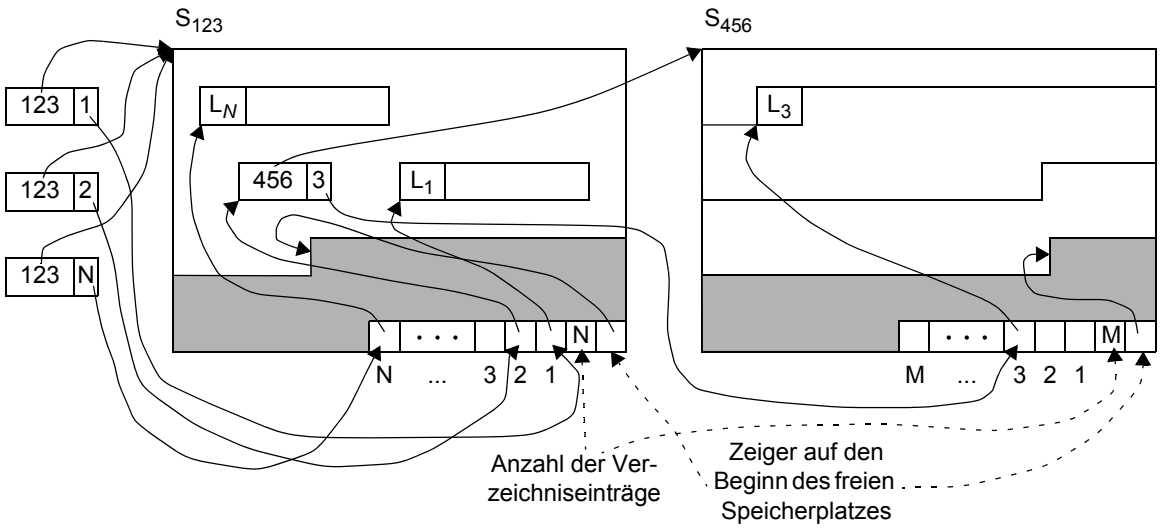
Implementierungskonzepte für Datenbanksysteme

Kurseinheit 1: Architektur eines Datenbanksystems – Externspeicher- und Systempufferverwaltung

Autor: Markus Schneider



Vorwort zur Version ab WS 2004/2005

Liebe Fernstudentin, lieber Fernstudent,

wir freuen uns, daß Sie am Kurs 1664 “Implementierungskonzepte für Datenbanksysteme” teilnehmen und wünschen Ihnen viel Spaß und Erfolg beim Durcharbeiten des Kursmaterials.

Der Kurs wurde von Prof. Markus Schneider verfaßt. Er wird seit 1999 angeboten und nach dem Wechsel von Herrn Schneider an eine amerikanische Universität von mir betreut.

Der Kurs hat einen rechnerischen Kursumfang von 2 SWS. Mir als Betreuer fiel auf, daß der Kurstext relativ dazu sehr viel Material enthält. Darüber hinaus wird das Material der bisher vorhandenen Kapitel 5 “Transaktionen und Concurrency Control” und 6 “Recovery” auch, wenn auch etwas weniger detailliert, im Kurs 1672 “Datenbanken II” behandelt. Die Kurskombination 1664/1672 ist eine beliebte Bachelorprüfung.

Um den Stoffumfang auf ein angemessenes Maß zu reduzieren, habe ich deshalb schon seit einiger Zeit Prüfungskandidaten mitgeteilt, daß die Kapitel 5 und 6 dieses Kurses nicht prüfungsrelevant seien, bzw. daß ich die Inhalte in der Kurskombination nach dem Kurs 1672 prüfe. Es lag nahe, konsequent zu sein und auch den Kurstext entsprechend zu kürzen. Das ist nun geschehen. In der vorliegenden Version sind die früheren Kapitel 5 und 6 nicht mehr vorhanden und das Material ist auf die vier Kurseinheiten entsprechend umverteilt worden. Das frühere Kapitel 7 “Anfrageverarbeitung” ist nun Kapitel 5.

Ich hoffe, daß diese Änderung dazu beiträgt, Ihnen das Studium zu erleichtern.

Im Zuge der Umstrukturierung des Kurses wurde der Text komplett neu formatiert. Es ist kaum zu vermeiden, daß sich dabei kleine Fehler einschleichen (hoffentlich keine großen); dafür bitte ich im Voraus um Entschuldigung. Selbstverständlich bitten wir Sie, uns auf Fehler aufmerksam zu machen.

Ralf Hartmut Güting

Vorwort

Datenbanken und Datenbanktechnologie spielen heutzutage in praktisch allen Bereichen, in denen Computer eingesetzt werden, eine tragende Rolle. Die Nachteile konventioneller Dateisysteme, die zunehmenden Anforderungen an die Datenhaltung und an die Auswertung von Daten sowie neue Anwendungsgebiete sorgten und sorgen für eine rasante Entwicklung und Verbesserung der Konzepte und Techniken von Datenhaltungssystemen. Die Forderung nach verallgemeinerten und standardisierten Funktionen zur Datendefinition und Datenmanipulation sowie zur Integritätsüberwachung und Zugriffskontrolle bewirkten in einem stufenweisen Prozeß die Entwicklung von einfachen Dateisystemen zu allgemeinen *Datenbanksystemen*. Ein Datenbanksystem übernimmt alle Aufgaben der effizienten Datenhaltung und -verwaltung in einem Anwendungssystem. Es zeichnet sich vor allem durch einen hohen Grad an Datenunabhängigkeit, durch ein logisches Datenmodell und durch die Bereitstellung einer Anfragesprache aus. Weiterhin unterstützt es den Mehrbenutzerbetrieb und bietet ein Transaktionskonzept sowie Maßnahmen zur Datensicherung an.

In diesem Kurs werden wir der Frage nachgehen, wie Datenbanksysteme, die bekanntermaßen große Softwaresysteme darstellen, realisiert werden, d.h., welchen Implementierungsanforderungen sie unterliegen, wie die Architektur solcher Systeme aussieht und welche allgemeinen Konzepte für die Implementierung von Datenbanksystemen existieren. Hierbei sind verschiedene Sichtweisen auf Datenbanksysteme möglich, die spezielle Konzepte und Techniken erfordern können. Datenbanksysteme können klassifiziert werden nach dem zugrundeliegenden Datenmodell, nach Anwendungsgebieten und nach Einsatzformen.

Im Laufe der Zeit sind verschiedene Datenbankmodelle entworfen worden, die zur Entwicklung der entsprechenden Datenbanksysteme geführt haben. Diese Modelle beschreiben im wesentlichen verschiedene Sichtweisen auf Datenbankobjekte und den auf ihnen definierten Operationen. Wichtige Beispiele sind das hierarchische Modell, das Netz-

werkmodell, das relationale Modell, das objekt-orientierte Modell und das objekt-relationale Modell.

Aus Anwendungssicht haben sich Datenbanksysteme mittlerweile in den administrativen und betriebswirtschaftlichen Anwendungsgebieten etabliert. „Klassische“ Anwendungsbereiche sind zum Beispiel Personalwesen, Produktionsplanung und -steuerung, Waren- und Materialwirtschaft, Rechtswesen, Medizin, Bankwesen, Buchungswesen, usw. Diesen sogenannten „Standard“-Datenbankanwendungen ist gemein, daß die von ihnen verarbeiteten Daten – im wesentlichen handelt es sich um alphanumerische Daten – eine einfache Struktur aufweisen. Andererseits haben sich in den letzten Jahren in zunehmendem Maße eine ganze Reihe neuer, „nicht-klassischer“ Computeranwendungen herauskristallisiert, die aus der Sicht herkömmlicher Datenbanksysteme als „Nicht-Standard“-Datenbankanwendungen bezeichnet werden. Beispiele sind CAD-Datenbanksysteme für den rechnergestützten Entwurf, geographische Informationssysteme und Geo-Datenbanksysteme für geowissenschaftliche Anwendungen wie Kartographie und Anwendungen zur räumlichen Analyse, Expertendatenbanksysteme, deduktive Datenbanksysteme oder Wissensbankverwaltungssysteme für wissensbasierte Anwendungen wie Expertensystemanwendungen und entscheidungsunterstützende Anwendungen sowie Multimedia-Datenbanksysteme für Anwendungen, die Text, Bild, Ton und Sprache integrieren. Alle diese Nicht-Standard-Anwendungen zeichnen sich durch das Auftreten enormer Datenmengen, durch eine große Anwendungskomplexität und durch komplex strukturierte und große Objekte aus. So werden beispielsweise in CAD-Datenbanksystemen, Geo-Datenbanksystemen und Multimedia-Datenbanksystemen geometrische Objekte wie Punkte, Polylinien, Polygone und Polyeder verwaltet, die eine komplexe Struktur aufweisen und besondere Speicherungs-, Zugriffs- und Anfragemechanismen erfordern.

Unabhängig vom zugrundeliegenden Datenmodell und vom betrachteten Anwendungsgebiet haben sich ausgehend von monolithischen Systemen (wir verstehen hierunter zentralisierte Systeme auf Ein-Prozessor-Rechnern) unterschiedliche neue Einsatzformen von Datenbanksystemen entwickelt. Diese beruhen im wesentlichen auf Entwicklungen im Bereich der Hardware (Workstation-Server-Konfigurationen, Mehrprozessorsysteme, homogene/heterogene Rechnernetze, erweiterte Speicherarchitekturen mit sehr großen und teilweise nicht-flüchtigen Hauptspeichern) und auf der Zunahme verteilter Systeme (insbesondere von Client-Server-Systemen). Entsprechende Datenbanksysteme umfassen verteilte Datenbanksysteme, Client-Server-Datenbanksysteme und parallele Datenbanksysteme.

Die Mannigfaltigkeit der Entwicklungsrichtungen weist auf das weite Spektrum der Anforderungen in bezug auf die Architektur und die Implementierung von Datenbanksystemen hin und zwingt aus Platzgründen zu einer Auswahl des behandelten Lernstoffs. In

diesem Kurs werden wir uns auf Implementierungskonzepte und -techniken konzentrieren, die (im wesentlichen) allen Arten von Datenbanksystemen gemeinsam sind. An einigen Stellen erfolgt eine Einschränkung auf ein bestimmtes Datenmodell. So werden wir uns beispielsweise bei der Behandlung der Anfrageverarbeitung und -optimierung auf das relationale Modell beschränken. Aus Anwendungssicht stehen Standard-Datenbanksysteme im Vordergrund. Allerdings werden wir auch Aspekte diskutieren, die sich speziell und ausschließlich auf Nicht-Standard-Datenbanksysteme beziehen. So besprechen wir bei der Betrachtung von Indexstrukturen auch Indexstrukturen für geometrische Objekte, wie sie in Geo- und CAD-Datenbanksystemen vorkommen. Was die verschiedenen Einsatzformen von Datenbanksystemen anbetrifft, so beschränken wir uns im gesamten Kurs auf monolithische Systeme. Auf Einschränkungen und Spezialisierungen der einzelnen Themenbereiche wird an den entsprechenden Stellen hingewiesen.

Globales Lernziel dieses Kurses ist es, ein grundlegendes Verständnis für die Architektur und die Implementierung von Datenbanksystemen zu entwickeln. Hierbei wird auch Wert auf die Einführung der englischen Fachbegriffe gelegt, die jeweils in runden Klammern angegeben werden. Dies soll den Studenten beim Lesen englischsprachlicher Fachliteratur unterstützen. Manchmal wird auch eine Übernahme eines englischen Fachbegriffs erfolgen, wenn eine deutsche Übersetzung oder ein gleichwertiger deutscher Begriff zu ungelenk erscheint.

Der Kurs ist in sieben Kapitel unterteilt. Jedes Kapitel enthält am Ende Literaturhinweise, die dem Leser eine Vertiefung des Stoffs ermöglichen. Kapitel 1 beschäftigt sich mit der Architektur von Datenbanksystemen und den Anforderungen an ihre Implementierung. Die traditionelle Datenorganisation in Dateisystemen mit ihren gravierenden Nachteilen wird den Datenbanksystemen mit ihren Vorteilen gegenübergestellt. Ferner wird ein abstraktes 3-Ebenen-Modell vorgestellt, das heute weitgehend als Grundlage für den Aufbau von Datenbanksystemen gilt. Schließlich wird aus der Sicht des Software-Engineering eine hierarchische und modulare Systemarchitektur für ein Datenbanksystem beschrieben.

Kapitel 2 behandelt das Speichersystem eines Datenbanksystems. Die wesentlichen Aufgaben eines Speichersystems untergliedern sich in die beiden Bereiche der Externspeicherverwaltung und der Systempufferverwaltung. Wesentliches Ziel der Externspeicherverwaltung ist es, die Daten der Datenbank auf einem externen Speichermedium persistent und möglichst speicherplatzeffizient zu speichern und gleichzeitig einen möglichst laufzeiteffizienten Zugriff auf diese Daten zu ermöglichen. Um die Daten einer Datenbank verarbeiten zu können, müssen sie vom Externspeicher in einen bestimmten Bereich des Hauptspeichers, dem Systempuffer, geladen und bei Änderung später wieder vom

Hauptspeicher auf den Externspeicher zurückgeschrieben werden. Um die hierbei auftretenden Probleme kümmert sich die Systempufferverwaltung.

Kapitel 3 führt Indexstrukturen als spezialisierte, externe Datenstrukturen zum effizienten Zugriff auf die Daten einer Datenbank ein, die eine sonst erforderliche sequentielle Suche vermeiden. Nach der Erläuterung des Begriffs und der Aufgaben einer Indexstruktur werden verschiedene Klassifikationen für Indexstrukturen vorgestellt. Anschließend werden aus der Fülle der vorhandenen Indexstrukturen wichtige Vertreter betrachtet. Behandelt werden zum einen Indexstrukturen für alphanumerische Daten wie baumbasierte und hashbasierte Indexstrukturen und zum anderen geometrische Indexstrukturen, die weniger bekannt sind, aber in wesentlichem Maße Bedeutung für die Unterstützung von räumlichen oder geometrischen Datenbanksystemen besitzen. Nach einer kurzen Charakterisierung geometrischer Objekte, Operationen und Anfragetypen sowie der Aufgaben und Eigenschaften geometrischer Indexstrukturen werden als wichtige Vertreter eindimensionale Einbettungen, externe Strukturen für Punktmengen (z.B. das Grid-File) und externe Strukturen für Rechteckmengen (z.B. die R-Baum-Familie) beschrieben.

Kapitel 4 befaßt sich mit dem externen Sortieren großer externer Datenbestände auf Sekundärspeicher anhand eines Sortierschlüssels. Diese Sortierverfahren zeichnen sich im Gegensatz zu internen Verfahren durch einen sequentiellen Datenzugriff aus. Ziel ist es, beim Sortieren die Anzahl externer Seitenzugriffe zu minimieren. Sortieren ist eine wichtige Funktion eines Datenbanksystems und wird z.B. zur Beantwortung von Benutzeranfragen in einer gewünschten Sortierreihenfolge, zur Eliminierung von Duplikaten in einer Menge von Datensätzen, zur Unterstützung der Implementierung bestimmter relationaler Algebraoperationen wie einer bestimmten Join-Variante oder zur Erzeugung von Partitionen durch Zerlegung einer Datensatzmenge in disjunkte Gruppen benötigt. Als wichtigste Vertreter werden verschiedene Mergesort-Varianten betrachtet.

Kapitel 5 hat Transaktionen und Concurrency Control zum Thema. Eine Datenbank stellt eine Informationsbasis dar, auf die von vielen verschiedenen Anwendungsprogrammen und interaktiven Benutzern unabhängig, ohne Wissen voneinander und insbesondere nebenläufig zugegriffen wird. Es ist die Aufgabe des Concurrency Control, den nebenläufigen Zugriff auf geteilte Daten zu steuern und die beteiligten Prozesse zu synchronisieren, um inkonsistente Datenbankzustände zu verhindern. Synchronisation im Datenbankbereich bedeutet Serialisierung konkurrierender Zugriffe auf gemeinsam benutzte Datenbankobjekte. Das grundlegende Konzept für Concurrency Control (und Recovery) ist die Transaktion. Sie bezeichnet eine Folge von zusammengehörenden Operationen, die eine logische Arbeitseinheit bilden und entweder vollständig oder gar nicht ausgeführt werden. Einen breiten Raum nimmt die Darstellung von Sperrverfahren ein, die wohl als das wichtigste Synchronisationsmittel für Transaktionen in Datenbanksystemen anzusehen sind.

Kapitel 6 befaßt sich mit denjenigen Funktionen eines Datenbanksystems, die die Wiederherstellung eines korrekten Datenbankzustands nach dem Auftreten eines Datenbankfehlers ermöglichen und die Unteilbarkeit und Dauerhaftigkeit von Transaktionen sicherstellen. Unteilbarkeit wird erreicht, indem die Operationen aller nicht beendeten Transaktionen abgebrochen und rückgängig gemacht werden. Dauerhaftigkeit wird bewirkt, indem sichergestellt wird, daß alle Operationen von beendeten Transaktionen Fehler überleben. Die Wiederherstellung des letzten konsistenten Datenbankzustands wird auch Recovery genannt. Verschiedene Recovery-Techniken werden erläutert und Algorithmen hierzu im Rahmen der in Datenbanksystemen weit verbreiteten Log-basierten Recovery vorgestellt.

Kapitel 7 behandelt die Verarbeitung von Anfragen. Die Möglichkeit, mit Hilfe einer Anfragesprache Anfragen an eine Datenbank zu stellen, sei es durch ein Anwendungsprogramm oder aber ad hoc durch den Endbenutzer am Computer, und unmittelbar darauf eine Antwort zu erhalten, gehört mit zu den herausragenden Eigenschaften eines Datenbanksystems. Ziel der Anfrageverarbeitung ist es, eine gegebene Anfrage unter Ausnutzung logischer Gesetzmäßigkeiten und externer physischer Speicherungsstrukturen möglichst effizient auszuführen. Der Übersetzung einer Anfrage (lexikalische, syntaktische und semantische Analyse) folgt deren Optimierung und Ausführung. Die Optimierung einer Anfrage ist erforderlich, um aus der Menge der möglichen Ausführungsalternativen für eine Anfrage möglichst eine mit minimalen Kosten auszuwählen. Nach einer Anfrageumformung auf der Basis von Äquivalenzregeln (algebraische Optimierung) werden verschiedene Auswertungspläne erzeugt, die mit Hilfe der Kostenschätzung bewertet werden und zum letztendlich „optimalen“ Auswertungsplan führen. Eine effiziente Anfrageauswertung hängt natürlich insbesondere von der physischen Speicherung der Daten ab. Vorhandene Indexe können die Anfrageverarbeitung beträchtlich beschleunigen. Am Beispiel relationaler Operationen werden verschiedene Implementierungs- und Auswertungsmöglichkeiten aufgezeigt.

Der Kurs wendet sich an Studenten, die bereits Erfahrungen mit Datenbanksystemen haben und mehr über die Interna dieser Systeme erfahren möchten. Hilfreich sind allgemeine Kenntnisse über Datenbanksysteme, die man zum Beispiel über das Studium des Kurses 01665 „Datenbanksysteme“ erlangen kann. Studenten, die sich für verschiedene Einsatzformen von Datenbanksystemen interessieren, sei der Kurs 01666 „Datenbanken in Rechnernetzen“ empfohlen. Hier werden Fragestellungen verteilter Datenbanksysteme und Client-Server-Systeme behandelt.

Markus Schneider

Inhalt

Lehrziele

Nach der Bearbeitung der ersten Kurseinheit sollten Sie

- die Vor- und Nachteile von Datenbanksystemen insbesondere im Vergleich zu konventioneller Datenorganisation in Dateisystemen erläutern können,
- die Anforderungen an die Architektur von Datenbanksystemen insbesondere aus dem Blickwinkel der Implementierung beschreiben können,
- das *3-Ebenen-Modell* und den Begriff der *Datenunabhängigkeit* erklären können,
- die Software-Architektur eines Datenbankmanagementsystems beschreiben und die wesentliche Funktionalität der einzelnen Softwarekomponenten erklären können,
- die Aufgaben der *Externspeicherverwaltung* und der *Systempufferverwaltung* für das Speichersystem eines Datenbanksystems verstanden haben,
- in der Lage sein, verschiedene *Datensatzformate* und *Seitenformate* für Datensätze fixer, variabler und sehr großer Länge zu beschreiben und die Abbildung von Datensätzen in Seiten zu erklären,
- wissen, was *Clusterung* bedeutet,
- das *Dateikonzept* sowie grundlegende *Dateiorganisationen* verstanden haben,
- die Bedeutung des *Systemkatalogs* für Datenbanksysteme einschätzen können.

Kapitel 1

Architektur eines Datenbanksystems

Datenbanksysteme und Datenbanken sind heutzutage in allen Bereichen zu finden, in denen Computer auftreten. Eine stetig wachsende Zahl von neuen Anwendungsgebieten (insbesondere von Nicht-Standard-Anwendungen) und daraus resultierende neue Anforderungen an Datenbanksysteme haben bis heute zu einer geradezu stürmischen Entwicklung der Datenbanktechnologie geführt. Ziel dieses Kurses ist es, wesentliche Elemente dieser Technologie zu vermitteln und zu zeigen, wie Datenbanksysteme aufgebaut sind, aus welchen modularen Komponenten sie bestehen und wie diese realisiert werden können.

In diesem einleitenden Kapitel beschäftigen wir uns mit der Architektur von Datenbanksystemen und den Anforderungen bezüglich ihrer Implementierung. Abschnitt 1.1 beschreibt zunächst kurz die Nachteile traditioneller Datenorganisation in Dateisystemen, die letztendlich zur Entwicklung von Datenbanksystemen führten. Danach erläutern wir das Konzept des Datenbanksystems. Wir führen die wichtigste Terminologie ein und klären Begriffe wie „Datenbank“, „Datenbanksystem“, „Datenbankmanagementsystem“ und „Datenmodell“. Abschnitt 1.2 umreißt das breite Spektrum der Anforderungen an Datenbanksysteme, insbesondere aus Implementierungssicht. Abschnitt 1.3 stellt ein 3-Ebenen-Modell vor, das heute weitgehend als Grundlage für den Aufbau von Datenbanksystemen dient. Die physische Ebene als Bestandteil des 3-Ebenen-Modells steht im Mittelpunkt unserer Betrachtungen. Abschnitt 1.4 betrachtet DBMS aus der Sicht des Software-Engineering und beschreibt ihren Aufbau anhand einer hierarchischen und modularen Systemarchitektur. Abschnitt 1.5 beschreibt zusätzliche Komponenten in Form von Werkzeugen und Hilfsprogrammen.

1.1 Konzept des Datenbanksystems

Das Wesen traditioneller Datenverwaltung in von Betriebssystemen unterstützten Dateisystemen ist, daß jeder Anwendungsprogrammierer diejenigen Dateien definiert, die er für seine Anwendung braucht, unabhängig und vielleicht sogar ohne Kenntnis der Dateien von Anwendungen anderer Programmierer. Der Dateiaufbau ist unmittelbar an die jeweilige Verarbeitung angepaßt, und dementsprechend ist die Datei auch physisch abgespeichert. Die traditionelle Datenverwaltung mittels Dateisystemen führt im wesentlichen zu folgenden schwerwiegenden Problemen:

- *Redundanz* (*redundancy*). Die anwendungsspezifische Gestaltung von Dateien führt zu wiederholtem Auftreten von gleichen Daten in verschiedenen Dateien. Die dadurch herbeigeführte Redundanz hat insbesondere bei Änderungen Speicherverschwendung und erhöhten Verwaltungs- und Verarbeitungsaufwand zur Folge und wird in der Regel nicht zentral kontrolliert. Hieraus ergeben sich die folgenden, weiteren Probleme.

- *Inkonsistenz* (*inconsistency*). Die Konsistenz der Daten, d.h., die logische Übereinstimmung der Dateiinhalte, kann nur schwer aufrechterhalten werden. Bei der Änderung eines Datums müssen alle Dateien geändert und angepaßt werden, die dieses Datum enthalten. Ferner müssen diese Änderungen so aufeinander abgestimmt werden, daß nicht verschiedene Programme zum selben Zeitpunkt auf unterschiedliche Werte desselben Datums zugreifen können.

- *Daten-Programm-Abhängigkeit*. Ein weiteres Problem ergibt sich durch die sehr enge Abhängigkeit zwischen Anwendungsprogramm und Datenorganisation; das Anwendungsprogramm hat nämlich einen direkten Zugang zu den Daten einer Datei. Ändert sich der Aufbau einer Datei oder ihre Organisationsform, so müssen alle darauf basierenden Programme geändert werden. Umgekehrt kann eine Erweiterung der Funktionalität eines Anwendungsprogramms neue Anforderungen an den Dateiaufbau stellen und eine Restrukturierung von Dateien bewirken.

- *Inflexibilität*. Da die Daten nicht anwendungsneutral und in ihrer Gesamtheit, sondern ausschließlich anwendungsspezifisch gesehen werden, erweist sich die Realisierung neuer Anwendungen sowie die Auswertung vorhandener Daten als problematisch. Dies gilt insbesondere für Anwendungen und Auswertungen, die Daten aus verschiedenen Dateien benötigen würden. Mangelnde Anpassungsfähigkeit ist daher ein wesentliches Kennzeichen der traditionellen Datenorganisation.

Selbsttestaufgabe 1. Zwei Sachbearbeiter einer Universitätsverwaltung seien mit unterschiedlichen Aufgaben betraut. Sachbearbeiter *A* im Prüfungsamt ist für die Verwaltung der Leistungsdaten der Studenten zuständig. Er erfaßt für jeden Studenten neben privaten Daten die Ergebnisse der mündlichen Prüfungen, Klausuren, Seminare usw. Sachbearbeiter *B* im Studentensekretariat verwaltet neben privaten Daten die belegten Kurse/Vorlesungen der Studenten.

Skizzieren Sie für dieses Szenario eine traditionelle Datenorganisation, indem Sie davon ausgehen, daß jeder Sachbearbeiter eine, auf die jeweilige Anwendung zugeschnittene Datei verwendet, und erläutern Sie beispielhaft die oben genannten Nachteile.

Datenbanksysteme fielen daher nicht vom Himmel, sondern entstanden aus den erkannten Nachteilen traditioneller Datenorganisation sowie aus den zunehmenden Anforderungen an die Verwaltung und Analyse großer Datenbestände. Während bei der traditionellen Datenorganisation die Anwendungsprogramme im Vordergrund standen und die Daten mehr als deren Anhängsel betrachtet wurden, änderte sich nun die Sichtweise. Die Daten selbst stehen nun im Mittelpunkt der Betrachtung und werden als eigenständig und wesentlich angesehen. Die Daten werden einmal definiert und für alle Benutzer zentral und als integriertes Ganzes verwaltet.

Dies ist der entscheidende Schritt zum Konzept des Datenbanksystems, das wir im folgenden beschreiben werden. Beginnen wollen wir mit dem Begriff der Datenbank, für den sich eine präzise Beschreibung allerdings schwerlich finden läßt. Folgende Charakterisierung trifft jedoch das Wesentliche:

> Eine *Datenbank*[1] (*database*), kurz *DB*, ist eine integrierte und strukturierte Sammlung persistenter Daten, die allen Benutzern eines Anwendungsbereichs als gemeinsame und verläßliche Basis aktueller Information dient.

Das Attribut *integriert* (*integrated data*) bedeutet, daß eine Datenbank eine vereinheitlichende und anwendungsneutrale Gesamtsicht auf die interessierenden Daten bietet, die den natürlichen Gegebenheiten und Zusammenhängen der Anwendungswelt entspricht. Insbesondere sind die Daten also nicht danach angeordnet, wie einzelne Anwendungen sie benötigen. Das Attribut *strukturiert* (*structured data*) besagt, daß eine Datenbank keine zufällige Zusammenstellung von Daten darstellt, sondern daß logisch kohärente Informationseinheiten identifizierbar sind, deren zugehörige Daten (möglichst) redundanzfrei gespeichert werden. Das Attribut *persistent* (*persistent data*) beschreibt die

1. Manchmal wird neben dem Begriff der Datenbank auch der Begriff der *Datenbasis* verwendet. Wir unterscheiden nicht zwischen diesen beiden Begriffen.

Eigenschaft, daß die Daten in der Datenbank dauerhaft auf externen Speichermedien verfügbar sein sollen. Diese Daten unterscheiden sich also zum Beispiel von flüchtigen Ein- und Ausgabedaten. Die Daten in der Datenbank bilden eine *gemeinsame Basis* (*shared data*) für alle Benutzer und Anwendungsprogramme zu jeweils eigenen Zwecken. Auf gleiche Daten kann sogar von verschiedenen Benutzern gleichzeitig zugegriffen werden (*concurrent access*), wobei dafür gesorgt werden muß, daß sich diese nicht gegenseitig stören. Das Attribut *verläßlich* (*reliable data*) besagt, daß trotz etwaiger Systemabstürze oder Versuche des unautorisierten Zugriffs stets für die Sicherheit der gespeicherten Information gesorgt werden muß.

Alle Anwendungsprogramme und Benutzer arbeiten also auf einem gemeinsamen Datenbestand; sie greifen nun aber nicht mehr direkt auf die abgespeicherten Daten zu, sondern erhalten die gewünschten Daten durch das sogenannte Datenbankmanagementsystem. Dadurch wird erreicht, daß ihnen Betriebsystem- und Hardwaredetails verborgen bleiben.

> Ein *Datenbankmanagementsystem* (*database management system*), kurz *DBMS*, ist ein All-Zweck-Softwaresystem, das den Benutzer bei der Definition, Konstruktion und Manipulation von Datenbanken für verschiedene Anwendungen applikationsneutral und effizient unterstützt.

Zwischen der physischen Datenbank und seinen Benutzern liegt also eine Softwareschicht, die aus einer Menge von Programmen zur Verwaltung und zum Zugriff auf die Daten in der Datenbank besteht. Die Definition einer Datenbank umfaßt die Spezifikation der Typen der Daten, die in der Datenbank gespeichert werden sollen, mit einer entsprechenden Beschreibung jedes Datentyps. Ferner umfaßt sie die Angabe von Strukturen für die Speicherung von Werten dieser Datentypen. Die Konstruktion der Datenbank beinhaltet den Prozeß der Speicherung der Daten auf einem externen Speichermedium, das von dem DBMS kontrolliert wird. Die Manipulation einer Datenbank umfaßt solche Funktionen wie das Stellen von Anfragen (*queries*) zum Auffinden (*retrieval*) spezieller Daten, das Aktualisieren (*update*) der Datenbank und die Generierung von Berichten über die Daten. Die Aufgaben eines DBMS werden wir ausführlich in Abschnitt 1.2 beschreiben.

> Ein *Datenbanksystem* (*database system*), kurz *DBS*, faßt die beiden Komponenten Datenbankmanagementsystem und Datenbank zusammen: DBS = DBMS + DB.

Die Kernaufgabe eines DBS besteht also darin, große Mengen von strukturierten Informationen entgegenzunehmen, effizient zu speichern und zu verwalten, und auf Anforderung bereitzustellen. Selbstverständlich kann dasselbe DBMS (in einer oder mehreren Kopien) mehrere Datenbanken verwalten und damit mehrere DBS bilden. Wo Zweifel ausgeschlossen sind, werden wir trotzdem, wie dies auch allgemein üblich ist, den Begriff

DBS synonym zu DBMS verwenden. Bild 1.1 zeigt eine stark vereinfachte Sicht eines Datenbanksystems.

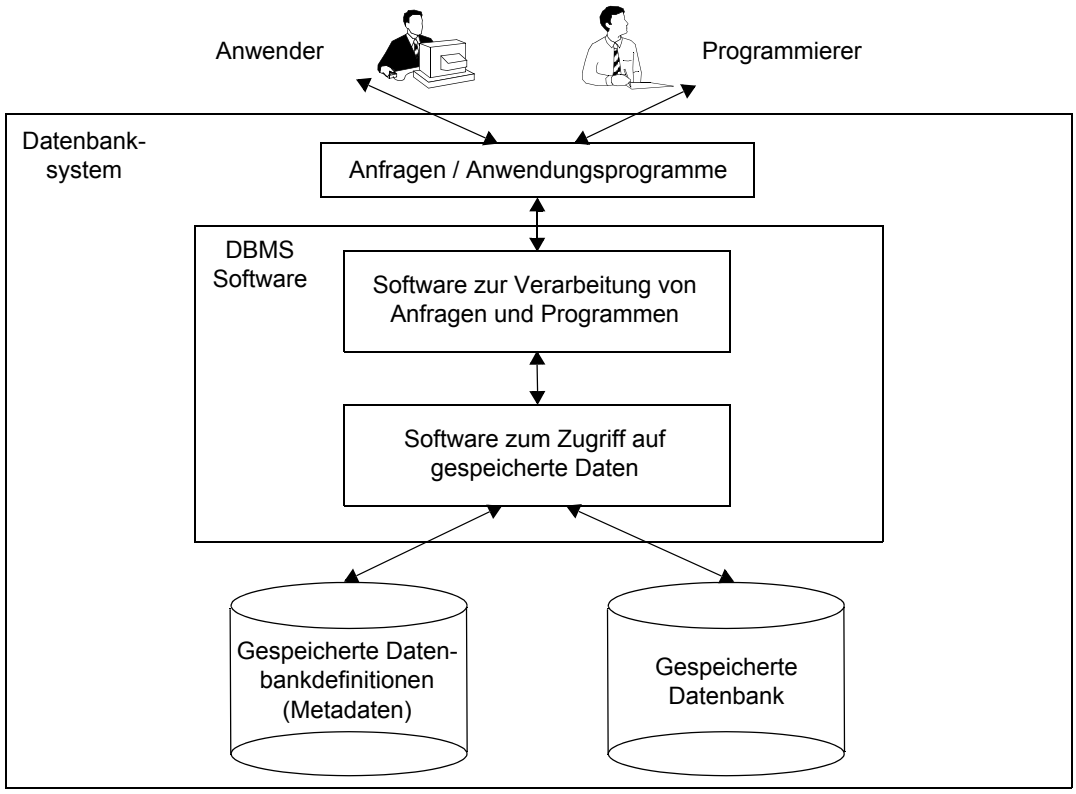


Bild 1.1: Eine stark vereinfachte Sicht eines Datenbanksystems

Jedem Datenbanksystem liegt ein abstraktes Datenmodell zugrunde, das dem Benutzer eine bestimmte Sicht auf die Daten der Datenbank bietet.

> Ein *Datenmodell* (*data model*) ist ein mathematischer Formalismus, der aus einer Notation zur Beschreibung der interessierenden Daten und aus einer Menge von Operationen zur Manipulation dieser Daten besteht.

Ein solches Datenmodell erlaubt es, die Struktur einer Datenbank – hierunter verstehen wir die Datentypen, Beziehungen und Bedingungen, die auf den Daten gelten sollen – angemessen zu beschreiben und dem Benutzer Daten in verständlicheren Begriffen zur Verfügung zu stellen. Daten kann man auf verschiedenen Abstraktionsebenen betrachten, wie wir in Abschnitt 1.3 erkennen werden. Wichtige Datenmodelle für DBS sind das relationale und das objekt-orientierte Datenmodell sowie das objekt-relationale Datenmodell, das zunehmend an Popularität gewinnt, indem es versucht, die positiven Eigenschaften des relationalen und des objekt-orientierten Modells zu verbinden. Ältere Modelle sind das hierarchische und das Netzwerkmodell.

1.2 Anforderungen an Datenbanksysteme

Moderne Datenbanksysteme bieten ein breites Spektrum von Funktionen an und haben eine Vielzahl von Anforderungen zu erfüllen, die sich in der Architektur und somit in der Implementierung solcher Systeme widerspiegeln (siehe Abschnitt 1.4). Einerseits werden die Nachteile der traditionellen Datenorganisation behoben, aber darüberhinaus werden eine ganze Anzahl weiterer Funktionen für den Benutzer sichtbar oder unsichtbar bereitgestellt. Alle im folgenden beschriebenen Anforderungen werden mehr oder minder in heutigen Datenbanksystemen realisiert und stellen somit gleichsam die Vorteile solcher Systeme dar. Wir betrachten diese Anforderungen insbesondere aus dem Blickwinkel der Implementierung.

- *Datenunabhängigkeit* (*data independence*). Anwendungsprogramme sollten so unabhängig wie möglich von den Einzelheiten der Datenrepräsentation und -speicherung sein. Das DBMS sorgt hierzu für eine abstrakte Sicht auf die Daten (siehe auch Abschnitt 1.3).

- *Effizienter Datenzugriff*. Ein DBMS verwendet eine Vielzahl von ausgeklügelten Techniken zur effizienten Speicherung von und zum effizienten Zugriff auf Daten. Dies ist insbesondere dann von Bedeutung, wenn die Datenmenge so groß ist, daß sie nicht im Hauptspeicher gehalten werden kann, sondern auf einem externen Medium gespeichert werden muß. Beabsichtigen wir zum Beispiel, in einer Datenbank, die Tausende von Artikeln gespeichert hat, anhand der Artikelnummer nach einem bestimmten Artikel zu suchen, ist es sehr kostenintensiv, den gesamten Datenbestand zu durchlaufen und jeden Datensatz mit der gesuchten Artikelnummer zu vergleichen. Effizienter ist es, einen *Index* einer *Indexstruktur* (*index structure*) über den Artikelnummern zu benutzen, der einen direkten Zugriff auf den gesuchten Artikel erlaubt.

- *Gemeinsame Datenbasis*. Alle jetzigen und zukünftigen Anwendungsprogramme und Benutzer greifen auf einen gemeinsamen Datenbestand zu (siehe auch Abschnitt 1.1).

- *Nebenläufiger Datenzugriff* (*concurrent access*). Ein DBMS ermöglicht es, daß auf gleiche Daten von verschiedenen Benutzern quasi gleichzeitig zugegriffen werden kann und daß jedem Benutzer der (fälschliche) Eindruck eines exklusiven Zugriffs auf diese Daten vermittelt wird. Versuchen zum Beispiel zwei Benutzer, das gleiche Datum gleichzeitig zu verändern, kann eine Änderung verloren gehen, weil sie durch den Wert der anderen Änderung überschrieben werden kann. DBMS benutzen das Konzept der *Transaktionen* (*transactions*), um nebenläufigen Datenzugriff zu

steuern (*Synchronisation*) und dafür zu sorgen, daß sich zwei Zugriffe nicht gegenseitig beeinflussen. Eine Transaktion ist eine konsistenzerhaltende Operation, die nach Ausführung die Datenbank in einem konsistenten Zustand zurückläßt, wenn diese vor Beginn der Transaktion schon konsistent war.

- *Fehlende oder kontrollierte Redundanz* (*redundancy*). Die Nachteile der traditionellen Datenorganisation werden in DBS durch eine integrierte Sicht der Daten, die Kopien desselben Datums nicht erlaubt, vermieden. Allerdings wird in gewissem Rahmen eine begrenzte und vom DBMS *kontrollierte Redundanz* (*controlled redundancy*) zugelassen, zum Beispiel um die logischen Beziehungen zwischen Daten aufrechtzuerhalten oder um die Leistungsfähigkeit oder *Performance* zu verbessern. Für konsistente Änderungen auf verschiedenen Kopien des gleichen Datums ist dann das DBMS verantwortlich.

- *Konsistenz der Daten* (*consistency*). Fehlende Redundanz der Daten bedingt ihre Konsistenz. Wenn ein Datum nur einmal in der Datenbank auftritt, muß jede Änderung eines Wertes nur einmal durchgeführt werden, und alle Benutzer haben sofortigen Zugriff auf diesen neuen Wert. Bei kontrollierter Redundanz muß das DBMS sicherstellen, daß die Datenbank aus der Sicht des Benutzers niemals inkonsistent ist. Dies geschieht durch automatisches Propagieren der Änderungen (*propagating updates*) eines Datums zu all seinen Kopien.

- *Integrität der Daten* (*integrity*). Integrität bedeutet ganz allgemein Korrektheit und Vollständigkeit der Daten. Selbst bei einer redundanzfreien Datenbank können die abgespeicherten Daten semantisch falsch sein. Zum Beispiel könnte die Datenbank die wöchentliche Arbeitszeit eines Angestellten mit 400 statt 40 Stunden ausweisen oder ihn der nicht existierenden Abteilung D17 zuordnen. *Integritätsbedingungen* oder *-regeln* (*integrity constraints*) sind ein Mittel, um solche Integritätsverletzungen zu vermeiden. Beispielsweise können solche Regeln festlegen, daß ein Angestellter zwischen 10 und 40 Stunden wöchentlich arbeitet und daß es die Abteilungen D1 bis D7 gibt. Das DBMS muß diese Bedingungen dann beim Einfügen, Ändern und Löschen von Daten überprüfen. Inkonsistenz ist ein Spezialfall mangelnder Integrität.

- *Datensicherheit* (*security*). Datensicherheit bezeichnet den Schutz der Datenbank vor unautorisiertem Zugriff. Beispielsweise sollte ein Personalsachbearbeiter einer Bank, der für Gehaltsabrechnungen verantwortlich ist, nur die Daten der Bankangestellten aber nicht die Daten der Kundenkonten einsehen können, d.h., er erhält eine bestimmte *Sicht* (*view*) auf die Daten. Dies erfordert eine *Zugriffskontrolle* (*access control*) mit Authentisierung und Verschlüsselung als möglichen Mechanismen zur Sicherung. Authentisierung bedeutet, daß Sicherheitsregeln, Sicherheitsverfahren

oder zusätzliche Kennwörter für jede Zugriffsart (Lesen, Einfügen, Löschen, Ändern, usw.) die Zugriffserlaubnis eines Benutzers auf bestimmte, sensitive Daten überprüfen. Zusätzlich kann das DBMS die Daten vor der Abspeicherung verschlüsseln. Wenn dann ein autorisierter Benutzer auf die entsprechenden Daten zugreifen möchte, werden sie von ihm unbemerkt automatisch entschlüsselt und zur Verfügung gestellt. Daten, auf die unautorisiert zugegriffen wird, erscheinen in verschlüsselter Form.

- *Bereitstellung von Backup- und Recovery-Verfahren*. Ein DBMS verfügt über Mechanismen, um Benutzer vor den Auswirkungen von Systemfehlern zu schützen. Meist nachts werden Sicherungskopien der Datenbank auf Bändern vorgenommen (*backup*). Während des Tagesablaufs wird diese Maßnahme gewöhnlich durch ein Protokoll der durchgeführten Änderungen ergänzt. Wird die Datenbank modifiziert, erfolgt ein Protokolleintrag. Bei einem Systemabsturz werden die Bänder und das Protokoll dazu benutzt, in der Datenbank den zuletzt aktuellen Zustand automatisch wiederherzustellen (*recovery*).

- *Stellen von Anfragen*. In der traditionellen Datenorganisation muß eine Anfrage an den Datenbestand stets mittels eines eigenen Anwendungsprogramms gestellt werden. Dies ist extrem inflexibel. DBMS stellen eine *Anfragesprache* (*query language*) zur Verfügung, die es dem Benutzer erlaubt, ad hoc am Bildschirm mittels der Tastatur eine Anfrage zu stellen und unmittelbar eine Antwort zu erhalten. Ferner kann eine solche Anfragesprache auch in eine Programmiersprache eingebettet sein. Eine wichtige Anforderung an das DBMS ist nun, eine solche Anfrage möglichst effizient zu verarbeiten (*query processing*), d.h., eine lexikalische Analyse (*query scanning*) und Syntaxanalyse (*query parsing*) der Anfrage durchzuführen, die Anfrage zu optimieren (*query optimization*) und effizient auszuführen (*query execution*).

- *Bereitstellung verschiedenartiger Benutzerschnittstellen*. Weil viele Arten von Benutzern mit unterschiedlichem inhaltlichen und technischen Wissen eine Datenbank nutzen, muß ein DBMS verschiedenartige Benutzerschnittstellen zur Verfügung stellen. Mögliche Schnittstellen sind Anfragesprachen für gelegentliche Benutzer, Programmierschnittstellen für Anwendungsprogrammierer, menügesteuerte Schnittstellen für naive Anwender, fenster- und graphikorientierte Benutzeroberflächen, *Datendefinitionssprachen* (*data definition languages*), *Datenmanipulationssprachen* (*data manipulation language*), usw.

- *Flexibilität*. Hierunter ist einerseits die Anforderung zu verstehen, daß die Struktur einer Datenbank auf bequeme Weise geändert werden kann, ohne existierende Anwendungsprogramme verändern zu müssen. Zum Beispiel sollte es möglich sein,

Datensätze um ein neues Feld zu erweitern oder eine neue Kollektion von Daten in die Datenbank einzufügen. Ferner besteht die Anforderung, auf einfache Weise Daten nach anderen Gesichtspunkten als bisher vorgesehen auswerten zu können.

- *Schnellere Entwicklung von neuen Anwendungen*. Ein DBMS bietet wichtige Funktionen an, die von vielen auf die Daten der Datenbank zugreifenden Anwendungen benötigt werden. Zum Beispiel können komplexe Anfragen leicht durch eine Anfragesprache beantwortet werden; nebenläufiger Zugriff und Fehlerbehandlung bei einem Systemabsturz werden vom DBMS unterstützt. Alle diese Funktionen erlauben eine schnelle Entwicklung von Anwendungen. Darüberhinaus sind Anwendungen, die auf Funktionen des DBMS aufbauen, meist robuster, weil viele Aufgaben bereits vom DBMS übernommen werden und nicht mehr von der Anwendung implementiert werden müssen.

Selbsttestaufgabe 2. Beschreiben Sie, welche Nachteile Datenbanksysteme haben können.

1.3 Das 3-Ebenen-Modell

Die Nachteile traditioneller Datenorganisation sowie die in Abschnitt 1.2 beschriebenen Anforderungen führten 1975 zum Vorschlag eines 3-Ebenen-Modells, einer abstrakten Architektur für DBS, durch das ANSI/SPARC-Standardisierungskomitee. Dieses Modell ist heute im wesentlichen Grundlage aller modernen DBS. Sein erklärtes Ziel ist es, dem Benutzer eine abstrakte Sicht auf die von ihm benötigten Daten zu geben und diese Sicht von der konkreten, physischen Sicht zu trennen. Das Modell besteht aus drei Abstraktionsebenen (Bild 1.2):

- Die *physische/interne Ebene* (*physical/internal level*) basiert auf einem *physischen/internen Schema* (*physical/internal schema*), das die physischen und persistenten Speicherstrukturen der Datenbank beschreibt. Dieses Schema benutzt ein *physisches Datenmodell* und beschreibt für die Datenbank alle Implementierungseinzelheiten über den Aufbau der Daten, die Datenspeicherung und die Zugriffspfade. Hier wird also beschrieben, *wie* Daten gespeichert werden.

- Die *konzeptuelle Ebene* (*conceptual level*) beruht auf einem *konzeptuellen Schema* (*conceptual schema*), das mittels eines vom DBMS bereitgestellten Datenmodells (z.B. relationales Modell) die logische Struktur der gesamten Datenbank für eine Gemeinschaft von Benutzern erfaßt. Das konzeptuelle Schema ist eine globale

Beschreibung der Datenbank, die Details der physischen Speicherstrukturen verbirgt, von ihnen abstrahiert und sich darauf konzentriert, Entitäten, Datentypen, Beziehungen und Integritätsbedingungen zu beschreiben. Hier wird also modelliert, *was* für Daten gespeichert werden.

- Die *externe Ebene* (*external level*, *view level*) umfaßt eine Anzahl von *externen Schemata* (*external schema*) oder *Benutzersichten* (*user view*). Jedes externe Schema beschreibt die Datenbanksicht einer Gruppe von Datenbanknutzern. Jede Sicht beschreibt typischerweise den Teil der Datenbank, für den sich eine bestimmte Benutzergruppe interessiert (und auf den sie auch zugreifen darf) und verbirgt den Rest der Datenbank vor ihr.

Transformationsregeln (*mappings*) definieren die Beziehungen zwischen jeweils zwei aufeinanderfolgenden Ebenen. Die *Transformationsregeln konzeptuelles*/*internes Schema* beschreiben, wie für jedes Objekt des konzeptuellen Schemas die Information aus den physisch abgespeicherten Sätzen, Feldern usw. des internen Schemas erhalten werden kann. Änderungen des internen Schemas wirken sich nur auf die Transformationsregeln, aber nicht auf das konzeptuelle Schema aus. In den *Transformationsregeln externes*/*konzeptuelles Schema* wird angegeben, wie eine spezielle externe Sicht mit dem konzeptuellen Schema zusammenhängt, d.h., welchen Ausschnitt des konzeptuellen Schemas eine bestimmte Benutzergruppe sieht. Daten über die Schemata aller drei Ebenen sowie über die Transformationsregeln (sog. *Meta-Daten*) werden in einem *Systemkatalog* (*system catalog*, *data dictionary*) persistent abgespeichert.

Ein wichtiger Vorteil des 3-Ebenen-Modells ist die *Datenunabhängigkeit* (*data independence*). Dies bedeutet, daß höhere Ebenen des Modells unbeeinflußt von Änderungen auf niedrigeren Ebenen bleiben. Zwei Arten von Datenunabhängigkeit werden unterschieden: *Logische Datenunabhängigkeit* (*logical data independence*) bedeutet, daß sich Änderungen des konzeptuellen Schemas (z.B. Informationen über neue Typen von Entitäten, weitere Informationen über existierende Entitäten) nicht auf externe Schemata (z.B. existierende Anwendungsprogramme) auswirken. *Physische Datenunabhängigkeit* (*physical data independence*) beinhaltet, daß Änderungen des internen Schemas (z.B. Wechsel von einer Zugriffsstruktur zu einer effizienteren, Benutzung anderer Datenstrukturen, Austausch von Algorithmen) keine Auswirkungen auf das konzeptuelle (und externe) Schema haben.

Auf die externe und konzeptuelle Ebene gehen wir im folgenden nicht weiter ein, da im Mittelpunkt unserer Betrachtungen die physische Ebene steht (Bild 1.2). Letztere beschäftigt sich also mit der physischen Implementierung einer Datenbank. Sie bedient sich dabei elementarer Betriebssystemmethoden, um Daten auf externen Speichermedien

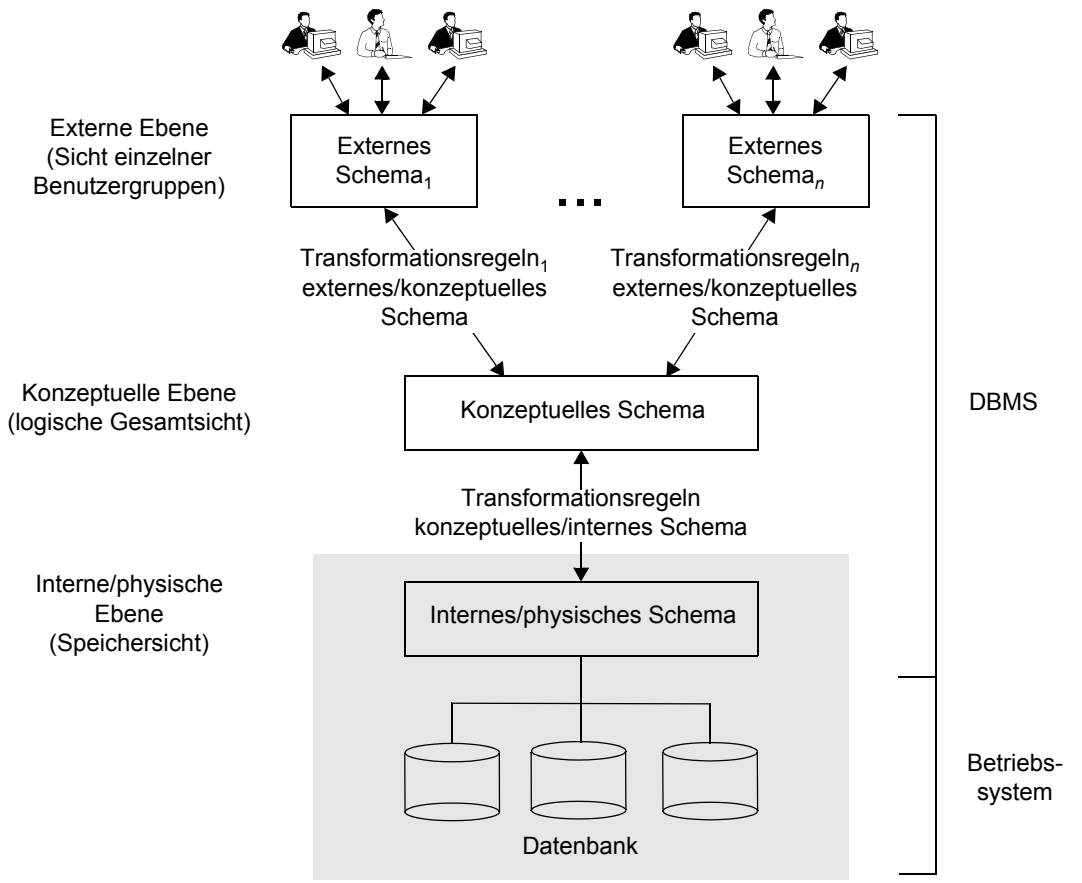


Bild 1.2: Das 3-Ebenen-Modell

auszulagern. Statistische Informationen über die Häufigkeit und Art von Zugriffen auf Entities, über Werteverteilungen von Attributen sowie eine Anzahl weiterer Faktoren beeinflussen die verwendeten Strukturen und Methoden zum Aufbau der Datenbank. Globales Ziel ist es, eine physische Datenorganisation zu entwerfen, so daß die im konzeptuellen Schema beschriebenen Objekte auf die physische Ebene abbildbar sind und die Aufgaben aller Benutzer insgesamt „gut“ und effizient erfüllen. Fragestellungen, die das physische Schema erfassen muß und die später genauer betrachtet werden, sind z.B. die Repräsentation von Attributwerten, der Aufbau gespeicherter Datensätze, Zugriffsmethoden auf Datensätze, zusätzliche Zugriffspfade (Indexe). Vom physischen Schema hängt also wesentlich die Leistungsfähigkeit des gesamten Datenbanksystems ab.

1.4 Softwarearchitektur eines DBMS

Bedingt durch eine Vielzahl von Anforderungen (siehe Abschnitt 1.2) sind heutige DBMS kompliziert aufgebaute Softwareprodukte. DBMS bieten einerseits nach oben hin eine Schnittstelle zum Endbenutzer und andererseits nach unten hin eine Schnittstelle zum Betriebssystem und zur Hardware an (Bild 1.3). In diesem Abschnitt befassen wir uns mit der systematischen und modularen Zerlegung dieser Softwareprodukte in handhabbare Komponenten sowie mit deren Wechselwirkungen und Schnittstellen untereinander, d.h., wir beschreiben die prinzipielle Systemarchitektur von Datenbanksystemen. Wir betrachten also ein *Modell* eines DBMS, das so allgemein gehalten ist, daß es weitgehend der Realität entspricht. Konkrete Architekturen und die dort verwendete Terminologie können sich natürlich in Einzelheiten unterscheiden.

Bild 1.3 zeigt die grundlegenden Komponenten eines typischen DBMS. Die DBMS-Software ist hierarchisch in Schichten organisiert (wir sprechen daher auch von einer *Schichtenarchitektur*), wobei jede Schicht auf der direkt unter ihr liegenden Schicht aufbaut und bestimmte Objekte und Operatoren der niedrigeren Schicht mittels einer von dieser bereitgestellten Schnittstelle zur eigenen Realisierung benutzen bzw. aufrufen kann. Die interne Struktur der Objekte und die Implementierung der Operatoren der niedrigeren Schicht bleibt der darüberliegenden Schicht verborgen.

Die unterste Schicht bildet der *Geräte- und Speicher-Manager* (*disk space manager*, *data manager*), dem die Verwaltung der dem DBMS zur Verfügung stehenden Hardware-Betriebsmittel obliegt und der zwischen externem Speichermedium und dem Hauptspeicher alle physischen Zugriffe auf die Datenbank und auf den Systemkatalog ausführt. Insbesondere ist diese Schicht also für die Bereitstellung und Freigabe von Speicherplatz zur Darstellung von Daten auf externen Speichermedien und für die physische Lokalisierung eines angeforderten Datums verantwortlich. Die von dieser Schicht angebotenen Objekte sind *Dateien* (*files*) und *Blöcke* (*blocks*), die von höheren Schichten allokiert, deallokiert, gelesen und geschrieben werden können. Von Gerätecharakteristika wie Art des Speichermediums, Zylinderanzahl, Spuranzahl, Spurlänge, usw. wird abstrahiert. Die Realisierung dieser Komponente erfolgt entweder durch das darunter liegende Betriebssystem oder durch ein spezialisiertes System, das auf die besonderen Bedürfnisse eines DBMS gezielt eingeht. Ein solch spezialisiertes System kann dann beispielsweise versuchen, Datenteile, auf die meistens als Einheit zugegriffen wird, benachbart (geclustert) auf einer Festplatte abzuspeichern. Dieses Vorgehen minimiert die Suchzeit, weil die gesamte Einheit in einem Lesevorgang erhalten werden kann.

Oberhalb dieser Schicht liegt der *Systempuffer-Manager* (*buffer manager*), der den verfügbaren Hauptspeicherbereich (den „Puffer“) in eine Folge von *Seiten* (*pages*) oder *Rah-*

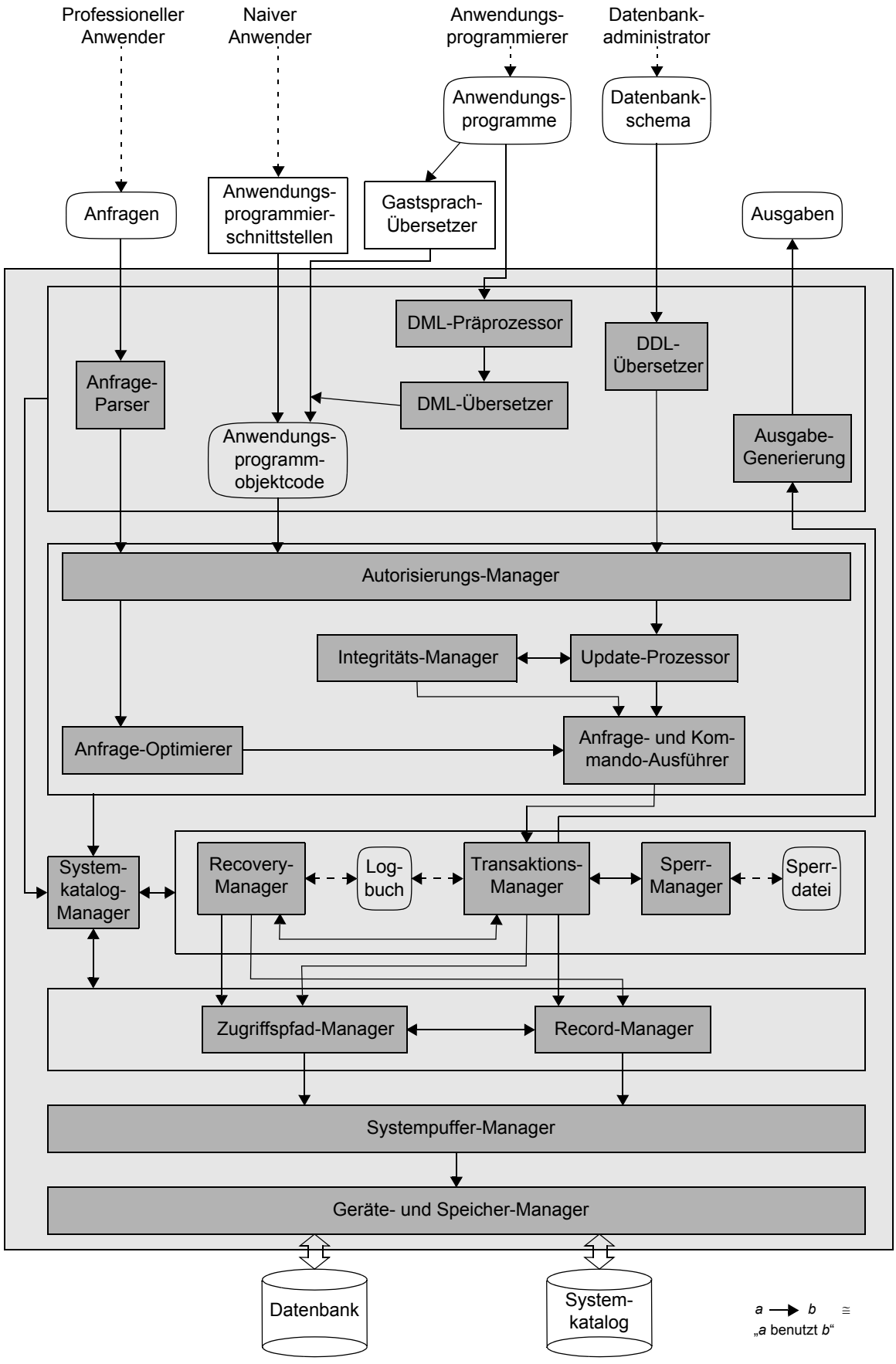


Bild 1.3: Komponenten eines Datenbanksystems

men (*frames*) unterteilt. In der Regel wird eine Seite auf einen Block abgebildet, so daß Lesen und Schreiben einer Seite nur einen Plattenzugriff erfordern. Auf Leseanforderung werden Seiten von einem externen Speichermedium in den Puffer gebracht, auf Schreibanforderung Seiten des Puffers auf einem persistenten Medium gesichert. Nach oben hin stellt dieser Manager *Segmente* (*segments*) mit sichtbaren Seitengrenzen als lineare Adreßräume im Systempuffer zur Verfügung. Dadurch erfolgt die konzeptionelle Trennung von Segment und Datei sowie Seite und Block.

Die nächst höhere Schicht wird durch den Zugriffspfad-Manager und den Record-Manager gebildet. Der *Zugriffspfad-Manager* (*access path manager*) verwaltet eine Anzahl von externen Datenstrukturen zur Abspeicherung von und zum Zugriff auf Kollektionen von Datensätzen. Beispiele solcher Strukturen sind sequentielle Speicherungsformen sowie Indexstrukturen wie der B*-Baum. Operationen beinhalten das Erzeugen und Löschen dieser Speicherstrukturen sowie das effiziente Speichern, Auffinden, Ändern und Löschen von Datensätzen innerhalb dieser Strukturen. Der *Record-Manager* (*record manager*) übernimmt alle Aufgaben, die sich mit der internen Darstellung eines *logischen* Datensatzes (z.B. eines Tupels oder Objekts) beschäftigen. Operationen erlauben den Aufbau und die Zerlegung eines *internen* Datensatzes (d.h., eines Satzes der internen Ebene) und den Zugriff auf dessen Komponenten. Nach unten hin werden interne Datensätze und physische Zugriffspfade auf Seiten von Segmenten abgebildet.

Der Datensicherungs- oder Recovery-Manager, der Transaktions-Manager sowie der Sperr-Manager befinden sich in der nächst höheren Schicht. Im allgemeinen steht eine Datenbank nicht nur einem Benutzer exklusiv, sondern mehreren Benutzern gleichzeitig zur Verfügung. Eine Folge von Befehlen, die nur in ihrer Gesamtheit und ansonsten überhaupt nicht ausgeführt wird, wird als *Transaktion* (*transaction*) bezeichnet. Für einen Mehrbenutzerbetrieb ergibt sich dann das zu lösende Problem der *Synchronisation* (quasi-) parallel ablaufender Transaktionen. Für die Verwaltung von Transaktionen ist der *Transaktions-Manager* (*transaction manager*) zuständig, der nach außen dem Benutzer die Datenbank als ein für ihn exklusiv verfügbares Betriebsmittel erscheinen läßt. Jedoch ist es intern im allgemeinen nicht sinnvoll, alle zu einem bestimmten Zeitpunkt aktuellen Transaktionen sequentiell ablaufen zu lassen bzw. zu verarbeiten. Dies würde die Blokkierung eines kurz andauernden Auftrags durch einen länger andauernden Auftrag zur Folge haben, obwohl beide vielleicht sogar mit disjunkten Teilen der Datenbank arbeiten. Daher werden die einzelnen Transaktionen im allgemeinen zeitlich verzahnt ausgeführt. Diese Verzahnung ist allerdings nicht frei von Problemen und kann zu Konflikten führen. Daher sind spezielle Kontrollstrategien notwendig (*concurrency control*).

Eine Transaktion setzt vor ihrer Ausführung gemäß einem geeigneten *Sperrprotokoll* *Sperren* (*locks*) auf die von ihr exklusiv benötigten Datenbankobjekte und Betriebsmittel

und gibt sie am Ende wieder frei. Der *Sperr-Manager* (*lock manager*) verwaltet Anforderungen auf Sperren und gewährt Sperren auf Datenbankobjekte, wenn sie verfügbar werden. Sperrinformationen werden in einer *Sperr-Datei* (*lock file*) abgespeichert.

Bei der Ausführung einer Transaktion kann es passieren, daß der Transaktions-Manager feststellt, daß die aktuell bearbeitete Transaktion nicht erfolgreich beendet werden kann. In diesem Fall übergibt er sie dem *Datensicherungs-* oder *Recovery-Manager* (*recovery manager*), dessen Aufgabe es ist, die Datenbank in den Zustand vor dem Start der Transaktion zurückzuversetzen. Hierzu muß er alle Änderungen, die die Transaktion an der Datenbank bereits vorgenommen hat, wieder rückgängig machen. Dies geschieht mit Hilfe des *Log-Buchs*, welches unter anderem derartige Änderungen protokolliert. Der Recovery-Manager ist auch dann zuständig, wenn das System einen Software- oder Hardware-Fehler erkennt oder wenn es „zusammenbricht“. Er ist dann für den Wiederanlauf des Datenbanksystems verantwortlich und muß die Datenbank in einen konsistenten Zustand (z.B. mit Hilfe von Sicherungskopien) zurückversetzen; alle „verlorengegangenen“ Transaktionen sind vollständig neu zu starten.

Die nächst höhere Schicht wird von mehreren Komponenten gebildet und umfaßt den Autorisierungs-Manager, den Integritäts-Manager, den Update-Prozessor, den Anfrage-Optimierer und den Anfrage- und Kommando-Ausführer. Benutzeranfragen (*queries*) und Änderungskommandos liegen an der oberen Schnittstelle dieser Schicht in einer „internen“ Form vor (vergleichbar mit dem von einem Compiler von der einen an die nächste Übersetzungsphase übergebenen Zwischencode). Diese Form kann z.B. für relationale Anfragen ein Syntaxbaum (Operatorbaum) sein, dessen Blätter die Operanden und dessen innere Knoten die auf diesen auszuführenden Operationen repräsentieren. Für Anfragen und Kommandos führt der *Autorisierungs-Manager* (*authorization manager*) eine *Zugriffskontrolle* durch. Diese ermittelt, ob der Benutzer auf die in der Anfrage oder im Kommando vorkommenden Daten überhaupt zugreifen darf. Informationen darüber, wer auf welche Daten zugreifen darf, sind in Autorisierungstabellen abgelegt, die Bestandteil des Systemkatalogs sind.

Benutzeranfragen und Änderungskommandos werden danach unterschiedlich gehandhabt. Bei Änderungen (*updates*), die vom *Update-Prozessor* bearbeitet werden, werden mit Hilfe des *Integritäts-Managers* (*integrity manager*) sogenannte *Integritätsbedingungen* (*integrity constraints*) überprüft, die die semantische Korrektheit der Datenbank überwachen. Beispiele solcher Bedingungen sind „Gehälter sind stets größer als 0“ oder „Kundennummern müssen eindeutig sein“. Integritätsbedingungen werden bei der Definition des konzeptuellen Schemas festgelegt und zur Laufzeit vom DBMS automatisch ohne Einwirkung des Benutzers überwacht. Die interne Zwischenform des Kommandos wird hierzu geeignet erweitert. Anfragen werden häufig aus Ausführungssicht unnötig

kompliziert vom Benutzer formuliert. Daher wird eine Anfrage an den *Anfrage-Optimierer* (*query optimizer*) übergeben, der die interne Zwischenform so verändert, daß sie einer effizienter ausführbaren Formulierung entspricht, ohne jedoch das Ergebnis zu verändern. Der Update-Prozessor für Kommandos und der *Anfrage-Ausführer* (*query executor*) für optimierte Anfragen in interner Form erstellen dann Ausführungsprogramme bzw. *Zugriffspläne* (*execution plans*), für die Code erzeugt wird. Hierzu benutzen beide Komponenten Implementierungen von Zugriffsstrukturen und -operatoren des DBMS.

Die oberste Ebene besteht ebenfalls aus mehreren Komponenten und bietet nach oben hin eine mengenorientierte Schnittstelle an. Naive, d.h., ungeübte oder gelegentliche Benutzer verwenden *Anwendungsschnittstellen* (*application interfaces*), die bereits in permanenten *Anwendungsprogrammobjectcode* (*application program object code*) übersetzt sind, zur Kommunikation mit der Datenbank. Der *DDL-Übersetzer* (*DDL compiler*) verarbeitet Schemadefinitionen, die in einer *Datendefinitionssprache* (*data definition language*, *DDL*) spezifiziert sind, und speichert die Beschreibungen der Schemata im Systemkatalog. Geübte, professionelle Benutzer stellen interaktiv und ad hoc *Anfragen* (*queries*) an das System. Die Formulierung von Anfragen erfolgt mittels einer deskriptiven, nicht-prozeduralen Hochsprache, die *Anfragesprache* (*query language*) (z.B. SQL) genannt wird. Charakteristisch für solche Sprachen ist, daß sie das zu lösende Problem, aber nicht den Lösungsweg beschreiben und daß das DBMS stets Mengen von Datenobjekten liefert, die die Anfrage erfüllen. Anfragen werden mit Hilfe des *Anfrage-Parsers* (*query parser*) einer lexikalischen Analyse und einer Syntaxanalyse unterzogen und bei Korrektheit in eine äquivalente, interne, aber effizientere Form, z.B. einem Syntaxbaum, überführt. *Anwendungsprogrammierer* (*application programmer*) interagieren mit dem System durch Kommandos einer *Datenmanipulationssprache* (*data manipulation language*, *DML*), die in ein Programm einer *Gastsprache* (*host language*) eingebettet werden. Die Syntax einer DML ist gewöhnlich sehr verschieden von der Syntax der Gastsprache. Der *DML-Präprozessor* (*DML preprocessor*) filtert die besonders markierten DML-Kommandos aus dem Anwendungsprogramm heraus und führt sie dem *DML-Übersetzer* (*DML compiler*) zu, der sie in Objektcode übersetzt. Die Objektcodes für die DML-Kommandos und das restliche Programm, das durch den Gastsprachübersetzer (*host language compiler*) übersetzt wird, werden danach gebunden.

Der *Systemkatalog-Manager* (*system catalog manager*, *dictionary manager*) steht zu fast allen Komponenten des DBMS in Verbindung und verwaltet den *Systemkatalog* (*system catalog*, *data dictionary*). Der Systemkatalog ist wie die Datenbank auf einem externen Speichermedium abgelegt und enthält *Meta-Daten* (*meta data*) („Daten über Daten“) über die Struktur der Datenbank. Beispiele für Metadaten sind Beschreibungen der Daten, Angaben zu deren Beziehungen untereinander, Beschreibungen der Programme, Konsistenzbedingungen, Angaben über Zugriffsbefugnisse und Speicherdetails.

1.5 Weitere Komponenten eines Datenbanksystems

Zusätzlich zu den oben beschriebenen Softwarekomponenten gibt es eine Reihe von unterschiedlichen *Werkzeugen* (*tools*), die die Entwicklung von Anwendungen auf Datenbanken erleichtern, und *Hilfsprogrammen* (*utilities*), die den Datenbankadministrator bei seiner Arbeit unterstützen. Werkzeuge gibt es nicht nur für den professionellen Anwendungsprogrammierer, sondern auch für den Endbenutzer, dem es ermöglicht wird, Anwendungen zu erzeugen, ohne selbst konventionell programmieren zu müssen. Beispiele sind *Abfragesysteme*, die es auch dem Nicht-Fachmann erlauben, ad hoc Anfragen an das System zu stellen, *Reportgeneratoren*, die formatierte Berichte erzeugen und Berechnungen auf Daten ausführen, Spreadsheets, Werkzeuge zur Erstellung von Geschäftsgraphiken, Werkzeuge für den Datenbankentwurf und CASE-Werkzeuge für den Entwurf von Datenbankanwendungen.

Beispiele für wichtige Hilfsprogramme sind die folgenden. Eine *Import*-Funktion erlaubt es, in einem genau spezifizierten Format vorliegende Dateien (z.B. Textdateien oder sequentielle Dateien) in eine Datenbank einzuladen. Hierdurch können auf recht einfache Weise gleich oder ähnlich strukturierte Massendaten in Objekte der Datenbank überführt werden. Die *Export*-Funktion erzeugt eine Sicherungskopie der Datenbank in einem gewünschten, genau spezifizierten Format. Dies ist für Recovery-Maßnahmen bei einem Systemabsturz von großem Nutzen. Ferner kann diese Funktion zur Archivierung von Daten benutzt werden. Ein Programm zur *Dateireorganisation* ermöglicht es, den physischen Aufbau einer Datei der Datenbank umzustrukturieren, um eine größere Performance und Effizienz zu erreichen. Ein Programm zur *Leistungsüberwachung* kontrolliert die Auslastung des DBMS und liefert statistische Daten zur Entscheidung, ob und welche Maßnahmen ergriffen werden müssen, um die Performance des Systems zu steigern.

Kapitel 2

Externspeicher- und Systempufferverwaltung

Nachdem wir im vorhergehenden Kapitel die aus Implementierungssicht wichtigsten Konzepte von Datenbanksystemen betrachtet haben, erfolgt nun eine detailliertere Erörterung der Interna eines DBMS. Wir beginnen mit den beiden untersten Schichten des Systems (siehe Bild 1.3), nämlich dem Geräte- und Speicher-Manager und dem Systempuffer-Manager. Beide Komponenten bilden zusammen das *Speichersystem*. Der *Geräte- und Speicher-Manager* (*disk space manager*, *data manager*) ist für die Externspeicherverwaltung verantwortlich. Im wesentlichen bestehen seine Aufgaben in der Speicherung von physischen Datenobjekten auf Sekundärspeichern, in der Verwaltung freier Bereiche auf Sekundärspeichern, in der Abschottung aller höheren Systemschichten von Geräteeigenschaften, in der Abbildung von physischen Blöcken auf externe Speicher (Zylinder, Spuren) und in der Kontrolle des Datentransports physischer Datenobjekte zwischen dem Hauptspeicher des Computers und dem externen Speicher. Der *Systempuffer-Manager* (*buffer manager*) verwaltet einen ausgezeichneten, begrenzten Hauptspeicherbereich, den *Systempuffer*, in dem Datenobjekte abgelegt werden. Auf diese Datenobjekte ist dann ein direkter Zugriff möglich. Ferner stellt er für das Zugriffssystem eine Schnittstelle bereit, die eine Unabhängigkeit von den Speicherzuordnungsstrukturen wie die aktuelle Zuordnung von Blöcken zu Dateien, die relative Lage der Blöcke zueinander, usw. sicherstellt. Die Idealvorstellung ist hierbei, daß diese Schnittstelle einen potentiell unendlichen, linearen Adreßraum anbietet, der eine direkte Adressierung der einzelnen Datenobjekte erlaubt.

Im diesem Kapitel beschreiben wir Techniken zur Speicherung großer Mengen strukturierter Daten.[1] In der Regel wird es mehrere Optionen zur Organisation dieser Daten geben, und der Prozeß des *physischen Datenbankentwurfs* beinhaltet eine Auswahl allge-

meiner und effizienter Datenorganisationstechniken, die den Anforderungen der meisten Anwendungen genügen. Abschnitt 2.1 faßt einige Eigenschaften von Primär- und Sekundärspeichern zusammen, soweit sie für ein Verständnis dieses Kapitels notwendig sind. Abschnitt 2.2 erläutert das einer Datenbank zugrundeliegende physische Datenmodell, das im wesentlichen auf dem Dateikonzept beruht. In Abschnitt 2.3 werden Datensatzformate für Datensätze fixer, variabler und sehr großer Länge vorgestellt. Ferner werden die Ausrichtung von Feldwerten sowie Zeiger zur Referenzierung von Datensätzen angesprochen. Abschnitt 2.4 behandelt Seitenformate für Datensätze fixer, variabler und sehr großer Länge. Abschnitt 2.5 beschreibt die Abbildung von Datensätzen in Seiten und erläutert den effizienzsteigernden Aspekt der physischen Clusterung. Abschnitt 2.6 behandelt das Dateikonzept und skizziert die wichtigsten Operationen auf Dateien. Abschnitt 2.7 diskutiert grundlegende Dateiorganisationen wie Haufendateien, sequentielle Dateien und Hash-Dateien und vergleicht ihre Effizienz in bezug auf einige der allgemeinen Dateioperationen. In Abschnitt 2.8 wird auf die wichtige Bedeutung des Systemkatalogs für Datenbanksysteme zur Verwaltung von Meta-Daten eingegangen. Abschnitt 2.9 (in der nächsten Kurseinheit) beschreibt die Systempufferverwaltung. Wichtige Aspekte sind hier das Segment-Konzept mit sichtbaren Seitengrenzen, die Abbildung von Segmenten in Dateien, indirekte Einbringstrategien für Änderungen, die Verwaltung des Systempuffers sowie Unterschiede der Pufferverwaltung in Datenbanksystemen und in Betriebssystemen.

2.1 Primär- und Sekundärspeicher

Eine Sammlung von Datenobjekten, die eine rechnerunterstützte Datenbank bildet, muß physisch auf einem Speichermedium aufbewahrt werden. Bekanntermaßen unterscheidet man (wenigstens) zwei Arten von Speichermedien. Auf *Primärspeichern* wie dem Hauptspeicher und den kleineren aber schnelleren Cache-Speichern kann die Zentraleinheit eines Computers direkt arbeiten. Primärspeicher bieten schnellen Datenzugriff, aber sind von begrenzter Speicherkapazität und insbesondere *flüchtige Speicher* (*volatile storage*). Letztere Eigenschaft besagt, daß Daten nicht persistent im Speicher gehalten werden können. *Sekundärspeicher* oder externe Speichermedien wie Festplatten, optische Platten, Magnettrommeln und Magnetbänder haben gewöhnlich eine größere Kapazität, niedrigere Kosten, aber dafür auch langsameren Datenzugriff als Primärspeicher. Sie sind *nichtflüchtige Speicher* (*non-volatile storage*), deren Daten allerdings nicht direkt von einer

1. Spezielle Zugriffsmechanismen für diese Daten (sogenannte *Indexstrukturen*) werden wir im nächsten Kapitel behandeln. Die Übergänge zwischen Speicherungsstrukturen und Zugriffsmethoden sind allerdings fließend.

Zentraleinheit verarbeitet werden können. Diese müssen erst in den Hauptspeicher zur Verarbeitung kopiert werden. Wegen der Flüchtigkeit dieses Speichertyps werden die Daten nach ihrer Änderung wieder auf das externe Medium zurückgeschrieben. Alle physischen Datenobjekte eines Datenbanksystems werden folglich auf persistenten Sekundärspeichern aufbewahrt.

Kennzeichnend für Sekundärspeicher ist eine Vielzahl unterschiedlicher und von der technologischen Entwicklung abhängiger Geräteeigenschaften wie Speicherkapazität, Zugriffsgeschwindigkeit, Schreib-/Lesetechnik, usw. Auf eine Diskussion der verschiedenen Arten von Sekundärspeichern und ihren Eigenschaften verzichten wir hier. Auf einem Sekundärspeicher wie z.B. einer Festplatte werden Daten in Einheiten gespeichert, die man *Blöcke* (*blocks*) nennt. Ein Block ist eine zusammenhängende Folge von Bytes und ist die Einheit, in der Daten auf die Platte geschrieben oder von Platte gelesen werden. Ein Block umfaßt gewöhnlich ein Vielfaches von 512 Bytes und ist typischerweise zwischen 1 und 8 KB groß. Blöcke werden in konzentrischen Ringen, *Spuren* (*tracks*) genannt, angeordnet. Jede Spur ist in Abschnitte, den *Sektoren* (*sectors*), unterteilt. Mit Hilfe eines Schreib-/Lesekopfes ist ein direkter Zugriff auf einen Block möglich. Während der direkte Zugriff zu jeder gewünschten Position im Hauptspeicher annähernd die gleichen Kosten verursacht, ist die Bestimmung der Kosten für einen direkten Zugriff zu einer Position auf der Platte komplizierter und von drei Zeiten abhängig. Die *Zugriffszeit* (*access time*) ergibt sich dann aus der Summe dieser drei Zeiten. Die *Suchzeit* (*seek time*) ist die Zeit, die für die Positionierung des Lese-/Schreibkopfs auf die richtige Spur, auf der sich ein gewünschter Block befindet, benötigt wird. Die *Latenzzeit* (*rotational delay time*, *latency time*) ist die Wartezeit, die für die Rotation des Lese-/Schreibkopfs auf den gewünschten Block gebraucht wird. Die *Übertragungszeit* (*transfer time*) ist die Zeit, die tatsächlich für das Lesen oder Schreiben der Daten im Block nach Positionierung des Lese-/Schreibkopfs aufgewendet werden muß. Die Zugriffszeit für die Übertragung von Blöcken von und zum externen Speicher ist der wesentliche Kostenfaktor für Datenbankoperationen.

2.2 Das physische Datenmodell

Zur Beschreibung des physischen Schemas wird ein *physisches Datenmodell* benutzt, das alle Implementierungseinzelheiten in bezug auf den Aufbau der Daten, die Datenspeicherung und die Zugriffspfade modelliert. Grundsätzlich läßt sich der gesamte, von einer Datenbank beanspruchte physische Speicherbereich als eine Einheit realisieren und verwalten. Praktische Gründe befürworten allerdings schon auf der Speicherebene eine Zerlegung der Datenbank in disjunkte Teilbereiche. Diese Teilbereiche werden als *Dateien*

(*files*) bezeichnet. Dateien bieten als Einheiten der Externspeicherverwaltung für den Datenbankaufbau folgende Vorteile:

- Typische Datenbankanwendungen benötigen zu einem Zeitpunkt nur einen kleinen Teil der Datenbank. Daher müssen nur tatsächlich benötigte Dateien für einen direkten Zugriff aktiviert werden.
- Da Dateien dynamische Strukturen sind, kann eine Datenbank wachsen oder auch schrumpfen. Der Speicherplatz für temporär benötigte Dateien kann anschließend wieder freigegeben werden.
- Durch Zuordnung von Dateien auf unterschiedlich schnelle Speichermedien können die Zugriffsanforderungen spezieller Anwendungen unterstützt werden.
- Zur Adressierung der Objekte innerhalb einer Datei genügen kürzere Adreßlängen.
- Logisch zusammengehörende, d.h., gleich oder ähnlich strukturierte Daten können in einer Datei abgespeichert werden.

Eine Datei ist logisch als eine Folge von *Datensätzen* (*records*) und physisch als eine Folge von *Blöcken* (*blocks*) organisiert. Auf einen Block können ein oder mehrere Datensätze abgebildet werden. Ein Datensatz beinhaltet eine Sammlung von Datenwerten, die als Aussagen über Entitäten, ihre Attribute und ihre Beziehungen interpretiert werden können. Dateien gelten als Basiselement von *Betriebssystemen* (*operating systems*), so daß wir das Vorhandensein eines zugrundeliegenden *Dateisystems* (*file system*) annehmen dürfen.

2.3 Datensatzformate

Daten werden gewöhnlich in Datensätzen gespeichert. Jeder *Datensatz* (*record*) enthält eine Sammlung von zueinander in Beziehung stehenden *Werten* (*values*, *items*). Jeder Wert besteht aus einem oder mehreren Bytes und entspricht einem bestimmten *Feld* (*field*) des Datensatzes. Datensätze beschreiben in der Regel Entitäten, deren Attribute und deren Beziehungen. Beispielsweise repräsentiert ein Datensatz über einen Angestellten eine Angestellten-Entität; jeder Wert eines Feldes beschreibt ein Attribut oder eine Beziehung dieses Angestellten, wie den Namen, das Geburtsdatum, das Gehalt oder den Vorgesetzten. In relationalen Datenbanken besitzen Datensätze, die Tupel einer Relation repräsentieren, ein Feld für jedes Attribut der Relation; das einem Attribut zugeordnete Feld hat den Datentyp, der dem Attribut zugewiesen ist. Eine Sammlung von Feldnamen und eine Zuordnung von Datentypen zu Feldnamen legen ein *Datensatzformat* (*record*

format) fest. Der einem Feld zugewiesene Datentyp spezifiziert den Typ der Werte, die ein Feld annehmen kann. Alle Datensätze eines gegebenen Datensatzformats haben die gleiche Anzahl von Feldern, und entsprechende Felder haben den gleichen Datentyp, den gleichen Feldnamen und die gleiche intuitive Bedeutung. Informationen über Datensatzformate werden im Systemkatalog abgespeichert.

Der Datentyp eines Feldes ist gewöhnlich einer der von Programmiersprachen bekannten Standarddatentypen. Diese umfassen numerische Datentypen (Integer, Long Integer, Real, Long Real), String-Datentypen von fixer oder variabler Länge sowie Boolsche Datentypen. Die Byterepräsentation dieser Datentypen ist vom verwendeten Computersystem abhängig. Ein Integer-Wert kann z.B. 4 Bytes, ein Long Integer-Wert 8 Bytes, ein Real-Wert 4 Bytes, ein Long Real-Wert 8 Bytes, ein Boolscher Wert ein Byte und ein String-Wert der fixen Länge k eine Anzahl von k Bytes erfordern.

Während Blöcke von fixer Größe (z.B. 4 KB) sind, die durch die physischen Eigenschaften der Festplatte und durch das Betriebssystem vorgegeben ist, können die Größen von Datensätzen gleicher und verschiedener Dateien variieren. So sind z.B. in einer relationalen Datenbank die Tupel verschiedener Relationen in der Regel von unterschiedlicher Größe. Das Ablegen von Datensätzen in Dateien kann im wesentlichen auf drei Arten erfolgen. Die am leichtesten zu implementierende Alternative ist, Datensätze mit nur einer bestimmten Länge in einer Datei abzuspeichern. Eine andere Alternative strukturiert Dateien in einer Art und Weise, daß auch Datensätze unterschiedlicher Länge auf einer Seite der Datei verwaltet werden können. Wachsen Datensätze in einem Ausmaß, daß sie auf keine Seite passen, benötigt man Strategien, um das Problem sehr großer Objekte zu lösen. Unabhängig von der Länge eines Datensatzes ist es häufig notwendig, die Feldwerte in einem Datensatz auszurichten, was zu einem höheren Speicherplatzverbrauch führt. Ferner kann ein Problem entstehen, wenn es Zeiger auf Datensätze gibt. Datensätze sind nämlich dann unter Umständen nicht mehr frei von einer zur anderen Seite verschiebbar.

2.3.1 Datensätze fixer Länge

Hat jeder Datensatz in einer Datei genau die gleiche Größe in Bytes, spricht man von *Datensätzen fixer Länge* (*fixed-length records*). Die *Felder* eines gegebenen Datensatzes, die alle von fixer Länge sind, können aufeinanderfolgend gespeichert werden. Ist die Basisadresse, d.h., die Startposition, eines Datensatzes gegeben, kann die Adresse eines bestimmten Feldes mittels der Summe der Längen der vorhergehenden Felder berechnet werden. Die jedem Feld zugeordnete Summe wird *Offset* dieses Feldes genannt. Daten-

satz- und Feldinformationen sind im Systemkatalog gespeichert. Die Datensatzorganisation wird in Bild 2.1 gezeigt.

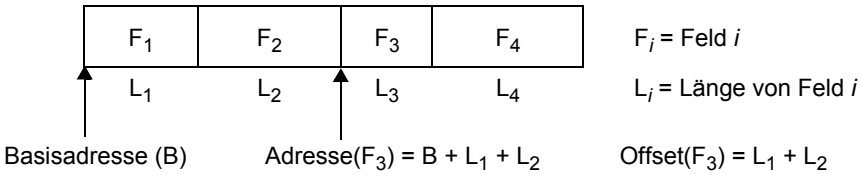


Bild 2.1: Organisation von Datensätzen mit Feldern fixer Länge

Datensätze fixer Länge sind relativ einfach zu verwalten und erlauben den Einsatz spezieller, schneller Suchverfahren. Andererseits müssen bei festen Satzlängen die Sätze selbst und somit deren Felder so groß gemacht werden, daß alle potentiell abzuspeichernden Daten und Werte Platz finden. Dies führt zu Speicherplatzverschwendung und zu ungünstigeren Zugriffszeiten, da der zu durchsuchende Speicherbereich bzw. die Zahl der zu übertragenden Blöcke zum Aufsuchen einer bestimmten Satzmenge größer wird.

2.3.2 Datensätze variabler Länge

Da jeder Datensatz einer Datei die gleiche, feste Anzahl von Feldern hat, kann ein *Datensatz variabler Länge* (*variable-length record*) nur dadurch entstehen, daß einige seiner Felder von variabler Länge sind. Der Name eines Angestellten kann z.B. als ein String variabler Länge definiert sein anstelle eines Strings fixer Länge, der so viele Buchstaben enthält wie der längste vorkommende Namenwert. Einige Felder können auch optionale Werte enthalten, d.h., nur für manche aber nicht für alle Datensätze einer Datei existieren Feldwerte.

Eine mögliche Organisation ist die aufeinanderfolgende und durch Separatoren getrennte Speicherung von Feldern. *Separatoren* sind dabei spezielle Symbole, die in den Daten selbst nicht vorkommen. Ein spezielles *Terminatorsymbol* markiert das Ende des Datensatzes. Diese Organisation erfordert allerdings einen Durchlauf (Scan) des Datensatzes, um ein gewünschtes Feld zu finden (Bild 2.2). Anstelle von Separatoren kann jedes Feld variabler Länge auch mit einem Zähler beginnen, der angibt, wie viele Bytes ein Feldwert beansprucht. Auch hier kostet es Zeit, um auf die Felder jenseits des ersten Feldes variabler Länge zuzugreifen, weil wir den Offset eines Feldes nur berechnen können, wenn wir alle vorherigen Felder besuchen, um deren Längen zu bestimmen.

Eine andere Alternative besteht darin, am Anfang des Datensatzes Speicherplatz für ein Verzeichnis von Integer-*Offsets* oder *Zeigern* zu reservieren (Bild 2.2). Die *i*-te Integer-

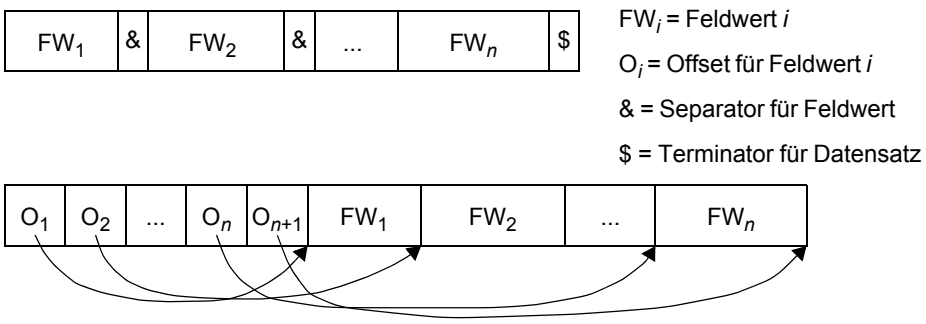


Bild 2.2: Alternative Datensatzorganisationen für Felder variabler Länge

zahl in diesem Verzeichnis ist dann die Startadresse des *i*-ten Feldwertes relativ zum Beginn des Datensatzes. Auch für das Ende eines Datensatzes muß ein Zeiger gespeichert werden, um erkennen zu können, wo der letzte Feldwert endet. Diese Alternative ist in der Regel die bessere. Kosten entstehen nur durch das Offset-Verzeichnis; dafür erhalten wir direkten Feldzugriff. Probleme treten insbesondere bei Änderungen auf. Bei einer Änderung kann ein Feld wachsen, was eine „Verschiebung“ aller nachfolgenden Felder erforderlich macht, um Platz für die Änderung zu schaffen. Außerdem kann passieren, daß ein modifizierter Datensatz nicht mehr auf den ihm auf einer Seite zugewiesenen Platz paßt und auf eine andere Seite bewegt werden muß. Falls Identifikatoren für Datensätze eine Seitennummer enthalten, entsteht ein Problem. Es muß auf dieser Seite ein Zeiger zum neuen Ort des Datensatzes zurückgelassen werden.

2.3.3 Datensätze sehr großer Länge

Ein weiteres Problem von Datensätzen variabler Länge besteht darin, daß ein Datensatz in einem solchen Ausmaße wächst, daß er nicht länger auf eine Seite paßt. Dies kann bei Standarddatenbanken infolge sehr langer Zeichenfolgen geschehen. Solche *Datensätze sehr großer Länge* ((*binary*) *large objects* (*blobs*)) sind aber insbesondere charakteristisch für Nicht-Standard-Datenbanken. In Geo-Datenbanken z.B. werden komplexe geometrische Objekte wie Polylinien, Polygone und Polyeder in Feldern verwaltet, die „beliebig“ groß werden können.

Zur Speicherung kann solch ein Datensatz in kleinere Datensätze unterteilt werden. Jeder kleinere Datensatz erhält zudem einen Zeiger auf den nächsten kleineren Datensatz, so daß eine Verkettung dieser Datensätze untereinander erreicht wird. Der Datensatz insgesamt erstreckt sich also über Seitengrenzen hinaus. Eine andere Möglichkeit ist, nur die wichtigsten Informationen über den zu speichernden Wert im Feld eines Datensatzes abzulegen und die genaue Repräsentation in einer Folge von speziellen Datenseiten aus-

zulagern. Gespeichert wird hier also zusätzlich eine *Seitenreferenz* auf die erste Seite der Objektdarstellung oder sogar die gesamte *Seitenreferenzfolge*. Dies ist sinnvoll, da häufig für Anfragen der Informationsteil ausreicht und die gesamte Objektdarstellung nicht benötigt wird. Im Falle eines Polygons könnte im Informationsteil das minimale, achsenparallele Rechteck abgespeichert werden, das das Polygon enthält. Weitere Daten könnten die Anzahl der Kanten, den Flächeninhalt und den Umfang enthalten. Alle diese Informationen sind dann in konstanter Zeit erhältlich. Die Beschreibung der geometrischen Struktur des Polygons könnte dann auf ausgelagerten Seiten gespeichert sein.

2.3.4 Ausrichtung von Feldwerten

Ein allgemeines Problem, das Datensätze aller Längen betrifft, ist die Forderung nach der *Ausrichtung* (*alignment*) von Feldwerten. Rechnerarchitekturen und Programmiersprachen verlangen häufig, daß Werte eines Datentyps in einem Datensatz ausgerichtet werden. Dies bedeutet, daß die absolute Anfangsadresse eines Feldwertes durch die Anzahl der Bytes zur Darstellung des zugehörigen Typs ganzzahlig teilbar sein muß. Nehmen wir an, daß ein Integer-Wert 4 Bytes, ein Real-Wert 4 Bytes, ein Long Real-Wert 8 Bytes, ein Boolscher Wert ein Byte und ein Zeichen ein Byte zu ihrer Darstellung benötigen, so muß die absolute Anfangsadresse eines Feldwertes eines solchen Typs durch 4, 4, 8, 1 bzw. 1 ganzzahlig teilbar sein. D.h., Einschränkungen gibt es für Integer-, Real- und Long Real-Werte. Die absolute Anfangsadresse des Datensatzes muß durch die größte Anzahl an Bytes teilbar sein, die einer seiner Feldwerte zur Darstellung benötigt.

Da Feldwerte aufeinanderfolgend im Datensatz angeordnet werden, hat dies gegebenenfalls ungenutzten Speicherplatz zur Folge. Enthält ein Datensatz z.B. aufeinanderfolgend einen Boolschen Wert, einen Real-Wert und ein Zeichen, so werden für die Darstellung des Boolschen-Werts 1+3, des Real-Werts 4+0 und des Zeichens 1+3 Bytes benötigt, insgesamt also 12 Bytes. Der zweite Summand bezeichnet jeweils den ungenutzten Speicherplatz (hier also insgesamt 6 Bytes). Wird die Reihenfolge verändert, so daß dem Boolschen Wert das Zeichen und dann der Real-Wert folgt, so werden für die Darstellung des Boolschen-Werts 1+0, des Zeichens 1+2 und des Real-Werts 4+0 Bytes verwendet. Insgesamt braucht die Darstellung eines solchen Datensatzes 8 Bytes, wovon 2 Bytes ungenutzt bleiben. In beiden Fällen muß die Anfangsadresse des Datensatzes durch 4 teilbar sein.

Selbsttestaufgabe 3. Gegeben sei das folgende Datensatzformat fixer Länge für Studenten:

Student(*Name*: *String*[24]; *Matrikelnr*: *String*[9]; *Geschlecht*: *Boolean*;
Semesteranzahl: *Integer*; *Straße*: *String*[14]; *Nr*: *Integer*;
Plz: *Integer*; *Ort*: *String*[18])

Geben Sie die Datensatzgröße in Bytes unter Beachtung der Ausrichtung von Feldwerten an. Gibt es eine speicherplatzeffizientere, d.h., kompaktere, Alternative, wenn die Reihenfolge der Feldwerte vertauscht werden darf?

2.3.5 Zeiger

Um einen Datensatz zu referenzieren, ist häufig ein *Zeiger* (*pointer*) auf ihn ausreichend. Aufgrund der Verschiedenheit der Anforderungen zur Abspeicherung von Datensätzen kann der Aufbau von Zeigern variieren. Die offensichtlichste Art eines Zeigers ist die physische Adresse des Datensatzes in einem virtuellen Speicher oder in dem Adreßraum der Festplatte. Aus ihr läßt sich die Seite, die gelesen werden muß, sehr leicht berechnen. Diese Adressierungsart hat den Vorteil, daß der Zugriff auf einen gesuchten Datensatz sehr schnell ist. Sie ist aber dennoch nicht wünschenswert, da sie das Verschieben eines Datensatzes innerhalb einer Seite oder innerhalb einer Gruppe von Seiten nicht ermöglicht. Bei der Verschiebung eines Datensatzes müßten nämlich alle Zeiger auf diesen Datensatz gefunden und verändert werden. Man spricht hier auch von *physischen Zeigern*.

Häufig wird daher ein Zeiger als ein Paar (s, p) beschrieben, wobei s die Nummer der Seite ist, auf der der Datensatz zu finden ist, und p auf eine Indexposition in einem Verzeichnis am Seitenanfang verweist. Der Eintrag an dieser Indexposition gibt die relative Position des Datensatzes in der Seite an. Muß der Datensatz innerhalb der Seite verschoben werden, muß lediglich der Eintrag an der Indexposition geändert werden. Alle Zeiger (s, p) bleiben unverändert. Man spricht hier von *seitenbezogenen Zeigern* (siehe Bild 2.5).

Gegenüber Verschiebungen im Speicher völlig stabile Zeiger erhält man, wenn Zeiger logisch realisiert werden. Dem Datensatz wird eine logische oder symbolische Adresse zugeordnet, die nichts über die Abspeicherung des referenzierten Datensatzes aussagt. Der Datensatz kann also beliebig im Adreßraum der Datei bewegt werden, ohne daß Zeiger verändert werden müssen. Realisiert wird diese Zeigerform beispielsweise über indirekte Adressierung. Der Zeiger verweist auf eine Indexposition einer Zuordnungstabelle im Speicher, deren Eintrag die Position des Datensatzes im Speicher angibt. Bei der Verschiebung eines Datensatzes muß lediglich der zugehörige Eintrag in der Zuordnungstabelle geändert werden. Alle Zeiger bleiben unverändert. In diesem Fall spricht man von *logischen* oder *symbolischen Zeigern* (siehe Bild 2.3). Als großer Nachteil erweist sich, daß jeder Zugriff auf einen Datensatz einen zusätzlichen Zugriff auf die Zuordnungsta-

belle kostet. Ferner kann die Tabelle sehr groß werden, so daß sie oft nicht vollständig im Hauptspeicher gehalten werden kann.

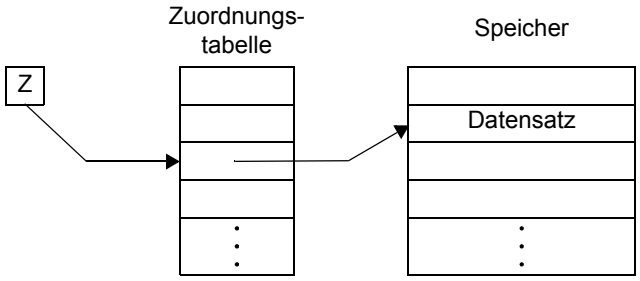


Bild 2.3: Realisierung logischer Zeiger

Wenn auf Datensätze Zeiger unbekannter Herkunft zeigen, so sprechen wir von *fixierten Datensätzen* (*pinned records*), ansonsten von *unfixierten Datensätzen* (*unpinned records*). Wenn Datensätze unfixiert sind, können sie innerhalb und zwischen verschiedenen Seiten ohne nachteilige Folgen verschoben werden, falls dies im Rahmen der gewählten Speicherstruktur Sinn macht. Wenn jedoch die Datensätze fixiert sind, können sie überhaupt nicht bewegt werden, falls Zeiger absolute Adressen sind, nur innerhalb der Seite, falls eine seitenbezogene Zeigervariante benutzt wird, und allgemein im Adreßraum der Datei, falls logische Zeiger Verwendung finden.

Wird ein fixierter Datensatz gelöscht, ergibt sich das Problem, daß noch Zeiger auf ihn vorhanden sein können, die nun semantisch sinnlos sind. Wird anstelle des gelöschten Datensatzes ein neuer Datensatz eingefügt, verweisen Zeiger plötzlich auf falsche Datensätze. Man spricht hier auch von „baumelnden“ Zeigern (*dangling pointers*, *dangling references*). Um diese Art von Zeigern zu vermeiden, muß für jeden Datensatz eine *Löschmarkierung* verwaltet werden, die gesetzt wird, falls der Datensatz gelöscht wird. Der freigewordene Speicherplatz kann dann allerdings nicht mehr für weitere Einfügungen benutzt werden, aber der gelöschte Datensatz kann z.B. bei einem Durchlauf über die Seite als gelöscht erkannt und ignoriert werden.

2.4 Seitenformate

In der gleichen Weise, wie wir Felder in Datensätzen lokalisieren müssen, müssen wir Datensätze innerhalb einer Seite (bzw. extern in einem Block) auffinden können. Wir betrachten daher, wie Datensätze auf einer Seite angeordnet werden können. Dazu stellen wir uns eine Seite als eine Sammlung von *Slots* (Aufnahmebereichen, *slots*) vor, wobei jeder Slot genau einen Datensatz enthält. Ein *Datensatzidentifikator* (*record identifier*

(*RID*)), kurz *DID* genannt, der als Zeiger auf einen fixierten Datensatz dient, besteht typischerweise aus einem Paar (s, n), wobei s die Nummer der Seite ist, auf der der Datensatz zu finden ist, und n eine Zahl ist, die angibt, wo auf der Seite der Datensatz abgespeichert ist. Der Parameter n kann verschieden interpretiert werden, z.B. als relative Byteadresse auf der Seite, als Nummer eines Slots oder als Index eines im *Seitenkopf* (*page header*) zusätzlich eingerichteten *Verzeichnisses* (*directory*). Solch ein Identifikator hängt nicht vom Inhalt des zugehörigen Datensatzes ab. Mit der Verwaltung von Datensätzen in Seiten geht auch die *Freispeicherverwaltung* innerhalb einer Seite einher. Zudem müssen ähnlich wie Feldwerte auch Slots und somit die in ihnen enthaltenen Datensätze ausgerichtet werden. Hierauf gehen wir im weiteren nicht mehr näher ein. Wir betrachten nun einige Alternativen, um für Datensätze Slots auf einer Seite einzurichten.

2.4.1 Seitenformate für Datensätze fixer Länge

Falls alle Datensätze von gleicher Länge sind, sind auch alle Slots gleich groß und können aufeinanderfolgend innerhalb der Seite angeordnet werden. Es können soviele Datensätze in einer Seite aufgenommen werden, wie Slots auf eine Seite passen. Abzuziehen ist allerdings der Speicherplatz für Seiteninformationen in speziellen Feldern des Seitenkopfes wie z.B. für Zeiger auf Vorgänger- und Nachfolgerseiten in einer Folge von Seiten oder für Verzeichnisse. Speicherplatz für spezielle Felder muß in jeder Seite an bestimmten Stellen reserviert sein.

Die erste Alternative zur Anordnung einer Menge von N Datensätzen fixer Länge ist, sie in den ersten N Slots abzulegen (Bild 2.4). Wenn ein Datensatz gelöscht wird, kann der letzte Datensatz auf der Seite in den freigewordenen Slot bewegt werden. Dies verursacht jedoch Probleme, wenn der zu bewegende Datensatz fixiert ist und nun die Slotnummer geändert werden muß. Die Möglichkeit, alle auf den gelöschten Datensatz nachfolgenden Datensätze um einen Slot vorrücken zu lassen, erweist sich als sehr ineffizient und noch problemhaltiger. Somit ist diese „gepackte“ Organisation unflexibel, obwohl sie es erlaubt, den i-ten Datensatz durch eine einfache Offsetberechnung zu ermitteln.

Eine zweite Alternative besteht darin, auf jeder Seite Löschungen von Datensätzen und somit Informationen über freie Slots mittels einer Verzeichnisstruktur, die eine Bitmap darstellt, zu verwalten (Bild 2.4). Das Auffinden des i-ten Datensatzes auf einer Seite erfordert dann einen Durchlauf durch das Verzeichnis, aber Datensätze müssen nicht verschoben werden. Auf eine Verzeichnisstruktur kann verzichtet werden, wenn es ein spezielles Feld gibt, das einen Zeiger auf den ersten Slot enthält, dessen Löschmarkierung gesetzt ist. Der Slot enthält dann einen Zeiger auf den nächsten freien Slot, so daß eine Verkettung von freien Slots erreicht wird.

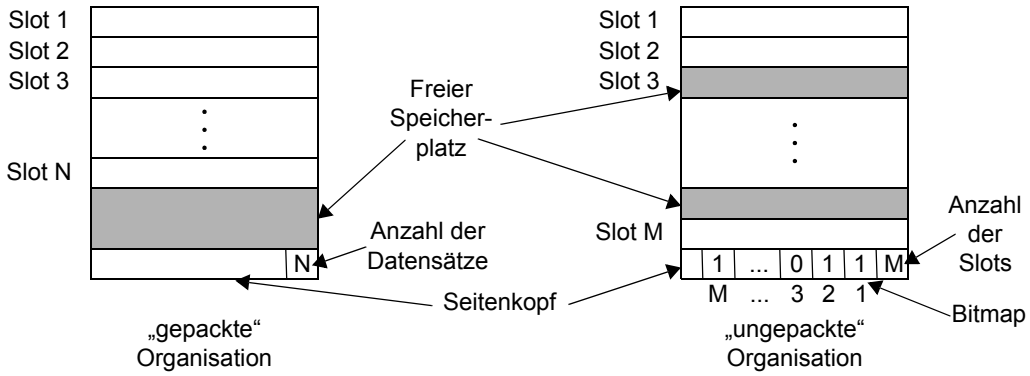


Bild 2.4: Alternative Seitenorganisationen für Datensätze fixer Länge

2.4.2 Seitenformate für Datensätze variabler Länge

Auch wenn die Datensatzlängen variabel sind, können wir Datensätze aufeinanderfolgend auf einer Seite anordnen. Löschungen von Datensätzen können wir allerdings nicht mehr durch die oben beschriebenen Bitmaps oder durch verkettete Listen handhaben, weil Slots nicht mehr auf einfache Weise wiederbenutzbar sind. Wenn ein neuer Datensatz eingefügt werden soll, muß zunächst ein leerer Slot „der richtigen Länge“ gefunden werden. Ist der Slot zu groß, wird Speicherplatz verschwendet; ist der Slot zu klein, kann er nicht benutzt werden. In der Regel wird also ein Slot, der einen Datensatz aufnehmen kann, nicht vollständig ausgefüllt werden, sondern am Ende des Slots entsteht ein relativ kleiner ungenutzter Speicherplatzbereich. Um diese ungenutzten Speicherplatzbereiche zwischen Datensätzen zu vermeiden – wir sprechen hier auch von *Fragmentierung* –, müssen Datensätze auf einer Seite verschoben und komprimiert werden können. Dadurch erhalten wir einen zusammenhängenden freien Speicherplatzbereich.

Wenn die Datensätze einer Seite unfixiert sind, externe Zeiger also keine Rolle spielen, kann die „gepackte“ Repräsentation für Datensätze fixer Länge angepaßt werden. Entweder verwendet man ein spezielles Terminatorsymbol zur Kennzeichnung des Datensatzendes oder aber eine Längenangabe am Datensatzanfang.

Für den allgemeinen Fall ist bei der Satzadressierung eine Indirektion vorzusehen, die Verschiebungen von fixierten Datensätzen innerhalb einer Seite ohne nachteilige Auswirkungen erlaubt, aber nach Möglichkeit keine weiteren Zugriffskosten mit sich bringt. Die flexibelste Organisation für Datensätze variabler Länge wird durch das *Datensatzidentifikator-Konzept* (*DID-Konzept*) beschrieben (Bild 2.5). In relationalen Datenbanksystemen ist es als *Tupelidentifikator-Konzept* (*TID-Konzept*) bekannt. Jedem Datensatz wird ein eindeutiger, stabiler DID zugeordnet, der aus einer Seitennummer und einem Index in

ein seiteninternes Verzeichnis besteht. Der durch den Index i beschriebene Verzeichniseintrag enthält die relative Position des Slots i und somit des i-ten Datensatzes (d.h., einen Zeiger hierauf) innerhalb einer Seite. Die Längeninformation eines Datensatzes wird entweder ebenfalls im Verzeichniseintrag oder aber am Anfang der Datensatzrepräsentation abgelegt (L_i in Bild 2.5). Datensätze, die Wachstums- oder Schrumpfungsprozessen unterworfen werden, können nun innerhalb der Seite verschoben werden, ohne daß der DID als extern sichtbare Zugriffsadresse modifiziert werden muß. Wird ein Datensatz gelöscht, so wird dies im entsprechenden Indexeintrag durch eine Löschmarkierung registriert.

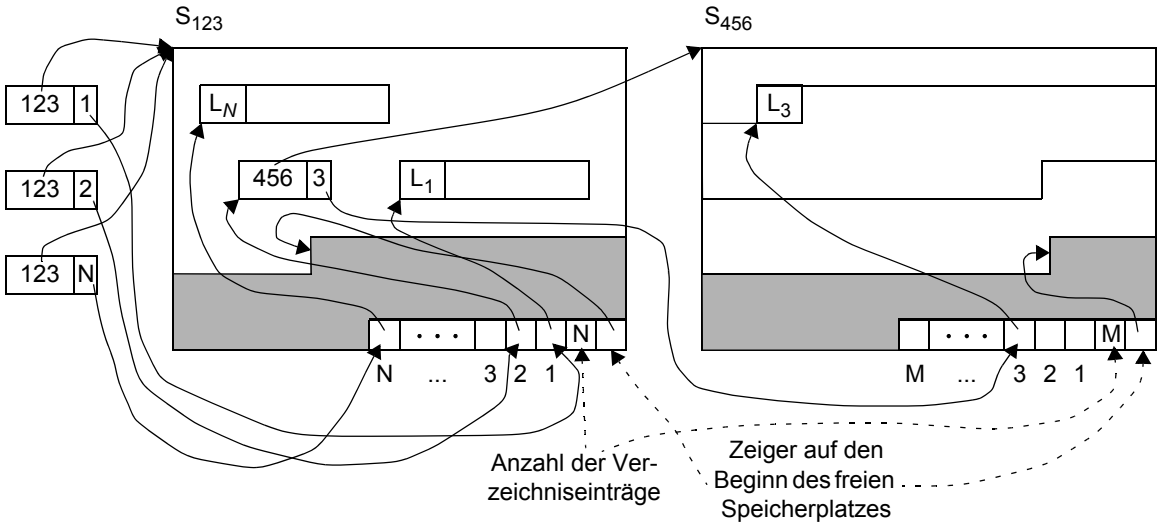


Bild 2.5: Seitenorganisation für Datensätze variabler Länge

Da eine Seite nicht in vorgegebene Slots eingeteilt werden kann, muß eine andere Art der Freispeicherverwaltung angewendet werden. Eine Methode ist, einen Zeiger auf den Anfang des freien Speicherplatzbereichs der Seite im Seitenkopf zu verwalten. Ist ein Datensatz zu groß, um vom aktuell verfügbaren, freien Speicherplatzbereich aufgenommen zu werden, wird die Seite komprimiert, d.h., Speicherplatzlücken zwischen Datensätzen bis auf diejenigen, die durch Ausrichtung von Datensätzen entstehen, werden beseitigt. Hierdurch wird erreicht, daß alle Datensätze lückenlos aufeinanderfolgend angeordnet sind und der maximal verfügbare freie Speicherplatz daran anschließt. Wenn ein Datensatz trotz Komprimierung der Seite nicht in den noch verfügbaren Speicherplatzbereich paßt, muß er von seiner „Hausseite“ auf eine „Überlaufseite“ ausgelagert werden. Der zugehörige DID kann stabil gehalten werden, indem in der Hausseite anstelle des Datensatzes ein „Stellvertreter-DID“ gespeichert wird, der genauso wie ein normaler DID funktioniert und auf den Satz in der neu zugewiesenen Überlaufseite zeigt. Da ein Überlaufsatz nicht weiter überlaufen darf, wird die Überlaufkette auf die Länge 1

beschränkt. Falls ein Überlaufsatz seine ihm zugewiesene Seite wieder verlassen muß, wird zunächst versucht, ihn wieder auf seiner Hausseite zu plazieren. Ist nicht genügend Speicherplatz vorhanden, erfolgt der neue Überlauf wieder von der Hausseite aus. Daher ist jeder Datensatz mit maximal zwei Seitenzugriffen aufzufinden.

Nach dem Löschen eines Datensatzes kann der Speicherplatz für den Verzeichniseintrag an der entsprechenden Indexposition nicht einfach gelöscht und das seiteninterne Verzeichnis komprimiert werden, weil die Indizes des Verzeichnisses dazu benutzt werden, um Datensätze zu identifizieren. Löschen wir einen Eintrag und schieben wir den restlichen Teil des Verzeichnisses heran, so vermindern sich die Indizes für die nachfolgenden Slots im Verzeichnis, so daß DIDs auf falsche Slots und somit auf falsche Datensätze zeigen. Das Verzeichnis kann nur dann komprimiert werden, wenn der Datensatz im letzten Slot entfernt wird. Wenn ein Datensatz eingefügt wird, ist das Verzeichnis nach einem Eintrag abzusuchen, der nicht auf einen Datensatz zeigt. Der dem zugehörigen Index zugeordnete Slot nimmt dann den neuen Datensatz auf.

Das DID-Konzept kann auch für Datensätze fixer Länge eingesetzt werden, falls diese häufig verschoben werden müssen. Dies kann z.B. notwendig sein, wenn Datensätze in sortierter Reihenfolge gehalten werden müssen. Haben alle Datensätze gleiche Länge, kann auch die Längeninformation weggelassen und im Systemkatalog abgespeichert werden. In einigen speziellen Situationen (z.B. die internen Seiten eines B-Baums in Abschnitt) sind Datensatzidentifikatoren nicht von Interesse. In diesem Fall kann ein Verzeichnis nach dem Löschen eines Datensatzes komprimiert werden; die Anzahl der Einträge im Verzeichnis ist dann gleich der Anzahl der Datensätze auf der Seite. Auch das Sortieren von Datensätzen wird vereinfacht. Nicht die Datensätze, sondern die Verzeichniseinträge werden sortiert und somit bewegt.

2.4.3 Seitenformate für Datensätze sehr großer Länge

Ist ein Datensatz von solcher Länge, daß er auf keine (auch nicht auf eine neue) Seite paßt, so gibt es im wesentlichen nur die beiden bereits in Abschnitt 2.3.3 besprochenen Strategien. Die erste Strategie, die gelegentlich auch zur Organisation von Datensätzen variabler Länge eingesetzt wird, sieht vor, daß ein Datensatz zunächst in eine Folge von mittels Zeigern verketteten, kleineren Datensätzen zerlegt wird. Diese Folge „überspannt“ oder erstreckt sich dann über eine Anzahl von Seiten (*spanned records*). Als Konsequenz ergeben sich entsprechend viele Seitenzugriffe und hoher Zusatzaufwand bei der Integritätsüberwachung. Falls möglich, z.B. bei Datensätzen fixer Länge, wird daher versucht, ein Überschreiten von Seitengrenzen zu vermeiden (*unspanned records*).

Die andere Strategie speichert für die Feldwerte, die für die große Gesamtlänge eines Datensatzes verantwortlich sind, nur die wichtigsten Informationen ab und lagert die genauen Repräsentationen der Feldwerte jeweils auf eine Folge von speziellen Datenseiten aus. Der Datensatz enthält dann in den Feldwertdarstellungen jeweils zusätzlich eine Seitenreferenz auf die erste Seite der Objektdarstellung oder sogar die gesamte Seitenreferenzfolge.

2.5 Abbildung von Datensätzen in Seiten

In Abschnitt 2.4 haben wir betrachtet, wie Datensätze technisch auf Seiten plaziert werden können und wie die zugehörigen Seitenformate aussehen. Bei einer Folge von Datensätzen gleichen Formats, die zusammen eine Datei ausmachen, ist nun zu entscheiden, welche dieser Datensätze welchen Seiten zugeordnet werden. Da Daten zwischen externem Speichermedium und Hauptspeicher in Einheiten von Blöcken übertragen werden, ist es sinnvoll, Datensätze auf Seiten in einer Weise abzubilden, daß eine einzelne Seite miteinander in Beziehung stehende Datensätze enthält. Diese Art der „Datensatzgruppierung“, die logisch verwandte (und daher häufig zusammen benutzte) Datensätze physisch nahe beieinanderliegend auf einem externen Speichermedium anordnet, wird auch *Clusterung* (*clustering*) genannt. Folgendes Beispiel belegt, daß die physische Clusterung von Daten ein extrem wichtiger Faktor in bezug auf Effizienz ist und daß es bedeutsam ist, die Clusterung von Datensätzen in Seiten auf Seiten in Dateien fortzusetzen. Dies wird sich später insbesondere bei der Behandlung von Indexstrukturen zeigen.

Nehmen wir an, daß d_1 derjenige Datensatz ist, auf den zuletzt zugegriffen wurde, und daß d_2 der nächste angeforderte Datensatz ist. Sei ferner d_1 auf Seite s_1 und d_2 auf Seite s_2 gespeichert. Dann können sich folgende Situationen ergeben:

1. Sind s_1 und s_2 identisch, dann erfordert der Zugriff auf d_2 keinen zusätzlichen externen Seitenzugriff, weil sich die gewünschte Seite s_2 bereits im Systempuffer im Hauptspeicher befindet.

2. Sind s_1 und s_2 verschieden und liegen beide Seiten physisch nahe beieinander oder sind sogar physisch benachbart, dann erfordert der Zugriff auf d_2 (vorausgesetzt, s_2 ist nicht schon vorher in den Hauptspeicher geladen worden) einen weiteren externen Seitenzugriff. Aber die Suchzeit hierfür wird sehr klein sein, weil sich der Lese-/Schreibkopf bereits nahe der gewünschten Position befindet.

3. Sind s_1 und s_2 verschieden und liegen beide Seiten physisch weiter voneinander entfernt, so erfordert der Zugriff auf d_2 (vorausgesetzt, s_2 ist nicht schon vorher in den

Hauptspeicher geladen worden) einen weiteren externen Seitenzugriff und die volle Zugriffszeit.

Wir sind bisher immer davon ausgegangen, daß nur Datensätze des gleichen Formats auf einer Seite abgelegt werden. Obiges Beispiel läßt aber offen, ob d_1 und d_2 zur gleichen oder zu verschiedenen Dateien gehören (und somit also verschiedene Formate haben) und ob also über eine oder mehr als eine Datei geclustert wird. Betrachten wir eine Datenbank, die Informationen über Lieferanten und Warensendungen enthält. Wenn der sequentielle Zugriff auf alle Lieferanten in einer numerierten Reihenfolge eine häufige Anwendungsanforderung ist, dann sollten die Lieferanten-Datensätze so geclustert werden, daß der erste solche Datensatz physisch nahe beim zweiten Datensatz angeordnet wird, der zweite Datensatz physisch nahe beim dritten Datensatz, usw. Geclustert wird hier also innerhalb einer Datei (*intra-file clustering*). Ist die Anwendungsanforderung aber dergestalt, daß häufig auf Lieferanten zusammen mit den von ihnen gelieferten Waren zugegriffen werden muß, so empfiehlt sich, Lieferanten- und Warensendungen-Datensätze verzahnt miteinander zu speichern. Informationen über Warensendungen des ersten Lieferanten werden zusammen mit dem ersten Lieferanten-Datensatz abgespeichert, Informationen über Warensendungen des zweiten Lieferanten zusammen mit dem zweiten Lieferanten-Datensatz, usw. Das Clustering erstreckt sich hierbei also über mehr als eine Datei (*inter-file clustering*).

2.6 Dateien

Eine *Datei* (*file*) ist logisch gesehen eine Folge von Datensätzen gleichen Formats. Diese Definition unterscheidet sich etwas von dem gewöhnlichen Verständnis einer Datei als ein Strom von Zeichen oder Elementen anderer Typen und der Vorstellung, daß solch ein Strom nur vom Anfang bis zum Ende durchlaufen werden kann. In der Datenbankpraxis kann auf eine Datei aber auf vielerlei Art und Weise zugegriffen werden, und die Datensätze sind häufig auch nicht als ein einzelner Strom angeordnet.

Physisch ist eine Datei als eine Folge von direkt adressierbaren Blöcken organisiert. Prinzipiell sind für die physischen Blöcke der Datei variable Längen denkbar. Konstante Blocklängen sollten aber bevorzugt werden, da sie neben der einfachen Adressierung die flexible Ausnutzung des gesamten verfügbaren, externen Speicherplatzes ohne Fragmentierungsprobleme gestatten, ferner die Systempufferverwaltung (siehe Abschnitt 2.9) vereinfachen und eine saubere, geräteunabhängige Schnittstelle anbieten. Meta-Daten über eine Datei (z.B. Länge, Adresse des ersten und letzten Blocks) stehen gewöhnlich auf der ersten Seite im *Dateikopf* (*file header*).

Operationen auf Dateien werden gewöhnlich in *Such-* (*retrieval*) und *Änderungs-* (*update*) Operationen unterschieden. Suchoperationen ändern nicht die Daten in einer Datei, sondern lokalisieren bestimmte Datensätze, so daß deren Feldwerte überprüft und verarbeitet werden können. Änderungsoperationen verändern die Datei durch Einfügen oder Löschen von Datensätzen oder durch Modifizieren von Feldwerten. In allen Fällen müssen ein oder mehrere Datensätze zum Auffinden, Löschen oder Ändern mittels eines *Selektionskriteriums* ausgewählt werden. Dieses Kriterium legt Bedingungen fest, die gewünschte Datensätze erfüllen müssen. Die im folgenden kurz beschriebenen Operationen zur Lokalisierung und zum Zugriff auf Datensätze sind repräsentativ und variieren von System zu System. Hierbei wird der Datensatz, auf den zuletzt zugegriffen worden ist, als *aktueller* Datensatz bezeichnet.

- *Datensatz auffinden*. Diese Operation sucht nach dem ersten Datensatz, der ein Selektionskriterium erfüllt. Die den Datensatz enthaltende Seite wird in den Systempuffer geladen (falls sie dort nicht bereits vorhanden ist), und der Datensatz wird im Puffer lokalisiert und zum aktuellen Datensatz.

- *Datensatz lesen*. Diese Operation kopiert den aktuellen Datensatz aus dem Systempuffer in eine Programmvariable oder in den Arbeitsbereich eines Anwendungsprogramms. Die Anweisung kann auch ein Vorwärtsschreiten zum nächsten Datensatz zur Folge haben.

- *Nächsten Datensatz finden*. Diese Operation sucht den nächsten Datensatz, der das Selektionskriterium erfüllt. Die den Datensatz enthaltende Seite wird in den Systempuffer geladen (falls sie dort nicht bereits vorhanden ist), und der Datensatz wird im Puffer lokalisiert und zum aktuellen Datensatz.

- *Datensatz löschen*. Diese Operation löscht den aktuellen Datensatz aus einer Seite im Systempuffer und aktualisiert (eventuell) die externe Datei.

- *Datensatz verändern*. Diese Operation verändert einige Feldwerte des aktuellen Datensatzes aus einer Seite im Systempuffer und und aktualisiert (eventuell) die externe Datei.

- *Datensatz einfügen*. Diese Operation fügt einen neuen Datensatz in die Datei ein, indem sie zunächst die Seite feststellt, in die eingefügt werden muß, und dann die Seite in den Systempuffer lädt (falls sie dort nicht bereits vorhanden ist). Danach fügt sie den Datensatz in die Seite ein und aktualisiert (eventuell) die externe Datei.

Andere Operationen werden benötigt, um existierende Dateien für den Zugriff zu öffnen, nach Benutzung zu schließen, um neue, leere Dateien zu erzeugen und um existierende Dateien zu löschen.

Mengen von logischen Objekten der konzeptuellen Ebene werden nun auf Dateien abgebildet. In relationalen DBMS wird üblicherweise jede Relation in einer Datei gespeichert. Jedes Tupel wird als Datensatz in der Datei abgelegt, und Attributwerte des Tupels entsprechen Feldwerten des Datensatzes. In objektorientierten DBMS ist dies ähnlich. Objektklassen werden in Dateien, Objekte in Datensätzen und Attributwerte in Feldwerten abgelegt. Ausnahmen ergeben sich zum Beispiel bei der Clusterung von Relationen oder Objektklassen; mehrere Relationen oder Objektklassen können hier in einer Datei zusammen gespeichert sein. Informationen über Relationennamen, Objektklassennamen, deren Schemata, zugeordnete Dateien, Dateistrukturen, Attributnamen, Attributtypen, usw. sind systemweit mittels des Systemkatalogs (Abschnitt 2.8) erhältlich.

2.7 Grundlegende Dateiorganisationen

Wir stellen nun drei grundlegende Dateiorganisationen, nämlich Dateien mit ungeordneten Datensätzen (Haufendateien), Dateien mit bezüglich eines Feldes geordneten Datensätzen (sortierte Dateien) und Dateien mit bezüglich eines Feldes verstreuten Datensätzen (Hash-Dateien), vor und vergleichen die Kosten einiger in Abschnitt 2.6 angesprochener, einfacher Operationen. Die betrachteten Operationen sind (1) der Durchlauf (Scan) aller Datensätze in einer Datei, (2) die Suche nach einem Datensatz, der eine bestimmte Gleichheitsbedingung erfüllt (z.B. „finde den Angestellten-Datensatz mit der Personalnummer 728“), (3) die Suche nach allen Datensätzen, die eine Bereichsbedingung erfüllen (z.B. „finde alle Angestellten-Datensätze, deren Name lexikographisch zwischen Meier und Schmitt liegt“), (4) das Einfügen eines Datensatzes und (5) das Löschen eines Datensatzes.

2.7.1 Kostenmodell

Um Kosten überhaupt vergleichen zu können, geben wir zunächst ein sehr vereinfachtes Kostenmodell mit entsprechenden Notationen und Annahmen an. Sei b die Anzahl der Datenseiten und r die Anzahl der Datensätze pro Seite. Man bezeichnet r auch als *Blokkungsfaktor* (*blocking factor*). Die Durchschnittszeit zum Lesen oder Schreiben einer Seite (eines Blocks) von bzw. auf ein externes Speichermedium sei d, und die Durchschnittszeit, um einen Datensatz zu verarbeiten, d.h., einen Feldwert mit einer Selektionskonstante zu vergleichen, sei c. Eine *Hash-Funktion* bildet einen Datensatz in einen Zahlenbereich ab und wird zur Organisation einer Hash-Datei verwendet, um anhand eines gegebenen Datensatzes die zugehörige Seite zu berechnen. Die Zeit, um die Hash-Funktion für einen Datensatz auszurechnen, sei h.

CPU-Kosten, Übertragungskosten usw. werden hier vernachläßigt, da sie gegenüber den wesentlich kostenintensiveren, externen (Platten-) Zugriffen nicht ins Gewicht fallen. Wir zählen daher in diesem sehr vereinfachten Modell nur die Anzahl der gelesenen oder geschriebenen externen Seiten. Ferner ignorieren wir auch den *geblockten Zugriff* (*blocked access*). D.h., Seiten werden normalerweise nicht einzeln gelesen, sondern mit einem Zugriff wird auch eine ganze Folge darauffolgender Seiten in den Systempuffer geladen. Die Kosten sind also gleich der Suche nach der ersten Seite der Seitenfolge und der Übertragung aller Seiten der Seitenfolge in den Hauptspeicher. Der Grund für geblockten Zugriff liegt im *Lokalitätsprinzip*. Man hofft, daß der nächste Seitenzugriff in unmittelbarer Nähe der letzten Seite geschehen wird. Die neue Seite befindet sich aber dann schon im Systempuffer, so daß kein externer Zugriff mehr erforderlich ist. Wir werden jeden Seitenzugriff einzeln zählen, was in der Regel teurer ist, da wir für jede Seite eine zusätzliche Suchzeit haben.

2.7.2 Haufendateien

Diese einfachste Dateiorganisation plaziert Datensätze in eine Datei in der ungeordneten, chronologischen Reihenfolge, wie sie eingefügt werden. Neue Datensätze werden am Ende der Datei eingefügt. Solch eine Organisation wird *Haufendatei* (*heap file*, *pile file*) genannt. Datensätze werden ungeordnet abgespeichert, wenn die Art ihrer Nutzung in der Zukunft unklar ist. Auch wird diese Organisationsform in Verbindung mit zusätzlichen Zugriffspfaden verwendet.

Für jede Haufendatei müssen wir die zugeordneten Seiten verwalten, um Durchläufe zu unterstützen, sowie die Seiten, die freien Speicherplatz enthalten, um das Einfügen effizient durchzuführen. Doppelt verkettete Listen von Seiten sowie Verzeichnisse von Seiten stellen zwei mögliche Realisierungsalternativen dar. Sie benutzen Zeiger (Seitenidentifikatoren, -nummern) zur Adressierung von Seiten. Bezüglich der ersten Alternative wird vom DBMS eine Tabelle mit (Haufendateiname, Adresse der ersten Seite)-Einträgen eingerichtet. Die erste Seite einer Haufendatei wird *Kopfseite* (*header page*) genannt. Eine wichtige Aufgabe besteht darin, Informationen über freien Speicherplatz, der durch Löschen von Datensätzen entstanden ist, aufzubewahren. Hierzu sind der freie Speicherplatz innerhalb einer Seite (bereits in Abschnitt 2.4 behandelt) und die Seiten mit freiem Speicherplatz zu verwalten. Letztere Aufgabe kann mittels einer *doppelt verketteten Liste* für die Seiten mit freiem Speicherplatz und einer doppelt verketteten Liste für die vollen Seiten erreicht werden. Beide Listen enthalten zusammen alle Seiten der Haufendatei. Nachteilig ist, daß, falls die Datensätze von variabler Länge sind, die Seiten in der Regel noch einige freie Bytes haben und somit fast alle der freien Liste zugeordnet werden. Um

einen Datensatz einzufügen, müssen mehrere Seiten der freien Liste gelesen und überprüft werden, bevor man eine Seite mit genügend freiem Speicherplatz findet.

Verzeichnisse von Seiten stellen eine Alternative dar. Das DBMS muß sich hier merken, wo sich die erste Verzeichnisseite der Haufendatei befindet. Das Verzeichnis selbst ist ebenfalls eine Menge von Seiten und kann z.B. als verkettete Liste organisiert sein. Jeder Verzeichniseintrag zeigt auf eine Seite in der Haufendatei. Genauso wie die Haufendatei wächst oder schrumpft, wächst oder schrumpft auch die Anzahl der Einträge. Freispeicherverwaltung kann mittels eines Biteintrags durchgeführt werden, der anzeigt, ob die jeweilige Seite freien Speicherplatz enthält, oder aber mittels eines Zählers pro Eintrag, der die freie Byteanzahl pro Seite angibt. Ein Vergleich der Länge eines Datensatzes variabler Länge mit dem Zählerwert eines Eintrags ergibt dann, ob der Datensatz auf die Seite, worauf der Eintrag zeigt, eingefügt werden kann.

Betrachten wir nun die Kosten für die obengenannten Operationen.

Durchlauf: Die Kosten sind $b(d + rc)$ Zeit, weil wir auf jede Seite zugreifen müssen, was Zeit d pro Seite kostet. Für jede Seite werden r Datensätze mit Zeit c pro Datensatz verarbeitet.

Suche mit Gleichheitsbedingung: Angenommen, wir wissen, daß es genau einen Datensatz gibt, der die gewünschte Bedingung erfüllt, d.h., die Bedingung ist bezüglich eines Schlüsselfeldes formuliert, und daß eine gleichmäßige Verteilung der Werte im Schlüsselfeld vorliegt. Im Durchschnitt müssen wir dann die Hälfte der Datei durchlaufen. Für jede gelesene Seite müssen wir alle Datensätze auf Übereinstimmung mit dem gewünschten Datensatz überprüfen. Die Kosten sind $0{,}5 \cdot b(d + rc)$ Zeit. Falls die Suche nicht bezüglich eines Schlüsselfeldes stattfindet, müssen wir die gesamte Datei durchlaufen, weil Datensätze mit dem gesuchten Feldwert über die gesamte Datei verstreut sein können und wir nicht wissen, wie viele dies sind. Dies gilt auch, wenn die Bedingung zwar bezüglich eines Schlüsselfeldes formuliert ist, aber der gewünschte Datensatz gar nicht in der Datei vorhanden ist.

Suche mit Bereichsbedingung: Die gesamte Datei muß durchlaufen werden, da Datensätze, die die Bereichsbedingung erfüllen, überall in der Datei auftreten können und wir nicht wissen, wie viele es sind. Die Kosten sind $b(d + rc)$ Zeit.

Einfügen eines Datensatzes: Angenommen, daß Datensätze immer am Ende der Datei eingefügt werden. Dann muß die letzte Seite der Datei, deren Adresse im Dateikopf steht, geladen, der Datensatz eingefügt und die Seite wieder zurückgeschrieben werden. Die Kosten hierfür sind $2d + c$. Gibt es eine Freispeicherverwaltung und wird versucht, Datensätze auf die erste passende Seite einzufügen, erhöht dies die Kosten. Diese zusätzlichen

Kosten hängen von der internen Struktur für die Haufendatei ab. Wir ignorieren diesen Kostenaspekt in unserer Analyse.

Löschen eines Datensatzes: Der zu löschende Datensatz ist aufzufinden, von der Seite zu löschen, und die geänderte Seite muß zurückgeschrieben werden. Die Kosten sind die Suchkosten plus $c + d$. Falls der zu löschende Datensatz über einen DID identifiziert wird, sind die Suchkosten vernachlässigbar, weil der Seitenidentifikator aus dem DID erhalten werden kann. Falls der zu löschende Datensatz durch eine Gleichheitsbedingung über einigen Feldern gegeben ist, sind zwei Fälle zu unterscheiden. Sind die Felder Schlüsselfelder und genau ein Datensatz erfüllt die Bedingung, so sind die Suchkosten gleich den Kosten für eine Suche mit Gleichheitsbedingung. Ansonsten müssen wir alle Datensätze der Datei betrachten.

2.7.3 Sequentielle Dateien

Die Datensätze einer Datei können physisch auch in sortierter Reihenfolge angeordnet werden. Die Sortierung erfolgt auf der Basis der Werte eines der Datensatzfelder. Dieses Feld wird auch *Ordnungsfeld* (*ordering field*) oder *Sortierfeld* genannt, und man spricht dementsprechend auch von *sortierten Dateien* oder *sequentiellen Dateien* (*sequential files*). Ist das Sortierfeld auch ein *Schlüsselfeld* (*key field*) der Datei, d.h., ein Feld, dessen Werte in der Datei eindeutig sind und somit einen Datensatz eindeutig identifizieren, dann spricht man auch von einem *Sortierschlüssel* (*ordering key*) für diese Datei. Für die Operationen ergeben sich folgende Kosten:

Durchlauf: Die Kosten betragen $b(d + rc)$ Zeit, weil auf jede Seite zugegriffen werden muß. Ein Scan ist also genauso teuer wie bei Haufendateien. Der Unterschied ist, daß hier gemäß der Sortierreihenfolge auf die Datensätze zugegriffen wird.

Suche mit Gleichheitsbedingung: Betrachten wir zunächst den Fall, daß alle Datensätze von fixer Länge und auf jeder Seite in gepackter Form organisiert sind (siehe Abschnitt). Ferner sollen die Datensätze in der Datei in aufeinanderfolgenden Blöcken abgelegt sein. Die Seite, die den oder die gewünschten Datensätze (falls sie existieren) enthält, kann mittels binärer Suche in $\log_2 b$ Schritten gefunden werden. Ist die Seite bekannt, kann wiederum bei Sortierung nach einem Sortierschlüssel mittels binärer Suche der gewünschte Datensatz in $c \cdot \log_2 r$ Zeit lokalisiert werden. Die Gesamtkosten sind daher $d \cdot \log_2 b + c \cdot \log_2 r$ Zeit, was eine wesentliche Verbesserung gegenüber Haufendateien darstellt. Gibt es mehrere Datensätze, die die Bedingung erfüllen (d.h., das Sortierfeld ist kein Schlüsselfeld), so liegen diese physisch benachbart zueinander. Die Gesamtkosten ergeben sich aus der Suche nach dem ersten solchen Datensatz ($d \cdot \log_2 b + c \cdot \log_2 r$ Zeit) plus den

Kosten für die nachfolgenden, die Bedingung erfüllenden Datensätze. Gibt es keine die Bedingung erfüllenden Datensätze, dann ergeben sich die Kosten aus der Suche nach der Seite, die den Datensatz bei Vorhandensein enthalten hätte.

Eine ähnliche binäre Suchstrategie ist anwendbar, wenn entweder die obige gepackte durch die ungepackte Organisationsform (siehe Abschnitt 2.4.1) ersetzt wird oder aber bei unfixierten Datensätzen variabler Länge (siehe Abschnitt 2.4.2) deren Ordnung durch eine Sortierung der Verzeichniseinträge erreicht wird. Einzelheiten hierzu seien dem Leser überlassen. Ansonsten bringt die Sortierung gegenüber Haufendateien keine Vorteile.

Selbsttestaufgabe 4. Geben Sie einen Algorithmus an, der für sequentielle Dateien mit Datensätzen fixer Länge die Suche mit Gleichheitsbedingung mittels binärer Suche realisiert.

Suche mit Bereichsbedingung: Es gilt ähnliches wie bei der Suche mit Gleichheitsbedingung. Ist die Suche mit Bereichsbedingung über einem Sortierschlüssel definiert, so wird der erste Datensatz, der die Bedingung erfüllt, lokalisiert. Anschließend werden die Datenseiten sequentiell solange durchlaufen, bis ein Datensatz gefunden wird, der die Bedingung nicht erfüllt. Die Gesamtkosten ergeben sich also aus den Kosten für die Suche nach dem ersten zutreffenden Datensatz plus den Kosten für die nachfolgenden Datensätze, die die Bedingung erfüllen. Befinden sich alle zutreffenden Datensätze auf einer Seite, so sind die Kosten unwesentlich höher als bei der Suche mit Gleichheitsbedingung.

Einfügen eines Datensatzes: Betrachten wir wiederum zunächst den Fall, daß alle Datensätze von fixer Länge sind, auf jeder Seite in gepackter Form organisiert sind und in der Datei in aufeinanderfolgenden Blöcken abgelegt sind. Um einen Datensatz unter Aufrechterhaltung der Sortierreihenfolge einzufügen, müssen wir die richtige Einfügeposition in der Datei finden, den Datensatz auf der gefundenen Seite plazieren und dann alle nachfolgenden Seiten laden und zurückschreiben, um die alten Datensätze um einen Slot zu verschieben. Im Durchschnitt können wir annehmen, daß der neue Datensatz in die Mitte der Datei eingefügt werden muß. Folglich müssen wir die zweite Hälfte der Datei lesen und sie dann nach Einfügen des neuen Datensatzes zurückschreiben. Letzterer Schritt kostet $2 \cdot (0{,}5 \cdot b(d + rc))$ Zeit. Die Gesamtkosten setzen sich also aus den Kosten für die Suche nach der Einfügeposition plus $b(d + rc)$) zusammen. Prinzipiell ist diese Strategie auf alle Seitenformate anwendbar, solange es sich nicht um fixierte Datensätze handelt. In diesem Fall kann die Sortierreihenfolge nicht aufrechterhalten werden.

Löschen eines Datensatzes: Es seien die gleichen Annahmen wie beim Einfügen gegeben. Setzen wir ferner voraus, daß der zu löschende Datensatz über einem Sortierschlüssel definiert ist. Dann müssen wir die richtige Seite finden, den Datensatz von dieser Seite löschen, die geänderte Seite zurückschreiben und alle nachfolgenden Seiten lesen und zurückschreiben, um alle darauffolgenden Datensätze heranzuschieben. Die Kosten belaufen sich auf die Suchkosten plus $b(d + rc)$) Zeit. Falls mehrere Datensätze die Löschbedingung erfüllen, wissen wir, daß diese physisch benachbart liegen, und die Kosten unterscheiden sich unwesentlich vom Löschen eines einzelnen Datensatzes. Kennzeichnen wir Löschungen durch Löschmarkierungen, ergeben sich die Gesamtkosten aus den Suchkosten plus $c + d$. Da freier Speicherplatz nicht zusammengefaßt wird, müssen die nachfolgenden Seiten nicht gelesen und nicht zurückgeschrieben werden.

Selbsttestaufgabe 5. Beim Einfügen und Löschen von Datensätzen in sequentiellen Dateien ist es aufgrund der Vielzahl von notwendigen Datensatzverschiebungen sehr schwierig und sehr zeitaufwendig, die physisch geordnete Struktur aufrechtzuerhalten. Ein Ausweg ist das Erzeugen einer temporären, *ungeordneten* Datei, die *Überlaufdatei* genannt wird. Die sequentielle Datei wird dann *Hauptdatei* genannt. Neue Datensätze werden am Ende der Überlaufdatei anstelle der richtigen Einfügeposition in der Hauptdatei eingefügt. Beschreiben Sie die Vor- und Nachteile dieser Alternative sowie die Auswirkungen auf die betrachteten Operationen.

2.7.4 Hash-Dateien

Hash-Strukturen sind ein Beispiel für das im Datenbankbereich sehr wichtige Konzept des *Indizierens* (*indexing*). Ziel des Indizierens ist es, einen effizienten Zugriff auf Datensätze mit gegebenen Werten eines (eindeutigen) *Suchschlüssels* (*search key*), der aus mehreren Suchfeldern bestehen kann, zu gewährleisten, auch wenn die Datensätze nicht nach ihnen sortiert sind. Eine ganze Reihe von speziellen *Indexstrukturen* (*index structures*) sind entwickelt worden, einschließlich ausgeklügelter Varianten der in diesem Abschnitt besprochenen Hash-Dateiorganisation. Wir werden uns dem Thema der Indexstrukturen im nächsten Kapitel ausführlich widmen. In diesem Abschnitt führen wir das Indizieren als eine allgemeine Technik ein und betrachten als Beispiel eine einfache Hash-Dateiorganisation.

Die grundlegende Idee von *Hash-Dateien* (*hash files*) ist die Verteilung der Datensätze einer Datei in sogenannte *Behälter* (*buckets*). Die Verteilung geschieht in Abhängigkeit vom Wert des Suchschlüssels. Die direkte Zuordnung eines Datensatzes zu einem Behäl-

ter erfolgt mittels einer *Hash-Funktion* (*hash function*) h, die als Argument einen Wert für den Suchschlüssel erhält und diesen Wert auf eine natürliche Zahl aus dem Bereich 0 bis B-1 abbildet, wobei B die Anzahl der dieser Datei zugeordneten Behälter ist. Die Zahl $h(v)$ wird *Behälternummer* (*bucket number*) für den Schlüsselwert v genannt. Auf diese Weise werden alle Datensätze einer Datei über den gesamten, der Datei zugeordneten Speicherbereich verstreut. Man spricht daher auch von *Streuspeicherung*. Beispielsweise kann die Anfrage „Finde den Datensatz für Rainer Meyer“ effizient ausgeführt werden, wenn der Datensatz Bestandteil einer Hash-Datei ist, deren Hash-Funktion über dem Namensfeld definiert ist.

Jeder Behälter enthält eine oder wenige Seiten von Datensätzen, wobei die Seiten eines jeden Behälters als Haufen organisiert sind. Zur Verwaltung der Behälter dient ein *Behälterverzeichnis* (*bucket directory*), das aus einem Array von Zeigern, indiziert von 0 bis B–1, besteht. Der Eintrag für Index i im Behälterverzeichnis zeigt auf die erste Seite für Behälter i. Alle Seiten für Behälter i sind in einer einfach verketteten Liste verbunden, wobei ein Nullzeiger auf der letzten Seite die Liste abschließt. B sollte hinreichend klein gewählt werden, so daß das Behälterverzeichnis in den Hauptspeicher paßt. Beim Einfügen wird ein neuer Datensatz gemäß der Hash-Funktion im entsprechenden Behälter abgespeichert. Gewöhnlich ist nur auf der letzten Seite Platz, so daß alle Seiten bis dorthin durchlaufen werden müssen. Dies kann man sich sparen, wenn das Behälterverzeichnis für jeden Behälter auch einen Zeiger auf die letzte Seite enthält. Falls auf der letzten Seite kein Platz mehr ist, werden gegebenenfalls Überlaufseiten bereitgestellt. In diesem Fall spricht man von einer *statischen Hash-Datei*. Nachteilig ist, daß lange Ketten von Überlaufseiten entstehen können, die alle durchsucht werden müssen. *Dynamische Hash-Dateien* behandeln dieses Problem durch eine variable Anzahl von Behältern.

Für eine Hash-Funktion h gibt es eine Reihe von Möglichkeiten. Es ist wesentlich, daß der Wertebereich von h 0 bis B–1 ist und daß h diese Werte mit möglichst gleicher Wahrscheinlichkeit annimmt. Ein einfaches Beispiel einer geeigneten Hash-Funktion ist $h(v) = v \bmod B$, $v \in \mathbb{N}$. Wir vertiefen dieses Thema an dieser Stelle nicht weiter, da bereits im Bereich von Algorithmen und Datenstrukturen vieles zu Hash-Funktionen gesagt worden ist (siehe Literaturhinweise).

Betrachten wir nun die Operationen. In unserer Analyse nehmen wir an, daß es keine Überlaufseiten gibt.

Durchlauf: In einer Hash-Datei werden Seiten nur zu ungefähr 80% gefüllt, um die Anzahl der Überlaufseiten bei Dateiexpandierung möglichst minimal zu halten. Die Anzahl der Seiten und somit die Kosten für einen Durchlauf aller Datenseiten sind 1,25 mal so hoch wie die entsprechenden Kosten bei einer Haufendatei, d.h., $1{,}25 \cdot b(d + rc)$ Zeit.

Suche mit Gleichheitsbedingung: Die Stärke von Hash-Dateien liegt gerade in dieser Operation. Angenommen, die Bedingung ist auf dem Suchschlüssel der Hash-Datei definiert. Enthält der Suchschlüssel mehr als ein Feld, muß eine Gleichheit auf jedem dieser Suchfelder definiert sein. Die Kosten, um den Behälter, der den Datensatz enthält, herauszufinden, betragen h Zeit. Umfaßt dieser Behälter genau eine Seite (d.h., es gibt keine Überlaufseiten), kostet die Ermittlung des Datensatzes d Zeit. Nehmen wir weiterhin an, daß durchschnittlich die Hälfte der Datensätze dieser Seite durchsucht werden muß, um den Datensatz zu finden, ergeben sich Gesamtkosten von $h + d + 0{,}5 \cdot rc$ Zeit. Dies ist sogar besser als im Falle der sequentiellen Dateien. Falls es mehrere oder keine, die Bedingung erfüllende Datensätze gibt (d.h., das oder die Suchfelder, über denen die Hash-Datei definiert ist, bilden keinen eindeutigen Suchschlüssel), müssen sich diese alle im gleichen Behälter befinden. Unter der Annahme, daß der Behälter aus einer Seite besteht, ergeben sich Gesamtkosten von $h + d + rc$ Zeit. In diesem Fall müssen also alle Datensätze der geladenen Seite durchsucht werden.

Suche mit Bereichsbedingung: Diese Operation wird nicht unterstützt, auch nicht, wenn sich die Bereichsbedingung auf die Suchfelder, über denen die Hash-Datei definiert ist, bezieht. Die gesamte Datei muß durchlaufen werden, so daß die Kosten $1{,}25 \cdot b(d + rc)$ Zeit betragen.

Einfügen eines Datensatzes: Der entsprechende Behälter muß mittels der Hash-Funktion ermittelt werden. Danach wird die zugehörige Seite geladen (keine Überlaufseiten), durch Einfügen des neuen Datensatzes geändert und zuletzt zurückgeschrieben. Alle Datensätze der Seite müssen gelesen werden, um festzustellen, ob sich der einzufügende Datensatz nicht schon auf der Seite befindet. Die Gesamtkosten betragen $h + 2 \cdot d + rc$ Zeit.

Löschen eines Datensatzes: Diese Operation ist analog zum Einfügen. Im Durchschnitt muß aber die Seite nur zur Hälfte durchsucht werden, um den Datensatz zu finden. Die Gesamtkosten sind also $h + 2 \cdot d + 0{,}5 \cdot rc$ Zeit. Erfüllen mehrere Datensätze die Löschbedingung, so muß die gesamte Seite durchsucht werden.

2.7.5 Vergleich der Organisationsformen

Die Beschreibung verschiedener Dateiorganisationen sowie der Kosten der auf ihnen durchführbaren Operationen hat gezeigt, wie wichtig die Wahl einer geeigneten Organisationsform ist. Eine Zusammenfassung der Analyse der drei Dateiorganisationen zeigt Bild 2.6; CPU-Kosten werden der Einfachheit halber vernachläßigt. Eine Haufendatei besitzt eine gute Speichereffizienz und erlaubt schnelles Einfügen und Löschen von

Datensätzen. Schwächen zeigt diese Struktur bei Suchoperationen; bei kompletten Durchläufen ist sie allerdings effizient.

Auch eine sequentielle Datei bietet eine gute Speichereffizienz, ist aber langsam beim Einfügen und Löschen von Datensätzen. Sie ist sehr schnell bei Suchoperationen und ist insbesondere die beste Struktur für Suchoperationen mit einer Bereichsbedingung. Es ist wichtig zu betonen, daß in wirklichen DBMS eine Datei in der Regel niemals sortiert gehalten wird. Der Grund liegt in der Existenz von B-, B^+- und B*-Bäumen (siehe Abschnitt), die ebenfalls alle Vorteile einer sortierten Datei beinhalten und außerdem Einfügen und Löschen von Datensätzen effizient unterstützen.

Eine Hash-Datei ist aufgrund der nur 80%igen Seitenbelegung zwar weniger speichereffizient als eine Haufendatei oder eine sequentielle Datei, aber Einfügen und Löschen sind schnell und Suchen mit Gleichheitsbedingung sogar sehr schnell. Die Struktur bietet allerdings keine Unterstützung für Suchen mit Bereichsbedingung, und komplette Durchläufe sind ein wenig langsamer, weil eine Hash-Datei aufgrund der unvollständigen Seitenbelegung mehr Seiten enthält. Weil B^+-Bäume Suchen mit Bereichsbedingung effizient

Dateiorganisation	Durchlauf	Suche mit Gleichheitsbedingung	Suche mit Bereichsbedingung	Einfügen	Löschen
Haufen	bd	$0{,}5 \cdot bd$	bd	$2 \cdot d$	Suche + d
Sortiert	bd	$d\log_2 b$	$d\log_2 b$ + # Treffer	Suche + bd	Suche + bd
Gestreut	$1{,}25 \cdot bd$	d	$1{,}25 \cdot bd$	$2 \cdot d$	$2 \cdot d$

Bild 2.6: Vergleich der Seitenzugriffskosten für verschiedene Dateiorganisationen

unterstützen und bei Suchen mit Gleichheitsbedingung den Hash-Dateien nur wenig nachstehen, unterstützen viele DBMS keine Hash-Dateien. Diese finden aber dann Verwendung, wenn sehr häufig Suchen mit Gleichheitsbedingung durchgeführt werden.

Die wohl wichtigste Erkenntnis ist, daß keine Dateiorganisationsform in allen Situationen überlegen ist. Eine Haufendatei ist die beste Struktur bei kompletten Dateidurchläufen. Eine Hash-Datei ist die beste Struktur bei häufig auftretenden Suchen mit Gleichheitsbedingung. Eine sequentielle Datei ist die beste Struktur bei Suchen mit Bereichsbedingung. Die Wahl einer geeigneten Dateiorganisation hängt also wesentlich von ihrem vorwiegenden Gebrauch ab.

2.8 Systemkatalog

Eine grundlegende Eigenschaft eines Datenbanksystems ist, daß es nicht nur die Datenbank mit den darin befindlichen Daten, sondern auch eine vollständige Definition der Struktur der Datenbank enthält. Diese Definition ist im *Systemkatalog* (*system catalog*, *data dictionary*) abgespeichert, der vom *Systemkatalog-Manager* (*system catalog manager*) (Bild 1.3) verwaltet und von fast allen Komponenten der DBMS Software sowie Datenbankanwendern genutzt wird. Die im Katalog gespeicherte Information bezeichnet man als *Meta-Daten* („Daten über Daten“, *meta data*). Da die DBMS Software universell einsetzbar und nicht für spezielle Datenbankanwendungen geschrieben worden ist, muß sie sich auf die Kataloginformation beziehen, um Angaben über die Struktur der Datenbank und der in ihr befindlichen Objekte zu erhalten.

Den Inhalt und die Realisierung von Systemkatalogen beschreiben wir anhand relationaler Datenbanksysteme. Systemkataloge enthalten zum einen systemweite Informationen wie die Größe des Systempuffers und die Seitengröße und zum anderen Beschreibungen der verschiedenen Schemata (extern, konzeptuell, intern) und Transformationsregeln. Zu den Schemainformationen gehören zum einen alle Angaben über Definition und Struktur der Daten wie Namen und zulässige Wertebereiche, logische Beziehungen, Integritäts- und Zugriffsregeln usw. und zum anderen alle Angaben über Speicherung, Codierung und Auffinden der Daten wie Adreß- und Längenangaben, Feldtypen, Zugriffspfade und physische Plazierung in der Datenbank.

Für jede Relation wird der Relationenname, der Dateiname oder Dateiidentifikator, die gewählte Dateiorganisation (z.B. Haufendatei), Attributname und Typ jedes seiner Attribute, der Indexname eines jeden auf ihm definierten Index und Integritätsbedingungen (z.B. Schlüsselbedingungen) festgehalten. Für jeden Index wird der Indexname oder ein Identifikator, die Struktur des Index (z.B. B^+-Baum) sowie die Suchschlüsselattribute abgespeichert. Für jede Sicht wird der Sichtenname und die durch eine Anfrage gegebene Definition abgelegt.

Zusätzlich gibt es gewöhnlich statistische und beschreibende Informationen über Relationen wie die Anzahl der Tupel einer Relation und die Art der Speicherung (clusternd oder nicht-clusternd) sowie die Anzahl der verschiedenen Suchschlüsselwerte für einen Index. Diese Art der Information wird insbesondere vom Anfrage-Optimierer (Abschnitt 5.2) benötigt. Ferner gibt es in den Katalogen auch Informationen über die Namen autorisierter Benutzer, deren Passwörter und deren Zugriffsrechte.

Da in relationalen Datenbanken Relationen die wesentlichen Objekte sind, liegt es nahe, alle Meta-Daten in speziellen Relationen, den sogenannten *Systemrelationen* oder *Kata-*

logrelationen, abzuspeichern. In diesem Sinne kann man den Systemkatalog als eine Art „Miniatur-Datenbank“ (Bild 1.1) verstehen. Wichtig ist, daß Katalogrelationen alle Relationen in einer Datenbank beschreiben, einschließlich der Katalogrelationen selbst. Diese Tatsache erlaubt es, mit den Möglichkeiten eines DBMS auf sie zuzugreifen. Mit Hilfe der Anfragesprache eines DBMS können dann, wie auf jeder anderen Relation auch, Anfragen gestellt werden. Hierbei ist natürlich insbesondere Wert auf eine entsprechende Zugriffskontrolle zu legen, da nicht jedem Datenbanknutzer auf jede Kataloginformation Zugriff gewährt werden sollte.

Selbsttestaufgabe 6. Geben Sie bezüglich Relationen, Attributen, Indexen, Sichten und Benutzern einen beispielhaften Schemaentwurf für Katalogrelationen an.

Literaturhinweise

Literaturhinweise zu Kapitel 1

Zu Datenbanksystemen allgemein gibt es eine Vielzahl von Büchern, von denen hier nur einige erwähnt werden können. Diese Bücher decken weitgehend den Gesamtbereich der Datenbanktechnologie ab und behandeln auch Architektur- und Implementierungsaspekte. Wichtige „Klassiker“ der Datenbankliteratur sind sicherlich das Werk von Date (2003), das weltweit wohl das am meisten verkaufte und eher praktisch orientierte Buch in diesem Bereich ist, und das zweibändige Werk von Ullman (1988, 1989), dessen Gewichtung mehr auf den theoretischen Grundlagen von Datenbanksystemen liegt. Weitere gute Darstellungen finden sich in den Büchern von Garcia-Molina et al. (2008), Elmasri & Navathe (2006) und Silberschatz *et al.* (2010). Gute deutsche Bücher sind Kemper & Eickler (2009), Saake et al. (2008) und Vossen (2008).

Der Schwerpunkt in allen genannten Büchern liegt auf relationalen Datenbanksystemen. Überblickhaft behandelt werden allerdings auch neuere Technologien wie objekt-orientierte, objekt-relationale, wissensbasierte, deduktive, geometrische, temporale, statistische und wissenschaftliche Datenbanksysteme, Entscheidungsunterstützungssysteme, Netzwerkdatenbanksysteme, Hauptspeicherdatenbanken, geographische Informationssysteme, Bild- und Video-Datenbanken und Textdatenbanken.

Thema dieses Kurses ist die Implementierung von Datenbanksystemen. Implementierungstechniken werden besonders in den Büchern von Garcia-Molina et al. (2008), Saake et al. (2005), sowie in Härder & Rahm (2001) behandelt. Dabei ist Härder & Rahm (2001) eine überarbeitete und aktualisierte Fassung der Kapitel 3 und 4 des sog. Datenbank-Handbuchs (Lockemann & Schmidt (1993)). Dieser Kurs hatte sich ursprünglich in Teilen an der Darstellung im Datenbank-Handbuch orientiert.

Es gibt auch einige wichtige Fachzeitschriften zum Thema Datenbanken wie z.B.

ACM Transactions on Database Systems (*TODS*)
IEEE Transactions on Knowledge and Data Engineering
Information Systems
The VLDB Journal
ACM SIGMOD Record

Die wichtigsten regelmäßigen Konferenzen sind

ACM SIGMOD International Conference on Management of Data (*SIGMOD*)
International Conference on Very Large Databases (*VLDB*)
International Conference on Data Engineering (*ICDE*)
European Conference on Database Technology (*EDBT*)
Datenbanksysteme für Büro, Technik, Wissenschaft (*BTW*)

Das 3-Ebenen-Modell wurde 1975 in einem Zwischenbericht (ANSI 1975) und 1978 in einem Endbericht (Tsichritzis & Klug (1978)) von der ANSI/SPARC-Arbeitsgruppe aufgestellt und beschrieben. Ziel der Arbeitsgruppe war es, Standardisierungsvorschläge für geeignete Bereiche der Datenbanktechnologie zu entwickeln. Die Arbeitsgruppe kam zum Schluß, daß Schnittstellen die einzige Komponente eines Datenbanksystems sind, die für eine Standardisierung geeignet sind. Daher wurde eine verallgemeinerte Datenbankarchitektur entworfen, die die Bedeutung solcher Schnittstellen betont. Trotz seines Alters hat dieses Modell immer noch Bestand. Beschreibungen dieses Modells befinden sich natürlich auch in allen oben genannten Lehrbüchern.

Das 3-Ebenen-Modell betont insbesondere die Aspekte der Datenunabhängigkeit, der Integration von Datenbeständen und der benutzerspezifizischen Sicht auf Teile der Datenbank. Ein anderes Modell, die sogenannte *Fünf-Schichten-Architektur* (Lockemann & Schmidt (1993), Härder & Rahm (2001)), setzt andere Schwerpunkte und betont den Schichten- und den Schnittstellenaspekt. Es ähnelt ein wenig der in Abschnitt 1.4 beschriebenen Softwarearchitektur eines DBMS. Ein Datenbanksystem wird in fünf Schichten zerlegt, wobei eine Schicht mittels einer Schnittstelle der direkt über ihr liegenden Schicht Objekte und Operatoren zu deren Realisierung zur Verfügung stellt. Die Struktur der Objekte und die Realisierung der Operatoren sind nach außen hin nicht sichtbar. Die fünf Schichten sind (1) Externspeicherverwaltung, (2) Systempufferverwaltung, (3) Record-Manager, Zugriffspfadverwaltung, Sperrverwaltung, Recovery-Komponente, (4) Systemkatalog, Transaktionsverwaltung und (5) Zugriffspfadoptimierung, Zugriffskontrolle, Integritätskontrolle. Sechs Schnittstellen werden unterschieden: (1) die Geräteschnittstelle, (2) die Dateischnittstelle, (3) die Systempufferschnittstelle, (4) die interne Satzschnittstelle, (5) die satzorientierte Schnittstelle und (6) die mengenorientierte Schnittstelle.

Die Architektur von DBMS aus Software-Sicht wird von Elmasri & Navathe (2006) und Silberschatz *et al.* (2010) allerdings nur in groben Zügen behandelt.

Literaturhinweise zu Kapitel 2

Wiederholt (1989) gibt eine sehr umfassende und ausführliche Darstellung und Analyse verschiedenster Dateiorganisationen (und externer Speichermedien). Ferner finden sich Diskussionen in den meisten der in den Literaturhinweisen zu Kapitel 1 genannten Textbücher. Smith und Barnes (1990) stellen ebenfalls Dateiorganisationen und Zugriffsmethoden vor. Die Hashing-Methode und insbesondere auch verschiedene Arten von Hash-Funktionen werden, da es auch eine interne Variante im Hauptspeicher gibt, in vielen Büchern über Algorithmen und Datenstrukturen, z.B. von Ottmann und Widmayer (2002), besprochen.

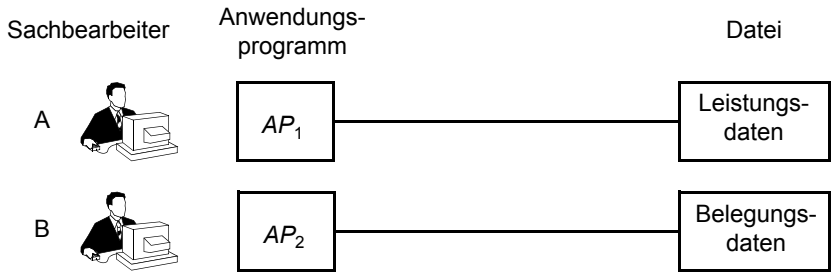
Die Aufgaben und mögliche Implementierungen von Systemkatalogen werden im Datenbank-Handbuch (Lockemann & Schmidt (1993)) beschrieben.

Aspekte der Systempufferverwaltung (wie z.B. auch das Schattenspeicher-Konzept) werden ziemlich ausführlich im Datenbank-Handbuch (Lockemann & Schmidt (1993), bzw. Härder & Rahm (2001)) dargestellt. Ein Überblick über Systempuffer und Ersetzungsalgorithmen in DBMS wird in einem Artikel von Effelsberg & Härder (1984) gegeben. Stonebraker (1981) diskutiert die Beziehung zwischen Datenbanksystempuffer-Managern und Betriebssystempuffer-Managern. Speichermanagement und Ersetzungsalgorithmen allgemein werden in der Betriebssystemliteratur behandelt, z.B. von Tanenbaum (2005).

Lösungsvorschläge zu den Selbsttestaufgaben

Selbsttestaufgabe 1 (Seite 3)

Folgendes Szenario ist vorstellbar:



Wir nehmen an, daß beide Dateien den Namen, den Vornamen, die Matrikelnummer sowie die Adresse eines Studenten enthalten. Leistungsdaten beinhalten ferner Daten über die Ergebnisse mündlicher Prüfungen, Klausuren, Seminare usw. Belegungsdaten umfassen zusätzlich Daten über die belegten Kurse/Vorlesungen jedes Studenten. Das Auftreten von *Redundanz* ist offensichtlich: Name, Vorname, Matrikelnummer und Adresse eines Studenten werden mehrfach gespeichert, was auf Speicherverschwendung hinausläuft. Ferner besteht die Gefahr, daß Dateien, die gleiche Daten enthalten, inkonsistent werden. Dies kann geschehen, wenn eine Änderung von Daten (hier zum Beispiel die Änderung einer Adresse) nur auf einigen aber nicht allen Dateien durchgeführt wird. *Inkonsistenz* kann auch auf andere Weise entstehen. Auch wenn eine Änderung auf allen betroffenen Dateien erfolgt, können diese geänderten Daten inkonsistent sein, weil die Änderungen dezentral und unabhängig voneinander ausgeführt werden und durch Eingabefehler der Anwender voneinander verschieden sein können.

Die *Daten-Programm-Abhängigkeit* zeigt sich in folgenden beiden Beispielen. Nehmen wir an, der Belegungsdatendatei soll für jeden Datensatz ein zusätzliches Datum (beispielsweise ein boolscher Wert, der angibt, ob die geforderten Leistungen für das Vordiplom bereits erbracht wurden) hinzugefügt werden, so ist es extrem aufwendig, eine Reorganisation dieser Datei herbeizuführen. Ist die Leistungsdatei ausschließlich als Index organisiert, weil im Anwendungsprogramm AP_1 stets über den Schlüssel „Matrikelnum-

mer“ auf die Daten eines Studenten zugegriffen wird, so ist es schwierig, eine Übersicht über die Leistungsdaten aller Studenten zu erstellen, da dies einen sequentiellen Zugriff und eine Betrachtung aller Datensätze erfordert.

Nehmen wir an, daß Sachbearbeiter *C* aus der Finanzabteilung beauftragt wird, die Zahlung der Semesterbeiträge aller Studenten zu verwalten. Hierzu muß zunächst ein Anwendungsprogramm AP_3 und die Struktur einer darauf abgestimmten Datei entwickelt werden. Genaugenommen muß Sachbearbeiter *C* danach alle persönlichen Daten wie Name, Vorname, Matrikelnummer und Adresse eines Studenten erfassen, da er aus Datenschutzgründen eine Kopie der Leistungsdatendatei oder der Belegungsdatendatei nicht erhalten darf. Hierin stehen nämlich auch noch weitere, für Sachbearbeiter *C* nicht bestimmte Daten. In dem gesamten Verfahren zeigt sich die extreme *Inflexibilität*.

Selbsttestaufgabe 2 (Seite 9)

Bei all den Vorteilen, die Datenbanksysteme haben, gibt es auch einige Nachteile. Ein DBMS ist ein komplexes Softwarepaket, und die hohen Anfangskosten und Hardwareanforderungen sowie das erforderliche Training zur Benutzung des Systems müssen berücksichtigt werden. In bestimmten Anwendungsbereichen kann die Leistungsfähigkeit eines DBMS nicht ausreichen. Typischerweise geschieht dies in Umgebungen mit Realzeitanforderungen, wo Operationen innerhalb eines kleinen Zeitintervalls ausgeführt werden müssen. Aus solchen Gründen können ad hoc- oder spezialisierte Lösungen einem DBMS vorzuziehen sein, insbesondere dann, wenn gewisse Vorteile von DBMS wie z.B. das Stellen von Anfragen, Recovery, Datensicherheit usw. nicht erforderlich sind. Auch wenn Datenbank und Anwendungen einfach strukturiert, wohl definiert und keinen oder kaum Änderungen unterworfen sind, kann der Verzicht auf ein DBMS ratsam sein. Dies kann auch gelten, wenn kein Mehrfachzugriff und keine Mehrbenutzerfähigkeit erforderlich ist.

Selbsttestaufgabe 3 (Seite 26)

Unter Beachtung der Ausrichtung von Feldwerten ergibt sich die folgende Datensatzgröße:

$$(24+0)+(9+0)+(1+2)+(4+0)+(14+2)+(4+0)+(4+0)+(18+2) = 84 \text{ Bytes}.$$

Hiervon enthalten 6 Bytes keine Information. Eine kompaktere Darstellung ist bei Vertauschung der Reihenfolge der Feldwerte möglich. Man kann sich vorstellen, daß das Datenbanksystem aus dem benutzergegebenen Datensatzformat (z.B. einem Schema

einer Relation) ein speicherplatzoptimiertes Datensatzformat generiert. Nachteil ist ein höherer Verwaltungsaufwand. Das Datensatzformat ist dann z.B.

Student(*Name*: *String*[24]; *Semesteranzahl*: *Integer*; *Nr*: *Integer*;
Plz: *Integer*; *Straße*: *String*[14]; *Ort*: *String*[18];
Matrikelnr: *String*[9]; *Geschlecht*: *Boolean*)

Als Datensatzgröße ergibt sich dann:

(24+0)+(4+0)+(4+0)+(4+0)+(14+0)+(18+0)+(9+0)+(1+2) = 80 Bytes.

Hiervon enthalten 2 Bytes keine Information.

Selbsttestaufgabe 4 (Seite 40)

Zunächst wird eine binäre Suche auf den Seiten und danach auf den Datensätzen der gefundenen Seite durchgeführt. Wir nehmen an, daß die *b* Seiten der Datei von 1 bis *b* durchnumeriert sind und daß die Datensätze gemäß des Sortierfelds aufsteigend geordnet sind. Gesucht wird nach einem Datensatz, dessen Sortierfeldwert gleich *k* ist.

```
algorithm search(k : key_type) : boolean;
   l := 1; r := b
   while r ≥ l do
      i := (l+r) div 2;
      Lese Seite (Block) i der Datei in den Puffer;
      if k < Sortierfeldwert des ersten Datensatzes in der Seite then r := i–1
      else if k > Sortierfeldwert des letzten Datensatzes in der Seite then l := i+1
      else { Seite gefunden }
         c := 1; d := Anzahl der Datensätze der Seite;
         while d ≥ c do
            j := (c+d) div 2;
            if k < Sortierfeldwert des j-ten Datensatzes in der Seite then d := j–1
            else if k > Sortierfeldwert des j-ten Datensatzes in der Seite then c := j+1
            else { Datensatz gefunden }
               return true
            fi
         end
      fi
   od
   return false
end search.
```

Selbsttestaufgabe 5 (Seite 41)

Beim Einfügen eines Datensatzes in eine Datei wird zunächst die richtige Seite gesucht. Befindet sich auf dieser Seite noch genügend Platz, so wird der neue Datensatz an der richtigen Position unter vorhergehendem Verschieben der nachfolgenden Datensätze eingefügt. Anderenfalls wird der Datensatz am Ende der Überlaufdatei eingefügt. Dies spart in wesentlichem Maße Zeit, da sonst alle Datensätze hinter der korrekten Einfügeposition für den neuen Datensatz verschoben werden müßten. Nachteilig ist, daß, falls beim Suchen der Datensatz nicht an der richtigen Position gefunden wird, zusätzlich noch die Überlaufdatei linear nach dem Datensatz durchsucht werden muß. Ähnliches gilt für das Löschen eines Datensatzes. Beim Ändern eines Feldwertes ist zu unterscheiden, ob das Sortierfeld oder ein anderes Feld modifiziert werden soll. Wird das Sortierfeld geändert, so ist dies gleichbedeutend mit einer Veränderung der Position des Datensatzes in der Datei. Die Änderungsoperation entspricht dann dem Löschen des alten Datensatzes gefolgt von dem Einfügen des modifizierten Datensatzes. Soll ein Nichtsortierfeld geändert werden, so wird der Datensatz gesucht, der entsprechende Feldwert geändert und der Datensatz an die gleiche physische Position zurückgeschrieben.

Hat die Überlaufdatei eine gewisse Größe erreicht, so ist eine Reorganisation durchzuführen, in der Haupt- und Überlaufdatei miteinander verschmolzen werden. Hierzu muß die Überlaufdatei zunächst sortiert werden. Ferner werden Datensätze, die mit einer Löschmarkierung versehen sind, entfernt.

Selbsttestaufgabe 6 (Seite 46)

Die Wahl der Katalogrelationen und ihrer Schemata ist nicht eindeutig. Ein beispielhafter Entwurf ist folgender:

Relationen(*RelName*: *string*, *AnzahlAttribute*: *integer*)

Attribute(*AttrName*: *string*, *RelName*: *string*, *TypName*: *string*, *Position*: *integer*)

Indexe(*IndexName*: *string*, *RelName*: *string*, *IndexTyp*: *string*, *IndexAttribute*: *string*)

Sichten(*SichtenName*: *string*, *Definition*: *string*)

Benutzer(*Benutzername*: *string*, *KodiertesPasswort*: *string*, *Gruppe*: *integer*)

Literatur

ANSI (1975). Study Group on Data Base Management Systems: Interim Report. *FDT* (*ACM SIGMOD bulletin*) 7(2).

DATE C. J. (2003). *An Introduction to Database Systems*. Eighth Edition, Addison-Wesley.

EFFELSBERG W. & HÄRDER T. (1984). Principles of Database Buffer Management. *ACM Transactions on Database Systems* (*TODS*), 9(4), 560-595.

ELMASRI R. & NAVATHE S. B. (2006). *Fundamentals of Database Systems*. Fifth Edition, Addison-Wesley.

GARCIA-MOLINA, H., ULLMAN, J.D. & WIDOM, J. (2008). *Database Systems: The Complete Book*. 2nd Edition, Prentice-Hall.

HÄRDER, T. & RAHM, E. (2001). *Datenbanksysteme: Konzepte und Techniken der Implementierung.* 2. Aufl., Springer-Verlag.

KEMPER, A. & EICKLER, A. (2009). *Datenbanksysteme: Eine Einführung.* 7. Aufl., Oldenbourg-Verlag.

LOCKEMANN P.C. & SCHMIDT J.W. (1993). *Datenbank-Handbuch*. Informatik-Handbücher, Springer-Verlag.

OTTMANN T. & WIDMAYER P. (2002). *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag.

SAAKE, G., HEUER, A. & SATTLER, K.-U. (2005). *Datenbanken: Implementierungstechniken.* 2. Aufl., mitp-Verlag, Heidelberg.

SAAKE, G., SATTLER, K.-U. & HEUER, A. (2008). *Datenbanken: Konzepte & Sprachen.* 3. Aufl., mitp-Verlag, Heidelberg.

SILBERSCHATZ A., KORTH H.F. & SUDARSHAN, S. (2010). *Database System Concepts*. Sixth Edition. McGraw-Hill.

SMITH P.D. & BARNES G.M. (1990). *Files and Databases: An Introduction*. Addison-Wesley.

STONEBRAKER M. (1981). Operating System Support for Database Management. *Communications of the ACM*, 24(7), 412-418.

TANENBAUM A.S. (2005). *Moderne Betriebssysteme*. Pearson Studium.

TSICHRITZIS D. C. & KLUG A. (1978). The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems. *Information Systems*, Volume 3.

ULLMAN J. D. (1988). *Principles of Database and Knowledge-Base Systems*, Volume I: Classical Database Systems. Computer Science Press.

ULLMAN J. D. (1989). *Principles of Database and Knowledge-Base Systems*, Volume II: The New Technologies. Computer Science Press.

VOSSEN, G. (2008). *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme.* 5. Aufl., Oldenbourg-Verlag.

WIEDERHOLT G. (1989). *Dateiorganisation in Datenbanken*. McGraw-Hill.

Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z