |
| *ptnr* | *NUMBER* | *+* | *number of polyline points* |
| *pts* | *LONG RAW* | *+* | *coordinates of points* |

Tabelle 9 PGON

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *pgonid* | *NUMBER* | *+* | *polygon identifier* |
| *acolor* | *NUMBER* | *-(0)* | *area color (default 0)* |
| *astyle* | *NUMBER* | *-(0)* | *interior style (default 0: hollow)* |
| *ahatch* | *NUMBER* | *-* | *hatch style* |
| *bvis* | *NUMBER* | *-(0)* | *border visible (default 0)?* |
| *bwidth* | *NUMBER* | *-(1)* | *border line width (default 1)* |
| *bcolor* | *NUMBER* | *-(0)* | *border color (default 0)* |
| *btexture* | *NUMBER* | *-(1)* | *border texture (default 1: solid)* |
| *ptnr* | *NUMBER* | *+* | *number of polygon points* |
| *pts* | *LONG RAW* | *+* | *coordinates of points* |

Tabelle 10 ARC

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *arcid* | *NUMBER* | *+* | *arc identifier* |
| *acolor* | *NUMBER(long)* | *-(0)* | *area color (default 0)* |

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *astyle* | *NUMBER* | *-(0)* | *interior style (default 0: hollow)* |
| *ahatch* | *NUMBER* | *-* | *hatch style* |
| *bvis* | *NUMBER* | *-* | *border visible?* |
| *bwidth* | *NUMBER* | *-(1)* | *border line width (default 1)* |
| *bcolor* | *NUMBER(short)* | *-(0)* | *border color (default 0)* |
| *btexture* | *NUMBER* | *-(1)* | *border texture (default 1: solid)* |
| *radius* | *NUMBER* | *+* | *arc radius* |
| *startangle* | *NUMBER* | *-* | *arc start angle* |
| *endangle* | *NUMBER* | *-* | *arc end angle* |
| *x* | *NUMBER* | *+* | *arc center* |
| *y* | *NUMBER* | *+* | |

Tabelle 11 TEXT

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *textid* | *NUMBER* | *+* | *text identifier* |
| *color* | *NUMBER* | *-(0)* | *text color (default 0)* |
| *font* | *NUMBER* | *-(1)* | *text font (default 1)* |
| *length* | *NUMBER* | *+* | *text length* |
| *height* | *NUMBER* | *-* | *character height* |
| *upx* | *NUMBER* | *-* | *character upvector (TEXT / TEXT16)* |
| *upy* | *NUMBER* | *-* | |
| *type* | *NUMBER* | *-* | *text type (for EDITOR use only)* |
| *x* | *NUMBER* | *+* | *location* |
| *y* | *NUMBER* | *+* | |
| *text* | *STRING* | *-* | *text* |

Tabelle 12 BAR

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *barid* | *NUMBER* | *+* | *bar identifier* |
| *fgcolor* | *NUMBER* | *-* | *foreground color* |
| *bgcolor* | *NUMBER* | *-* | *background color* |
| *bgx1* | *NUMBER* | *+* | *background rectangle* |
| *bgy1* | *NUMBER* | *+* | |
| *bgx2* | *NUMBER* | *+* | |
| *bgy2* | *NUMBER* | *+* | |
| *slx1* | *NUMBER* | *+* | *slider rectangle* |
| *sly1* | *NUMBER* | *+* | |
| *slx2* | *NUMBER* | *+* | |

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *sly2* | *NUMBER* | *+* | |

Tabelle 13 CURVE

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *curveid* | *NUMBER* | *+* | *curve identifier* |
| *x* | *NUMBER* | *+* | *lower left corner* |
| *y* | *NUMBER* | *+* | |
| *width* | *NUMBER* | *+* | *width* |
| *height* | *NUMBER* | *+* | *height* |

Tabelle 14 VARMA

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *maskid* | *NUMBER* | *+* | *display mask identifier* |
| *datatype* | *NUMBER* | *+* | *data type of information* |
| *boxlen* | *NUMBER* | *+* | *length of mask box* |
| *fmno* | *NUMBER* | *-* | *format number or figure number* |
| *attrgr* | *NUMBER* | *+* | *attribute group number* |
| *attral* | *NUMBER* | *+* | *attribute alternative number* |
| *posid* | *NUMBER* | *-* | *positioning identifier* |
| *info* | *NUMBER* | *-* | *additional info for user* |
| *syntype* | *NUMBER* | *+* | *syntax type for input* |
| *inpmin* | *NUMBER* | *-* | *input range (syntype = syInt / syReal)* |
| *inpmax* | *NUMBER* | *-* | |

Tabelle 15 VARME

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *messid* | *NUMBER* | *+* | *display message identifier* |
| *promode* | *NUMBER* | *-(?)* | *processing mode (default ?)* |
| *detana* | *NUMBER* | *+* | *decision table for attribute group* |
| *detanf* | *NUMBER* | *-* | *decision table for figure group* |
| *attrgr* | *NUMBER* | *+* | *attribute group number* |
| *attral* | *NUMBER* | *+* | *attribute alternative number* |
| *b1* | *STRING* | *+* | *b1 name* |
| *b2* | *STRING* | *+* | *b2 name* |
| *b3* | *STRING* | *+* | *b3 name* |
| *elem* | *STRING* | *+* | *element name* |
| *info* | *STRING* | *+* | *info name* |
| *syntype* | *NUMBER* | *+* | *syntax type for input* |

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *inpmin* | *NUMBER* | *-* | *input range (syntype = syInt / syReal)* |
| *inpmax* | *NUMBER* | *-* | |
| *incjogr* | *NUMBER* | *-(0)* | *index of invar. NC job group (default 0)* |

Tabelle 16 VARKE

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *keyid* | *NUMBER* | *+* | *display keybox identifer* |
| *boxtype* | *NUMBER* | *+* | *type of keybox* |
| *fukey* | *NUMBER* | *-* | *number of function key* |
| *keytype* | *NUMBER* | *-* | *type of key* |
| *datatype* | *NUMBER* | *-* | *type of representation* |
| *keyno* | *NUMBER* | *-* | *number of key* |
| *pos* | *NUMBER* | *-* | *position in group of key box* |
| *nkbgr* | *NUMBER* | *-* | *number of box in group of key box* |
| *figgr* | *NUMBER* | *-* | *number of figure group* |
| *attrgr* | *NUMBER* | *-* | *attribute group number* |
| *detana* | *NUMBER* | *-* | *decision table for attribute group* |
| *detanf* | *NUMBER* | *-* | *decision table for figure group* |
| *syntype* | *NUMBER* | *+* | *syntax type for input* |
| *inpmin* | *NUMBER* | *-* | *input range (syntype = syInt / syReal)* |
| *inpmax* | *NUMBER* | *-* | |
| *ta4disel* | *NUMBER* | *-* | *type of display selection with techno. address* |
| *keymark* | *NUMBER* | *+* | *key with / without additional info* |
| *nb1* | *STRING* | *-* | *norm b1 name* |
| *nb2* | *STRING* | *-* | *norm b2 name* |
| *nb3* | *STRING* | *-* | *norm b3 name* |
| *nelem* | *STRING* | *-* | *norm element name* |
| *ninfo* | *STRING* | *-* | *norm info name* |
| *xservice* | *NUMBER* | *-* | *service identification* |
| *xinfo1* | *NUMBER* | *-* | *additional info 1* |
| *xinfo2* | *NUMBER* | *-* | *additional info 2* |
| *xinfo* | *NUMBER* | *-* | *additional info* |

Tabelle 17 VARLO

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *logid* | *NUMBER* | *+* | *display logbook identifier* |
| *reptype* | *NUMBER* | *+* | *representation type* |
| *boxlen* | *NUMBER* | *+* | *length of box* |

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *varedes* | *NUMBER* | *+* | *number of value representation description* |
| *repctrl* | *NUMBER* | *-(?)* | *representation control (default ?)* |
| *signdir* | *NUMBER* | *-* | *direction of positive sign* |
| *proctype* | *NUMBER* | *-* | *processing type* |
| *advarind* | *NUMBER* | *-* | *index if B1, B2, B3 is variable* |
| *invarind* | *NUMBER* | *-* | *index if Elem, Info is variable* |
| *valind* | *NUMBER* | *-* | *index within value series* |
| *maxind* | *NUMBER* | *-* | *max number of values in series* |
| *asaccsel* | *NUMBER* | *+* | *access to archive or schedule* |
| *preseva* | *NUMBER* | *-* | *value for preselection* |
| *aspattgr* | *NUMBER* | *-* | *pattern group number* |
| *aspattno* | *NUMBER* | *-* | *pattern number for pattern access* |
| *grfmno* | *NUMBER* | *+* | *group / format number* |
| *zerooffs* | *NUMBER* | *-(0)* | *zero offset for representation (default 0)* |
| *moditime* | *NUMBER* | *-* | *modification of start time* |
| *timeunit* | *NUMBER* | *-* | *interval between values, units* |
| *deltunit* | *NUMBER* | *-* | *delta from start time, units* |
| *modiunit* | *NUMBER* | *-* | *modification of start time, units* |
| *timeint* | *NUMBER* | *-* | *count of units for interval* |
| *deltint* | *NUMBER* | *-* | *count of units for delta* |
| *modiint* | *NUMBER* | *-* | *count of units for modification* |
| *year* | *NUMBER* | *-* | *year* |
| *month* | *NUMBER* | *-* | *month* |
| *day* | *NUMBER* | *-* | *day* |
| *hour* | *NUMBER* | *-* | *hour* |
| *min* | *NUMBER* | *-* | *minute* |
| *sec* | *NUMBER* | *-* | *second* |
| *csec* | *NUMBER* | *-* | *centisecond* |
| *b1* | *STRING* | *+* | *b1 name* |
| *b2* | *STRING* | *+* | *b2 name* |
| *b3* | *STRING* | *+* | *b3 name* |
| *elem* | *STRING* | *+* | *element name* |
| *info* | *STRING* | *+* | *info name* |
| *syntype* | *NUMBER* | *+* | *syntax type for input* |
| *inpmin* | *NUMBER* | *-* | *input range (syntype = syInt / syReal)* |
| *inpmax* | *NUMBER* | *-* | |

Tabelle 18 VARVA

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *valid* | *NUMBER* | *+* | *display value identifier* |
| *reptype* | *NUMBER* | *+* | *representation type* |
| *boxlen* | *NUMBER* | *+* | *length of box* |
| *varedes* | *NUMBER* | *+* | *number of value representation description* |
| *repctrl* | *NUMBER* | *+* | *representation control* |
| *signdir* | *NUMBER* | *-* | *direction of positive sign* |
| *advarind* | *NUMBER* | *-* | *index if B1, B2, B3 is variable* |
| *invarind* | *NUMBER* | *-* | *index if Elem, Info is variable* |
| *preseva* | *NUMBER* | *-* | *value for preselection* |
| *grfmno* | *NUMBER* | *+* | *group / format number* |
| *zerooffs* | *NUMBER* | *-* | *zero offset for representation* |
| *b1* | *STRING* | *+* | *b1 name* |
| *b2* | *STRING* | *+* | *b2 name* |
| *b3* | *STRING* | *+* | *b3 name* |
| *elem* | *STRING* | *+* | *element name* |
| *info* | *STRING* | *+* | *info name* |
| *syntype* | *NUMBER* | *+* | *syntax type* |
| *inpmin* | *NUMBER* | *-* | *input range (syntype = syInt / syReal)* |
| *inpmax* | *NUMBER* | *-* | |

Tabelle 19 VARTE

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *textid* | *NUMBER* | *+* | *display text identifier* |
| *btype* | *NUMBER* | *+* | *type of text box* |
| *boxlen* | *NUMBER* | *+* | *length of text box* |
| *attrgr* | *NUMBER* | *+* | *attribute group number* |
| *attral* | *NUMBER* | *+* | *attribute alternative number* |

Tabelle 20 VARTI

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *timeid* | *NUMBER* | *+* | *display time identifier* |
| *fmno* | *NUMBER* | *+* | *format number of comment via text list* |
| *attrgr* | *NUMBER* | *+* | *attribute group number* |
| *attral* | *NUMBER* | *+* | *attribute alternative number* |
| *timeunit* | *NUMBER* | *-(4)* | *time unit (default 4: second)* |
| *timeint* | *NUMBER* | *-(?)* | *time interval per unit (default ?)* |

Tabelle 21 VARCU

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *curvid* | *NUMBER* | *+* | *display curve identifier* |
| *valsrc* | *NUMBER* | *+* | *value source (storage area)* |
| *zparts* | *NUMBER* | *-(0)* | *current time part to be zeroed (default 0)* |
| *year* | *NUMBER* | *-* | *year* |
| *month* | *NUMBER* | *-* | *month* |
| *day* | *NUMBER* | *-* | *day* |
| *hour* | *NUMBER* | *-* | *hour* |
| *min* | *NUMBER* | *-* | *minute* |
| *sec* | *NUMBER* | *-* | *second* |
| *timeper* | *NUMBER* | *-(60)* | *time periode berween 2 pts (default 60 sec.)* |
| *ptshift* | *NUMBER* | *-(0)* | *number of points to shift (default 0)* |
| *xb1* | *STRING* | *-* | *b1 name* |
| *xb2* | *STRING* | *-* | *b2 name* |
| *xb3* | *STRING* | *-* | *b3 name* |
| *xelem* | *STRING* | *-* | *element name* |
| *xinfo* | *STRING* | *-* | *info name* |
| *yb1* | *STRING* | *+* | *b1 name* |
| *yb2* | *STRING* | *+* | *b2 name* |
| *yb3* | *STRING* | *+* | *b3 name* |
| *yelem* | *STRING* | *+* | *element name* |
| *yinfo* | *STRING* | *+* | *info name* |
| *xdir* | *CHAR* | *-('N')* | *x axis direction (default 'N': north)* |
| *ydir* | *CHAR* | *-('E')* | *y axis direction (default 'E': east)* |
| *xval* | *CHAR* | *-('T')* | *x axis value (default 'T': time)* |
| *custyle* | *CHAR* | *-('L')* | *curve style (default 'L': slope)* |
| *xmin* | *NUMBER* | *-* | *x axis range* |
| *xmax* | *NUMBER* | *-* | |
| *ymin* | *NUMBER* | *+* | *y axis range* |
| *ymax* | *NUMBER* | *+* | |
| *yref* | *NUMBER* | *-(0)* | *y reference value (default 0)* |
| *acolor* | *NUMBER* | *-(0)* | *area color (default 0)* |
| *astyle* | *NUMBER* | *-(0)* | *area style (default 0: hollow)* |
| *ahatch* | *NUMBER* | *-* | *area hatch style* |
| *bvis* | *NUMBER* | *-(0)* | *border visible (default 0)?* |
| *bcolor* | *NUMBER* | *-(0)* | *border color (default 0)* |
| *bwidth* | *NUMBER* | *-(1)* | *border width (default 1)* |
| *bstyle* | *NUMBER* | *-(1)* | *border line style (default 1: solid)* |

Tabelle 22 VARVIDCU

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *curvid* | *NUMBER* | *+* | *display vid curve identifer* |
| *xdir* | *CHAR* | *-(‘N’)* | *x axis direction (default ‘N’: north)* |
| *ydir* | *CHAR* | *-(‘E’)* | *y axis direction (default ‘E’: east)* |
| *custyle* | *CHAR* | *-(‘L’)* | *curve style (default ‘L’: slope)* |
| *ymin* | *NUMBER* | *-* | *y axis range* |
| *ymax* | *NUMBER* | *-* | |
| *yref* | *NUMBER* | *-(0)* | *y reference value (default 0)* |
| *acolor* | *NUMBER* | *-(0)* | *area color (default 0)* |
| *astyle* | *NUMBER* | *-(0)* | *area style (default 0: hollow)* |
| *ahatch* | *NUMBER* | *-* | *area hatch style* |
| *bvis* | *NUMBER* | *-(0)* | *border visible (default 0)?* |
| *bcolor* | *NUMBER* | *-(0)* | *border color (default 0)* |
| *bwidth* | *NUMBER* | *-(1)* | *border width (default 1)* |
| *bstyle* | *NUMBER* | *-(1)* | *border line style (default 1: solid)* |

Tabelle 23 VARVCLO

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *curvid* | *NUMBER* | *+* | *display vid-curve identifier* |
| *logid* | *NUMBER* | *+* | *logbook variable identifier* |

## Anhang B.2 Das zweite Datenschema

Nach der zweiten Optimierung wird jede Globalfigur bzw. Lokalfigur zusammengefaßt und in einem einzigen Datensatz abgespeichert. Die Relationen GLOFIGDES und GLOFIG werden zu einer neuen Relation GLOFIG vereint, die sowohl die algemeine Beschreibung für die einzelnen Globalfiguren als auch deren Graphikprimitive enthält. Dabei nehmen wir die Änderungen zurück, die wir an der Speicherstruktur der Globalfiguren in der Modellierung gemacht haben (siehe Anhang A ***Aktuelle Implementierung: Tabellen und LONG RAW-Strukturen,*** Seite 91, Tabelle FXFIGURE).

Tabelle 24 GLOFIG / LOCFIG

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *figid* | *NUMBER* | *+* | *global or local figure number* |
| *data* | *LONG RAW* | *+* | *data* |

Der *data*-Eintrag der Relation GLOFIG besitzt die folgende C-Struktur als Header besitzt:

```
typedef struct {              /* relation fxFigure                      */
  short       fIdx;           /* 00 figure index                   KEY */
  short       version;        /* 02 fxfigure version (fxWMap_VERSION)  */
  long        free;           /* 04 next free addr. (rel.to record)    */
```

```
Fpoint      sr[2];          /* 08 surrounding rectangle (zoomable)   */
long        REFdots;        /* 24 dots of reference monitor          */
float       REFdist;        /* 28 distance view monitor of ref. mon. */
} fxFigure;                 /* 32                                    */
```

Die Graphikprimitive, die zu einer Globalfigur gehören, werden nach dem obigen Header hintereinander gespeichert. Da für die Lokalfiguren keine allgemeine Beschreibungen nötig sind, enthält der *data*-Eintrag der neuen Relation LOCFIG keinen Header, sondern nur die Graphikprimitive, die zu der Lokalfigur gehören.

## Anhang B.3 Das dritte Datenschema

Nach der dritten Optimierung haben wir die Relationen der Graphikbeschreibungen ganz beseitigt. Dabei wird die OBJECT-Relation um eine LONG RAW-Spalte erweitert, um die Graphikbeschreibungen zu enthalten, die zu einem Objekt gehören.

Tabelle 25 OBJECT

| Column | Datatype | Not Null | Description |
|---|---|---|---|
| *objid* | *NUMBER* | *+* | *object identifier* |
| *planeid* | *NUMBER* | *+* | *plane identifier* |
| *segid* | *NUMBER* | *+* | *segment identifier* |
| *wmapid* | *NUMBER* | *+* | *worldmap identifier* |
| *reptype* | *STRING(4)* | *+* | *graphical representation type* |
| *subid* | *NUMBER* | *-* | *global figure id. when objtype = OBJ_GFIG* |
| *data* | *LONG RAW* | *-* | *graphical rep. when objtype != OBJ_GFIG* |
| *vartype* | *STRING(4)* | *-* | *variable type (object is dyn)* |
| *varid* | *NUMBER* | *-* | *variable description identifier* |
| *zooml* | *NUMBER* | *+* | *zoom factor range* |
| *zoomh* | *NUMBER* | *+* | *zoom factor range* |
| *inpmode* | *NUMBER* | *-* | *input mode (dyn)* |
| *color* | *NUMBER* | *-* | *object color* |
| *color2* | *NUMBER* | *-* | *object color2 (objtype = LOCFIG)* |
| *x* | *NUMBER* | *-* | *location (dyn)* |
| *y* | *NUMBER* | *-* | |
| *x1* | *NUMBER* | *+* | *surrounding rectangle* |
| *y1* | *NUMBER* | *+* | |
| *x2* | *NUMBER* | *+* | |
| *y2* | *NUMBER* | *+* | |

Die Globalfiguren stellen jedoch eine Ausnahme dar, weil sie mehrfach verwendbar sind. Deshalb werden sie weiterhin außerhalb der OBJECT-Relation in einer eigenen Relation aufgefaßt. Im Vergleich zum zweiten Datenschema entfallen folgende Relationen: LOCFIG, PLINE, PGON, TEXT, ARC, CURVE.

# Anhang C ORACLE's Startparameter

Dieser Anhang gibt eine Übersicht ausschließlich über die Parameter wieder, die dem Anwender die Möglichkeit anbieten, Performanz der Zugriffe auf Hauptspeicher sowie auf Festplatte in ORACLE zu optimieren. Diese Liste der Parameter wird spezifisch für ORACLE Version 7 erstellt. Dies bedeutet, daß manche von ihnen für ORACLE Version 6 irrelevant sind. (Für eine komplette Beschreibung aller INIT.ORA-Parameter siehe [OraSerRef96])

- CHECKPOINT_PROCESS (TRUE / FALSE)

  Voreingestellt: FALSE

  Der Parameter aktiviert (TRUE) oder deaktiviert (FALSE) den ab der Version 7 zur Verfügung stehenden Hintergrundsprozeß CKPT. Wenn der CKPT-Prozeß aktiviert wird, übernimmt er die Funktion, die Datendateien und Kontrolldateien zu den Checkpoints zu aktualisieren, für die der LGWR-Prozeß normalerweise zuständig ist. Dadurch kann der LGWR-Prozeß sich auf seine eigentliche Arbeit konzentrieren, nämlich die Redo Log-Einträge vom Redo Log-Puffer im SGA in die Redo Log-Dateien zu schreiben.

- CURSOR_SPACE_FOR_TIME (TRUE / FALSE)

  Voreingestellt: FALSE

  Wenn der Parameter mit TRUE gesetzt wird, allokiert ORACLE mehr Speicher für Cursor, um Zeit zu sparen. Dies betrifft sowohl den gemeinsam genutzten SQL-Bereich (Shared SQL Area) als auch den privaten SQL-Bereich des Clientprozesses. Der Shared bzw. der private SQL-Bereich wird nicht vom Alterungsmechanismus betroffen bzw. deallokiert, so lange noch ein geöffneter Cursor ihn referenziert. Die Cursor müssen nicht nochmals geparst werden, denn jeder zugehörige Cursor-Bereich wird bis zum Ende seiner Ausführung im Shared Pool-Puffer beibehalten. Dabei muß man beachten, daß der Shared Pool-Puffer groß genug allokiert sein muß, um alle Cursor gleichzeitig enthalten zu können.

- DB_BLOCK_BUFFERS (4 - systemabhängig)

  Voreingestellt: 32

  Der Parameter schreibt vor, wieviele Datenblöcke (ORACLE-Blöcke) im SGA temporär gespeichert werden. Dieser Parameter ist der entscheidende Faktor für die Größe des SGA sowie für die Performanz einer ORALCE-Instanz. Der Database Cache-Puffer können Datenblöcke von Tabellen, Indizes, Clusters sowie von Rollback-Segmenten enthalten. Je größer der Puffer ist, desto größer ist die Wahrscheinlichkeit, daß ein gesuchter Datenblock schon im Speicher vorhanden ist. Dadurch kann man unnötige I/O-Operationen vermeiden.

- DB_BLOCK_CHECKPOINT_BATCH (0 - DB_BLOCK_WRITE_BATCH)

  Voreingestellt: 8

  Der Parameter gibt an, wieviele Datenblöcke der DBWR-Prozeß pro eine Chekpoint-Aktion in die Dateien schreiben kann. Man kann diesen Wert erhöhen, um die Datenbank zu den Checkpoints schneller zu aktualisieren. Wenn man einen niedrigeren Wert für diesen Parameter setzt, kann man die Intanz davon hindern, sich nur mit Checkpoint-Schreibaktivitäten zu beschäftigen, und andere modifizierte Datenblöcke können dann auch auf die Festplatte geschrieben. In allge-

meinen muß der Parameter mit einem Wert gesetzt werden, der dem DBWR erlaubt, seine Schreibaktivitäten zu einem Checkpoint zu beenden, bevor ein neuer Checkpoint stattfindet.

- DB_BLOCK_WRITE_BATCH (1 -128)

  Voreingestellt: 8

  Der Parameter gibt die Anzahl der Datenblöcke an, die der DBWR-Prozeß an das Betriebssystem in einer Schreibaktion übergibt. Wenn das Betriebssystem parallele Plattenzugriffe und Beschreiben der benachbarten Blöcke in einem Plattenzugriff erlaubt, kann man einen höheren Wert für den Parameter setzen, um eine bessere Performanz für Schreibaktivitäten des DBWR-Prozesses zu bekommen.

- DB_BLOCK_SIZE (systemabhängig)

  Voreingestellt: systemabhängig, oft 2048

  Der Parameter repräsentiert die Größe der ORACLE-Datenblöcke in Bytes. Diesen Parameter soll man wie voreingestellt lassen. Wenn die Datenbank aber speziell für größere bzw. kleinere Datensätze zu optimieren ist, kann mann die größe der Datenblöcke entsprechend erhöhen oder verkleinern für bessere Performanz in I/O-Aktivitäten. In einer Multi Instance-Konfiguration des Datenbankservers muß der Wert in allen Instanzen gleich sein.

- DB_FILE_MULTIBLOCK_READ_COUNT (systemabhängig)

  Voreingestellt: systemabhängig, oft 1

  Dieser Parameter gibt die maximale Anzahl der Datenblöcke an, die bei einer sequentiellen Suche gleichzeitig in einer I/O-Operation geladen werden. Für lang angelegte Transaktionen, in denen viele Full Table-Scans durchgeführt werden, kann man den Wert für diesen Parameter erhöhen, um eine bessere I/O-Transferrate zu erreichen.

- DB_FILE_SIMULTANEOUS_WRITES (1 - 24)

  Voreingestellt: 4

  Der Parameter gibt die Anzahl der Schreiboperationen an, die der DBWR-Prozeß in einem Batch für jede Datendatei ausführen kann. Wenn das Betriebssystem nur eine Schreiboperation pro Festplatte erlaubt und die benachbarten Blöcke nicht in einer Operation kombiniert beschreiben kann, dann soll der Parameter zu 1 gesetzt werden.

- DB_WRITERS (1 - ...)

  Voreingestellt: 1

  Der Parameter gibt die Anzahl der DBWR-Prozesse einer ORACLE-Instanz an. Man kann viel Performanz gewinnen, wenn die Datendateien auf verschiedenen Festplatten plaziert werden und die Anzahl der DBWR-Prozesse entsprechend gesetzt wird. ORACLE steuert die DBWR-Prozesse so, daß sie parallel die Festplatten beschreiben können. Als Richtlinie gilt, daß die Anzahl der DBWR-Prozesse nicht das Dreifache der Anzahl der Festplatten überschreiten darf, auf denen die Datendateien sich befinden (siehe [Corrigan94], Seite 329). Es besteht jedoch Kollisionsgefahr, wenn die Anzahl der Festplatten geringer ist als die Anzahl der DBWR-Prozesse.

- DISCRETE_TRANSACTIONS_ENABLED (TRUE / FALSE)

  Voreingestellt: FALSE

Wenn diskrete Transaktionen unterstützt werden sollen, setzt man den Parameter zu TRUE. Es gibt eine Einschränkung, wieviele Transaktionen im Diskret-Modus vorkommen können.

- LOG_ARCHIVE_BUFFERS (systemabhängig)

  Voreingestellt: systemabhängig

  Der Parameter gibt die Anzahl der für Archivierung zu allokierende Puffer an. Wenn der Server im ARCHIVELOG-Modus läuft, kopiert der ARCH-Hintergrundsprozeß die Redo-Informationen von den Redo Log-Dateien auf ein anderes Speichermedium. Je mehr Puffer für den ARCH-Prozeß allokiert werden, desto schneller kann er seine Schreiboperationen durchführen. Ein Wert von 3 ist angemessen, wenn man Archivierung auf Festplatten oder schnelle Tape-Laufwerke anordnet.

- LOG_ARCHIVE_BUFFER_SIZE (systemabhängig)

  Voreingestellte: systemabhängig

  Der Parameter gibt die Größe für jeden Archive-Puffer an. Hier gilt wieder, wenn der Puffer zu klein ist, erhöht sich die Schreibaktivität während der Laufzeit des ARCH-Prozesses. Also sollte man den Parameter auf den größt-möglichen Wert setzen, um die beste Performanz vom ARCH-Prozeß zu erreichen.

- LOG_ARCHIVE_START (TRUE / FALSE)

  Voreingestellt: FALSE

  Der Parameter indiziert, ob die Archivierung von Redo-Informationen automatisch von der ORACLE-Instanz (ARCHIVELOG-Modus) ausgeführt werden soll. Wenn der Parameter zu FALSE gesetzt wird, kann man trotzdem auch die Archivierung manuell anstoßen.

- LOG_BUFFER (systemabhängig)

  Voreingestellt: 4 * Block Size

  Der Parameter gibt die Anzahl von Bytes an, die für den Redo Log-Puffer im SGA allokiert werden sollen. Generell kann man mit einem größeren Redo Log-Puffer I/O-Operationen auf Redo Log-Dateien vermeiden, besonders wenn der Datenbankserver häufig mit update-intensiven Applikationen belastet wird. Ist der Puffer zu groß eingestellt, kann es aber lange dauern, bis der LGWR-Prozeß die Redo-Einträge auf die Festplatte geschrieben hat, um Platz im Puffer zu schaffen.

- LOG_CHECKPOINT_INTERVAL (systemabhängig)

  Voreingestellt: 2 - unbeschränkt

  Der Parameter gibt die Anzahl der Blöcke der Redo Log-Dateien (also Betriebssystemblöcke) an, die neu gefüllt sind, um einen Checkpoint zu veranlassen. Diesen Wert unbetrachtet, passiert ein Checkpoint immer, wenn ORACLE sich von einer Redo Log-Datei zu einer anderen umstellt. Wenn man niedrigen Wert für diesen Parameter spezifiziert, werden Checkpoints häufiger verursacht. Die Datenbank kann zwar schneller bei einem Fehlerfall restauriert werden, da die Daten stets in kleinen Portionen schon davor validiert werden. Für eine bessere Performanz der Transaktionen soll man aber einen höheren Wert setzen. Dadurch wird der I/O-Aufwand minimiert, die Checkpoint-Bedingungen zu prüfen bzw. Daten auf die Festplatte zu Checkpoints zu schreiben.

- LOG_ENTRY_PREBUILD_THRESHOLD (0 - unbeschränkt)

Voreingestellt: 0

Der Parameter gibt die Größe der Redo-Informationen in Bytes an, die zuerst zusammengefaßt werden, bevor sie in den Log-Puffer kopiert werden. Ein Wert, der ungleich 0 ist, zwingt den Clientprozeß, seine Redo-Informationen zusammenzufassen, bevor er von ORACLE das Redo Copy Latch verlangt. Dadurch kann der Engpaß mit dem Redo Copy Latch aufgelöst werden. Diese Parameter ist nur sinnvoll für System mit mehreren Prozessoren. Bei System nur mit einem Prozessor soll man den Parameter wie voreingestellt lassen.

- LOG_SIMULTANEOUS_COPIES (0 - unbeschränkt)

  Voreingestellt: Anzahl der CPUs

  Der Parameter gibt die Anzahl der Redo Copy Latches an, die gleichzeitig in den Redo Log-Puffer schreiben können. Wenn ein Engpaß mit dem Redo Copy Latches im System vorliegt, kann man trotzdem den Parameter doppelt soviel wie die Anzahl der CPUs setzen.

- LOG_SMALL_ENTRY_MAX_SIZE (systemabhängig)

  Voreingestellt: systemabhängig

  Der Parameter gibt die maximale Größe der Redo-Eintragsportion an, die in den Redo Log-Puffer kopiert werden kann, wenn der Clientprozeß das Redo Allocation Latch bekommen hat, ohne das Redo Copy Latch noch anfordern zu müssen. Dieser Parameter hat nur Wirkung, wenn LOG_SIMULTANEOUS_COPIES größer als 0 ist. Alle Redo-Informationseinheiten werden ohne das Redo Copy Latch in den Redo Log-Puffer geschrieben, wenn deren Größe den Wert dieses Parameters nicht überschritten hat. Ansonsten muß der Clientprozeß das erhaltene Redo Allocation Latch freigeben und das Redo Copy Latch anfordern, um größere Redo-Informationseinheiten in den Puffer zu kopieren.

- OPEN_CURSORS (1 - 255)

  Voreingestellt: 50

  Der Parameter gibt die maximale Anzahl der Cursor (Context Areas) an, die in einer Session geöffnet werden. Wenn der Fehler "MAX OPEN CURSORS EXCEEDED" erscheint bzw. die Antwortszeit ungewöhnlich lange ist, kann man den Parameter zu seinem Maximum 255 setzen.

- OPTIMIZER_MODE (CHOOSE / RULE / FIRST_ROWS / ALL_ROWS)

  Voreingestellt: CHOOSE

  Mit diesem Parameter kann man das Verhalten des SQL-Optimierer beeinflussen. Defaultmäßig arbeitet der Optimierer nach den Kostenbeobachtungen, die jedoch regelmäßige Analyse des Datenbestands fordert. Für die SQL-Statements, die schon sorgfältig (z.B. für ORACLE Version 6) optimiert werden, setzt man den Parameter zu RULE. Die Alternativen FIRST_ROWS und ALL_ROWS basiert auch auf der kostenorientierten Verhaltensweise des Optimierers.

- SHARED_POOL_SIZE (300 KB - systemabhängig)

  Voreingestellt: 3.5 MB

  Der Parameter gibt die Größe des Shared Pool-Puffers an, der im SGA beim Start der Instanz allokiert wird. Im Shared Pool-Puffer werden die Informationen vom Data Distionary bzw. SQL-Statements und PL/SQL-Blöcke in ausführungsbereiter Form zeitlich abgespeichert. Eine gute Trefferquote im Shared Pool-Puffer ist eine Voraussetzung für die hohe Performanz der

Transaktionen, deshalb soll man den Parameter so groß wie möglich setzen. Eine genaue Kalkulation zwischen der Größe des SGAs und der Größe des verbleibenden Hauptspeichers für Betriebssystem ist dabei erforderlich, um nicht die Swapping- und Paging-Gefahr in Kauf zu nehmen.

- SORT_AREA_SIZE (systemabhängig)

  Voreingestellt: systemabhängig

  Der Parameter gibt die maximale Größe des Speicherbereichs an, der von einem Clientprozeß für seine Sortierungen beansprucht werden kann. Um eine hohe Performanz für lange Jobs mit vielen Sortieraktivitäten zu erreichen, soll man den Parameter entsprechend hochsetzen. Dadurch werden Sortierungen vermieden, die auf der Festplatte aufgrund des Speichermangels im Sortierbereichs durchgeführt werden müssen.

- SORT_DIRECT_WRITES (AUTO / TRUE / FALSE)

  Voreingestellt: AUTO

  Der Parameter kann die Performanz bei großen Sortierungen verbessern, wenn das System über viel Hauptspeicher und temporären Plattenspeicher verfügt. Wenn der Parameter zu TRUE gesetzt wird, werden große Puffer bei jeder Sortierungen für direkte I/O-Operationen allokiert. Die Anzahl bzw. Größe dieser Puffer werden durch die Parameter SORT_WRITE_BUFFERS, SORT_WRITE_BUFFER_SIZE kontrolliert. Der ORACLE-Prozeß, der die Sortierung ausführt, schreibt die Sortierdaten direkt auf die Festplatte, indem ganze Puffer an das Betriebssystem übergegeben werden. Wenn der Parameter auf AUTO eingestellt ist, werden die Anzahl und Größe der Puffer für Direct Writes automatisch bestimmt.

- SQL_TRACE (TRUE / FALSE)

  Voreingestellt: FALSE

  Der Parameter aktiviert (TRUE) oder deaktiviert (FALSE) den Trace-Mechanismus der ORACLE-Instanz. Wenn der Mechanismus aktiviert wird, schreibt die Instanz die Performanzstatistiken, die sie nach der Ausführung jeden Befehls erstellt hat, in eine Datei für spätere Analyse. Der Mechanismus eignet sich nur für Optimierung von komplizierten und großen Anfragen. Er verlangsamt die gesamte Performanz des Datenbanksystems, deshalb soll man im normalen Betrieb des Datenbanksystems den Parameter wie voreingestellt lassen.